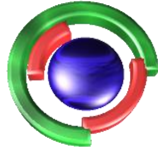
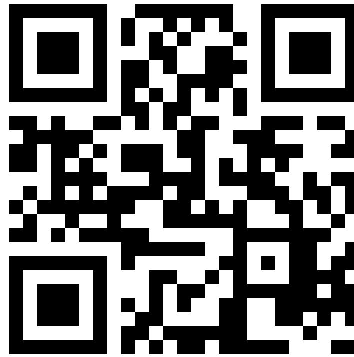
FUTURE VISION BIE

One Stop for All Study Materials & Lab Programs



By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

5 Syntax-Directed Translation **303**
5.1 Syntax-Directed Definitions 304
5.1.1 Inherited and Synthesized Attributes 304
5.1.2 Evaluating an SDD at the Nodes of a Parse Tree 306
5.1.3 Exercises for Section 5.1 309
5.2 Evaluation Orders for SDD's 310
5.2.1 Dependency Graphs 310
5.2.2 Ordering the Evaluation of Attributes 312
5.2.3 S-Attributed Definitions 312
5.2.4 L-Attributed Definitions 313
5.2.5 Semantic Rules with Controlled Side Effects 314
5.2.6 Exercises for Section 5.2 317
5.3 Applications of Syntax-Directed Translation 318
5.3.1 Construction of Syntax Trees 318
5.3.2 The Structure of a Type 321
5.3.3 Exercises for Section 5.3 323
5.4 Syntax-Directed Translation Schemes 324
5.4.1 Postfix Translation Schemes 324
5.4.2 Parser-Stack Implementation of Postfix SDT's 325
5.4.3 SDT's With Actions Inside Productions 327
5.4.4 Eliminating Left Recursion From SDT's 328
5.4.5 SDT's for L-Attributed Definitions 331
5.4.6 Exercises for Section 5.4 336
5.5 Implementing L-Attributed SDD's 337
5.5.1 Translation During Recursive-Descent Parsing 338
5.5.2 On-The-Fly Code Generation 340
5.5.3 L-Attributed SDD's and LL Parsing 343
5.5.4 Bottom-Up Parsing of L-Attributed SDD's 348
5.5.5 Exercises for Section 5.5 352
5.6 Summary of Chapter 5 353
5.7 References for Chapter 5 354

6 Intermediate-Code Generation **357**
6.1 Variants of Syntax Trees 358
6.1.1 Directed Acyclic Graphs for Expressions 359
6.1.2 The Value-Number Method for Constructing DAG's . . . 360
6.1.3 Exercises for Section 6.1 362
6.2 Three-Address Code 363
6.2.1 Addresses and Instructions 364
6.2.2 Quadruples 366
6.2.3 Triples 367
6.2.4 Static Single-Assignment Form 369
6.2.5 Exercises for Section 6.2 370
6.3 Types and Declarations 370
6.3.1 Type Expressions 371

https://hemanthrajhemu.github.io

6.3.2 Type Equivalence . . . 372
6.3.3 Declarations . . . 373
6.3.4 Storage Layout for Local Names . . . 373
6.3.5 Sequences of Declarations . . . 376
6.3.6 Fields in Records and Classes . . . 376
6.3.7 Exercises for Section 6.3 . . . 378

6.4 Translation of Expressions . . . 378
6.4.1 Operations Within Expressions . . . 378
6.4.2 Incremental Translation . . . 380
6.4.3 Addressing Array Elements . . . 381
6.4.4 Translation of Array References . . . 383
6.4.5 Exercises for Section 6.4 . . . 384

6.5 Type Checking . . . 386
6.5.1 Rules for Type Checking . . . 387
6.5.2 Type Conversions . . . 388
6.5.3 Overloading of Functions and Operators . . . 390
6.5.4 Type Inference and Polymorphic Functions . . . 391
6.5.5 An Algorithm for Unification . . . 395
6.5.6 Exercises for Section 6.5 . . . 398

6.6 Control Flow . . . 399
6.6.1 Boolean Expressions . . . 399
6.6.2 Short-Circuit Code . . . 400
6.6.3 Flow-of-Control Statements . . . 401
6.6.4 Control-Flow Translation of Boolean Expressions . . . 403
6.6.5 Avoiding Redundant Gotos . . . 405
6.6.6 Boolean Values and Jumping Code . . . 408
6.6.7 Exercises for Section 6.6 . . . 408

6.7 Backpatching . . . 410
6.7.1 One-Pass Code Generation Using Backpatching . . . 410
6.7.2 Backpatching for Boolean Expressions . . . 411
6.7.3 Flow-of-Control Statements . . . 413
6.7.4 Break-, Continue-, and Goto-Statements . . . 416
6.7.5 Exercises for Section 6.7 . . . 417

6.8 Switch-Statements . . . 418
6.8.1 Translation of Switch-Statements . . . 419
6.8.2 Syntax-Directed Translation of Switch-Statements . . . 420
6.8.3 Exercises for Section 6.8 . . . 421

6.9 Intermediate Code for Procedures . . . 422

6.10 Summary of Chapter 6 . . . 424

6.11 References for Chapter 6 . . . 425

https://hemanthrajhemu.github.io

7 Run-Time Environments **427**
7.1 Storage Organization . . . 427
7.1.1 Static Versus Dynamic Storage Allocation . . . 429
7.2 Stack Allocation of Space . . . 430
7.2.1 Activation Trees . . . 430
7.2.2 Activation Records . . . 433
7.2.3 Calling Sequences . . . 436
7.2.4 Variable-Length Data on the Stack . . . 438
7.2.5 Exercises for Section 7.2 . . . 440
7.3 Access to Nonlocal Data on the Stack . . . 441
7.3.1 Data Access Without Nested Procedures . . . 442
7.3.2 Issues With Nested Procedures . . . 442
7.3.3 A Language With Nested Procedure Declarations . . . 443
7.3.4 Nesting Depth . . . 443
7.3.5 Access Links . . . 445
7.3.6 Manipulating Access Links . . . 447
7.3.7 Access Links for Procedure Parameters . . . 448
7.3.8 Displays . . . 449
7.3.9 Exercises for Section 7.3 . . . 451
7.4 Heap Management . . . 452
7.4.1 The Memory Manager . . . 453
7.4.2 The Memory Hierarchy of a Computer . . . 454
7.4.3 Locality in Programs . . . 455
7.4.4 Reducing Fragmentation . . . 457
7.4.5 Manual Deallocation Requests . . . 460
7.4.6 Exercises for Section 7.4 . . . 463
7.5 Introduction to Garbage Collection . . . 463
7.5.1 Design Goals for Garbage Collectors . . . 464
7.5.2 Reachability . . . 466
7.5.3 Reference Counting Garbage Collectors . . . 468
7.5.4 Exercises for Section 7.5 . . . 470
7.6 Introduction to Trace-Based Collection . . . 470
7.6.1 A Basic Mark-and-Sweep Collector . . . 471
7.6.2 Basic Abstraction . . . 473
7.6.3 Optimizing Mark-and-Sweep . . . 475
7.6.4 Mark-and-Compact Garbage Collectors . . . 476
7.6.5 Copying collectors . . . 478
7.6.6 Comparing Costs . . . 482
7.6.7 Exercises for Section 7.6 . . . 482
7.7 Short-Pause Garbage Collection . . . 483
7.7.1 Incremental Garbage Collection . . . 483
7.7.2 Incremental Reachability Analysis . . . 485
7.7.3 Partial-Collection Basics . . . 487
7.7.4 Generational Garbage Collection . . . 488
7.7.5 The Train Algorithm . . . 490

https://hemanthrajhemu.github.io

7.7.6 Exercises for Section 7.7 493
7.8 Advanced Topics in Garbage Collection 494
7.8.1 Parallel and Concurrent Garbage Collection 495
7.8.2 Partial Object Relocation 497
7.8.3 Conservative Collection for Unsafe Languages 498
7.8.4 Weak References 498
7.8.5 Exercises for Section 7.8 499
7.9 Summary of Chapter 7 500
7.10 References for Chapter 7 502

8 Code Generation **505**
8.1 Issues in the Design of a Code Generator 506
8.1.1 Input to the Code Generator 507
8.1.2 The Target Program 507
8.1.3 Instruction Selection 508
8.1.4 Register Allocation 510
8.1.5 Evaluation Order 511
8.2 The Target Language 512
8.2.1 A Simple Target Machine Model 512
8.2.2 Program and Instruction Costs 515
8.2.3 Exercises for Section 8.2 516
8.3 Addresses in the Target Code 518
8.3.1 Static Allocation 518
8.3.2 Stack Allocation 520
8.3.3 Run-Time Addresses for Names 522
8.3.4 Exercises for Section 8.3 524
8.4 Basic Blocks and Flow Graphs 525
8.4.1 Basic Blocks 526
8.4.2 Next-Use Information 528
8.4.3 Flow Graphs 529
8.4.4 Representation of Flow Graphs 530
8.4.5 Loops 531
8.4.6 Exercises for Section 8.4 531
8.5 Optimization of Basic Blocks 533
8.5.1 The DAG Representation of Basic Blocks 533
8.5.2 Finding Local Common Subexpressions 534
8.5.3 Dead Code Elimination 535
8.5.4 The Use of Algebraic Identities 536
8.5.5 Representation of Array References 537
8.5.6 Pointer Assignments and Procedure Calls 539
8.5.7 Reassembling Basic Blocks From DAG's 539
8.5.8 Exercises for Section 8.5 541
8.6 A Simple Code Generator 542
8.6.1 Register and Address Descriptors 543
8.6.2 The Code-Generation Algorithm 544

https://hemanthrajhemu.github.io

8.6.3 Design of the Function *getReg* . . . 547
8.6.4 Exercises for Section 8.6 . . . 548
8.7 Peephole Optimization . . . 549
8.7.1 Eliminating Redundant Loads and Stores . . . 550
8.7.2 Eliminating Unreachable Code . . . 550
8.7.3 Flow-of-Control Optimizations . . . 551
8.7.4 Algebraic Simplification and Reduction in Strength . . . 552
8.7.5 Use of Machine Idioms . . . 552
8.7.6 Exercises for Section 8.7 . . . 553
8.8 Register Allocation and Assignment . . . 553
8.8.1 Global Register Allocation . . . 553
8.8.2 Usage Counts . . . 554
8.8.3 Register Assignment for Outer Loops . . . 556
8.8.4 Register Allocation by Graph Coloring . . . 556
8.8.5 Exercises for Section 8.8 . . . 557
8.9 Instruction Selection by Tree Rewriting . . . 558
8.9.1 Tree-Translation Schemes . . . 558
8.9.2 Code Generation by Tiling an Input Tree . . . 560
8.9.3 Pattern Matching by Parsing . . . 563
8.9.4 Routines for Semantic Checking . . . 565
8.9.5 General Tree Matching . . . 565
8.9.6 Exercises for Section 8.9 . . . 567
8.10 Optimal Code Generation for Expressions . . . 567
8.10.1 Ershov Numbers . . . 567
8.10.2 Generating Code From Labeled Expression Trees . . . 568
8.10.3 Evaluating Expressions with an Insufficient Supply of Registers . . . 570
8.10.4 Exercises for Section 8.10 . . . 572
8.11 Dynamic Programming Code-Generation . . . 573
8.11.1 Contiguous Evaluation . . . 574
8.11.2 The Dynamic Programming Algorithm . . . 575
8.11.3 Exercises for Section 8.11 . . . 577
8.12 Summary of Chapter 8 . . . 578
8.13 References for Chapter 8 . . . 579

9 Machine-Independent Optimizations 583
9.1 The Principal Sources of Optimization . . . 584
9.1.1 Causes of Redundancy . . . 584
9.1.2 A Running Example: Quicksort . . . 585
9.1.3 Semantics-Preserving Transformations . . . 586
9.1.4 Global Common Subexpressions . . . 588
9.1.5 Copy Propagation . . . 590
9.1.6 Dead-Code Elimination . . . 591
9.1.7 Code Motion . . . 592
9.1.8 Induction Variables and Reduction in Strength . . . 592

https://hemanthrajhemu.github.io

Chapter 5

Syntax-Directed Translation

This chapter develops the theme of Section 2.3: the translation of languages guided by context-free grammars. The translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. The techniques are also useful for implementing little languages for specialized tasks; this chapter includes an example from typesetting.

We associate information with a language construct by attaching *attributes* to the grammar symbol(s) representing the construct, as discussed in Section 2.3.2. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE	
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$	(5.1)

This production has two nonterminals, E and T; the subscript in E_1 distinguishes the occurrence of E in the production body from the occurrence of E as the head. Both E and T have a string-valued attribute *code*. The semantic rule specifies that the string $E.code$ is formed by concatenating $E_1.code$, $T.code$, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E_1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

From Section 2.3.5, a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \rightarrow E_1 + T \quad \{ \text{ print } '+' \ \} \tag{5.2}$$

By convention, semantic actions are enclosed within curly braces. (If curly braces occur as grammar symbols, we enclose them within single quotes, as in

https://hemanthrajhemu.github.io

'{' and '}'.) The position of a semantic action in a production body determines the order in which the action is executed. In production (5.2), the action occurs at the end, after all the grammar symbols; in general, semantic actions may occur at any position in a production body.

Between the two notations, syntax-directed definitions can be more readable, and hence more useful for specifications. However, translation schemes can be more efficient, and hence more useful for implementations.

The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree. In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called "L-attributed translations" (L for left-to-right), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called "S-attributed translations" (S for synthesized), which can be performed easily in connection with a bottom-up parse.

5.1 Syntax-Directed Definitions

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If X is a symbol and a is one of its attributes, then we write $X.a$ to denote the value of a at a particular parse-tree node labeled X. If we implement the nodes of the parse tree by records or objects, then the attributes of X can be implemented by data fields in the records that represent the nodes for X. Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

5.1.1 Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for nonterminals:

1. A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

2. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

https://hemanthrajhemu.github.io

An Alternative Definition of Inherited Attributes

No additional translations are enabled if we allow an inherited attribute $B.c$ at a node N to be defined in terms of attribute values at the children of N, as well as at N itself, at its parent, and at its siblings. Such rules can be "simulated" by creating additional attributes of B, say $B.c_1, B.c_2, \ldots$. These are synthesized attributes that copy the needed attributes of the children of the node labeled B. We then compute $B.c$ as an inherited attribute, using the attributes $B.c_1, B.c_2, \ldots$ in place of attributes at the children. Such attributes are rarely needed in practice.

While we do not allow an inherited attribute at node N to be defined in terms of attribute values at the children of node N, we do allow a synthesized attribute at node N to be defined in terms of inherited attribute values at node N itself.

Terminals can have synthesized attributes, but not inherited attributes. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Example 5.1: The SDD in Fig. 5.1 is based on our familiar grammar for arithmetic expressions with operators + and $*$. It evaluates expressions terminated by an endmarker **n**. In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Figure 5.1: Syntax-directed definition of a simple desk calculator

The rule for production 1, $L \to E$ **n**, sets $L.val$ to $E.val$, which we shall see is the numerical value of the entire expression.

Production 2, $E \to E_1 + T$, also has one rule, which computes the *val* attribute for the head E as the sum of the values at E_1 and T. At any parse-

https://hemanthrajhemu.github.io

tree node N labeled E, the value of *val* for E is the sum of the values of *val* at the children of node N labeled E and T.

Production 3, $E \rightarrow T$, has a single rule that defines the value of *val* for E to be the same as the value of *val* at the child for T. Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them. The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives $F.val$ the value of a digit, that is, the numerical value of the token **digit** that the lexical analyzer returned. □

An SDD that involves only synthesized attributes is called *S-attributed*; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

For simplicity, the examples in this section have semantic rules without side effects. In practice, it is convenient to allow SDD's to have limited side effects, such as printing the result computed by a desk calculator or interacting with a symbol table. Once the order of evaluation of attributes is discussed in Section 5.2, we shall allow semantic rules to compute arbitrary functions, possibly involving side effects.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser. In fact, the SDD in Fig. 5.1 mirrors the Yacc program of Fig. 4.58, which illustrates translation during LR parsing. The difference is that, in the rule for production 1, the Yacc program prints the value $E.val$ as a side effect, instead of defining the attribute $L.val$.

An SDD without side effects is sometimes called an *attribute grammar*. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

5.1.2 Evaluating an SDD at the Nodes of a Parse Tree

To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree. Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree*.

How do we construct an annotated parse tree? In what order do we evaluate attributes? Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends. For example, if all attributes are synthesized, as in Example 5.1, then we must evaluate the *val* attributes at all of the children of a node before we can evaluate the *val* attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree; the evaluation of S-attributed definitions is discussed in Section 5.2.3.

https://hemanthrajhemu.github.io

For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes. For instance, consider nonterminals A and B, with synthesized and inherited attributes $A.s$ and $B.i$, respectively, along with the production and rules

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$ $B.i = A.s + 1$

These rules are circular; it is impossible to evaluate either $A.s$ at a node N or $B.i$ at the child of N without first evaluating the other. The circular dependency of $A.s$ and $B.i$ at some pair of nodes in a parse tree is suggested by Fig. 5.2.

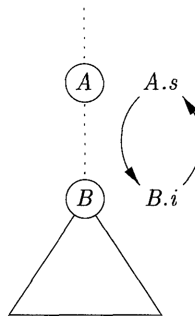


Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

It is computationally difficult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate.[1] Fortunately, there are useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists, as we shall see in Section 5.2.

Example 5.2: Figure 5.3 shows an annotated parse tree for the input string $3 * 5 + 4$ **n**, constructed using the grammar and rules of Fig. 5.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled $*$, after computing $T.val = 3$ and $F.val = 5$ at its first and third children, we apply the rule that says $T.val$ is the product of these two values, or 15. □

Inherited attributes are useful when the structure of a parse tree does not "match" the abstract syntax of the source code. The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation.

[1] Without going into details, while the problem is decidable, it cannot be solved by a polynomial-time algorithm, even if $\mathcal{P} = \mathcal{NP}$, since it has exponential time complexity.

https://hemanthrajhemu.github.io

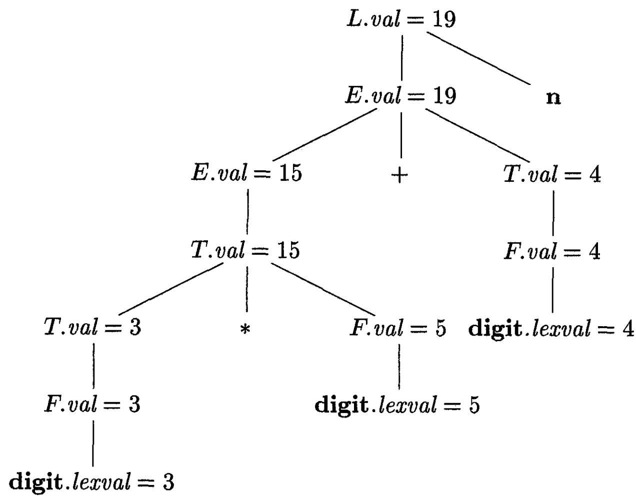


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

Example 5.3: The SDD in Fig. 5.4 computes terms like $3 * 5$ and $3 * 5 * 7$. The top-down parse of input $3 * 5$ begins with the production $T \to F\,T'$. Here, F generates the digit 3, but the operator $*$ is generated by T'. Thus, the left operand 3 appears in a different subtree of the parse tree from $*$. An inherited attribute will therefore be used to pass the operand to the operator.

The grammar in this example is an excerpt from a non-left-recursive version of the familiar expression grammar; we used such a grammar as a running example to illustrate top-down parsing in Section 4.4.

	PRODUCTION	SEMANTIC RULES
1)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to$ **digit**	$F.val =$ **digit**.$lexval$

Figure 5.4: An SDD based on a grammar suitable for top-down parsing

Each of the nonterminals T and F has a synthesized attribute *val*; the terminal **digit** has a synthesized attribute *lexval*. The nonterminal T' has two attributes: an inherited attribute *inh* and a synthesized attribute *syn*.

https://hemanthrajhemu.github.io

The semantic rules are based on the idea that the left operand of the operator $*$ is inherited. More precisely, the head T' of the production $T' \to * \; F \; T_1'$ inherits the left operand of $*$ in the production body. Given a term $x * y * z$, the root of the subtree for $* \, y * z$ inherits x. Then, the root of the subtree for $* \, z$ inherits the value of $x * y$, and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for $3 * 5$ in Fig. 5.5. The leftmost leaf in the parse tree, labeled **digit**, has attribute value *lexval* = 3, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, $F \to$ **digit**. The only semantic rule associated with this production defines $F.val =$ **digit**.*lexval*, which equals 3.

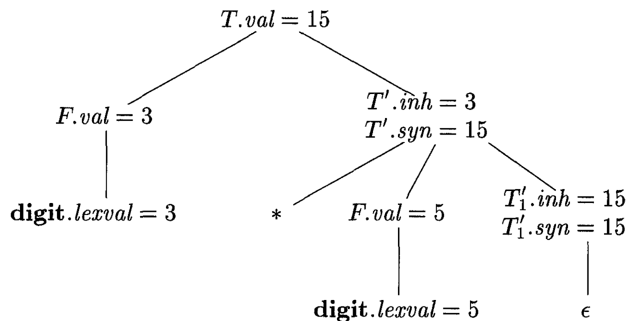


Figure 5.5: Annotated parse tree for $3 * 5$

At the second child of the root, the inherited attribute $T'.inh$ is defined by the semantic rule $T'.inh = F.val$ associated with production 1. Thus, the left operand, 3, for the $*$ operator is passed from left to right across the children of the root.

The production at the node for T' is $T' \to * \, F T_1'$. (We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T'.) The inherited attribute $T_1'.inh$ is defined by the semantic rule $T_1'.inh = T'.inh \times F.val$ associated with production 2.

With $T'.inh = 3$ and $F.val = 5$, we get $T_1'.inh = 15$. At the lower node for T_1', the production is $T' \to \epsilon$. The semantic rule $T'.syn = T'.inh$ defines $T_1'.syn = 15$. The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T, where $T.val = 15$. □

5.1.3 Exercises for Section 5.1

Exercise 5.1.1: For the SDD of Fig. 5.1, give annotated parse trees for the following expressions:

a) $(3 + 4) * (5 + 6)$ **n**.

https://hemanthrajhemu.github.io

b) $1 * 2 * 3 * (4 + 5)$ **n**.

c) $(9 + 8 * (7 + 6) + 5) * 4$ **n**.

Exercise 5.1.2: Extend the SDD of Fig. 5.4 to handle expressions as in Fig. 5.1.

Exercise 5.1.3: Repeat Exercise 5.1.1, using your SDD from Exercise 5.1.2.

5.2 Evaluation Orders for SDD's

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

In this section, in addition to dependency graphs, we define two important classes of SDD's: the "S-attributed" and the more general "L-attributed" SDD's. The translations specified by these two classes fit well with the parsing methods we have studied, and most translations encountered in practice can be written to conform to the requirements of at least one of these classes.

5.2.1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. In more detail:

- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.

- Suppose that a semantic rule associated with a production p defines the value of synthesized attribute $A.b$ in terms of the value of $X.c$ (the rule may define $A.b$ in terms of other attributes in addition to $X.c$). Then, the dependency graph has an edge from $X.c$ to $A.b$. More precisely, at every node N labeled A where production p is applied, create an edge to attribute b at N, from the attribute c at the child of N corresponding to this instance of the symbol X in the body of the production.[2]

- Suppose that a semantic rule associated with a production p defines the value of inherited attribute $B.c$ in terms of the value of $X.a$. Then, the dependency graph has an edge from $X.a$ to $B.c$. For each node N labeled B that corresponds to an occurrence of this B in the body of production p, create an edge to attribute c at N from the attribute a at the node M

[2]Since a node N can have several children labeled X, we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

https://hemanthrajhemu.github.io

that corresponds to this occurrence of X. Note that M could be either the parent or a sibling of N.

Example 5.4: Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute *val* at N is computed using the values of *val* at the two children, labeled E and T. Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid. □

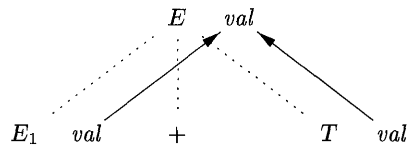


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Example 5.5: An example of a complete dependency graph appears in Fig. 5.7. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 5.5.

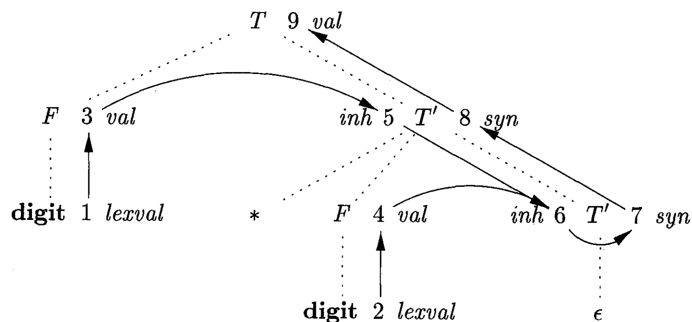


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled **digit**. Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled F. The edges to node 3 from 1 and to node 4 from 2 result

https://hemanthrajhemu.github.io

from the semantic rule that defines $F.val$ in terms of **digit**.*lexval*. In fact, $F.val$ equals **digit**.*lexval*, but the edge represents dependence, not equality.

Nodes 5 and 6 represent the inherited attribute $T'.inh$ associated with each of the occurrences of nonterminal T'. The edge to 5 from 3 is due to the rule $T'.inh = F.val$, which defines $T'.inh$ at the right child of the root from $F.val$ at the left child. We see edges to 6 from node 5 for $T'.inh$ and from node 4 for $F.val$, because these values are multiplied to evaluate the attribute *inh* at node 6.

Nodes 7 and 8 represent the synthesized attribute *syn* associated with the occurrences of T'. The edge to node 7 from 6 is due to the semantic rule $T'.syn = T'.inh$ associated with production 3 in Fig. 5.4. The edge to node 8 from 7 is due to a semantic rule associated with production 2.

Finally, node 9 represents the attribute $T.val$. The edge to 9 from 8 is due to the semantic rule, $T.val = T'.syn$, associated with production 1. □

5.2.2 Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency graph from N_i to N_j, then $i < j$. Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely find a node with no edge entering. For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes.

Example 5.6: The dependency graph of Fig. 5.7 has no cycles. One topological sort is the order in which the nodes have already been numbered: $1, 2, \ldots, 9$. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as $1, 3, 5, 2, 4, 6, 7, 8, 9$. □

5.2.3 S-Attributed Definitions

As mentioned earlier, given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles. In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order,

https://hemanthrajhemu.github.io

since they do not permit dependency graphs with cycles. Moreover, the two classes introduced in this section can be implemented efficiently in connection with top-down or bottom-up parsing.

The first class is defined as follows:

- An SDD is *S-attributed* if every attribute is synthesized.

Example 5.7: The SDD of Fig. 5.1 is an example of an S-attributed definition. Each attribute, $L.val$, $E.val$, $T.val$, and $F.val$ is synthesized. □

When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time. That is, we apply the function *postorder*, defined below, to the root of the parse tree (see also the box "Preorder and Postorder Traversals" in Section 2.3.4):

```
postorder(N) {
        for ( each child C of N, from the left ) postorder(C);
        evaluate the attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head. This fact will be used in Section 5.4.2 to evaluate synthesized attributes and store them on the stack during LR parsing, without creating the tree nodes explicitly.

5.2.4 L-Attributed Definitions

The second class of SDD's is called *L-attributed definitions*. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1 X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:
   (a) Inherited attributes associated with the head A.
   (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.

https://hemanthrajhemu.github.io

(c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

Example 5.8: The SDD in Fig. 5.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \rightarrow F\, T'$	$T'.inh = F.val$
$T' \rightarrow *\, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$

The first of these rules defines the inherited attribute $T'.inh$ using only $F.val$, and F appears to the left of T' in the production body, as required. The second rule defines $T'_1.inh$ using the inherited attribute $T'.inh$ associated with the head, and $F.val$, where F appears to the left of T'_1 in the production body.

In each of these cases, the rules use information "from above or from the left," as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed. □

Example 5.9: Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

The first rule, $A.s = B.b$, is a legitimate rule in either an S-attributed or L-attributed SDD. It defines a synthesized attribute $A.s$ in terms of an attribute at a child (that is, a symbol within the production body).

The second rule defines an inherited attribute $B.i$, so the entire SDD cannot be S-attributed. Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute $C.c$ is used to help define $B.i$, and C is to the right of B in the production body. While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined. □

5.2.5 Semantic Rules with Controlled Side Effects

In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table. With SDD's, we strike a balance between attribute grammars and translation schemes. Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment; translation schemes are discussed in Section 5.4.

We shall control side effects in SDD's in one of the following ways:

https://hemanthrajhemu.github.io

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a "correct" translation, where "correct" depends on the application.

- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

As an example of an incidental side effect, let us modify the desk calculator of Example 5.1 to print a result. Instead of the rule $L.val = E.val$, which saves the result in the synthesized attribute $L.val$, consider:

	PRODUCTION	SEMANTIC RULE
1)	$L \to E$ **n**	$print(E.val)$

Semantic rules that are executed for their side effects, such as $print(E.val)$, will be treated as the definitions of dummy synthesized attributes associated with the head of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end, after the result is computed into $E.val$.

Example 5.10: The SDD in Fig. 5.8 takes a simple declaration D consisting of a basic type T followed by a list L of identifiers. T can be **int** or **float**. For each identifier on the list, the type is entered into the symbol-table entry for the identifier. We assume that entering the type for one identifier does not affect the symbol-table entry for any other identifier. Thus, entries can be updated in any order. This SDD does not check whether an identifier is declared more than once; it can be modified to do so.

	PRODUCTION	SEMANTIC RULES
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to$ **int**	$T.type =$ integer
3)	$T \to$ **float**	$T.type =$ float
4)	$L \to L_1$, **id**	$L_1.inh = L.inh$
		$addType($**id**$.entry, L.inh)$
5)	$L \to$ **id**	$addType($**id**$.entry, L.inh)$

Figure 5.8: Syntax-directed definition for simple type declarations

Nonterminal D represents a declaration, which, from production 1, consists of a type T followed by a list L of identifiers. T has one attribute, $T.type$, which is the type in the declaration D. Nonterminal L also has one attribute, which we call inh to emphasize that it is an inherited attribute. The purpose of $L.inh$

https://hemanthrajhemu.github.io

is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol-table entries.

Productions 2 and 3 each evaluate the synthesized attribute $T.type$, giving it the appropriate value, integer or float. This type is passed to the attribute $L.inh$ in the rule for production 1. Production 4 passes $L.inh$ down the parse tree. That is, the value $L_1.inh$ is computed at a parse-tree node by copying the value of $L.inh$ from the parent of that node; the parent corresponds to the head of the production.

Productions 4 and 5 also have a rule in which a function *addType* is called with two arguments:

1. **id**.*entry*, a lexical value that points to a symbol-table object, and

2. $L.inh$, the type being assigned to every identifier on the list.

We suppose that function *addType* properly installs the type $L.inh$ as the type of the represented identifier.

A dependency graph for the input string **float** $\textbf{id}_1$, $\textbf{id}_2$, $\textbf{id}_3$ appears in Fig. 5.9. Numbers 1 through 10 represent the nodes of the dependency graph. Nodes 1, 2, and 3 represent the attribute *entry* associated with each of the leaves labeled **id**. Nodes 6, 8, and 10 are the dummy attributes that represent the application of the function *addType* to a type and one of these *entry* values.

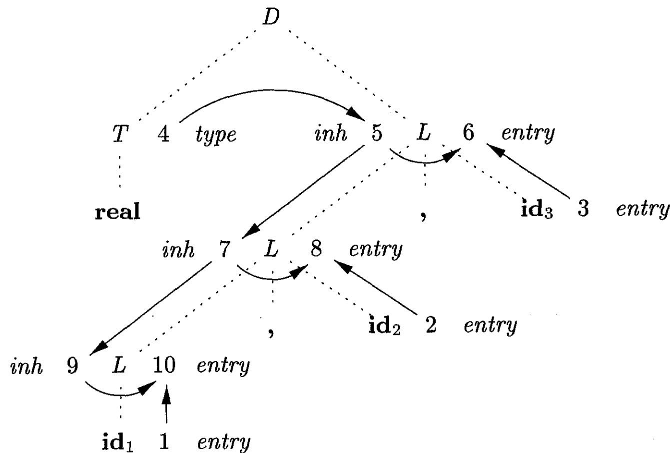


Figure 5.9: Dependency graph for a declaration **float** $\textbf{id}_1$, $\textbf{id}_2$, $\textbf{id}_3$

Node 4 represents the attribute $T.type$, and is actually where attribute evaluation begins. This type is then passed to nodes 5, 7, and 9 representing $L.inh$ associated with each of the occurrences of the nonterminal L. □

https://hemanthrajhemu.github.io

5.2.6 Exercises for Section 5.2

Exercise 5.2.1: What are all the topological sorts for the dependency graph of Fig. 5.7?

Exercise 5.2.2: For the SDD of Fig. 5.8, give annotated parse trees for the following expressions:

a) `int a, b, c.`

b) `float w, x, y, z.`

Exercise 5.2.3: Suppose that we have a production $A \to BCD$. Each of the four nonterminals A, B, C, and D have two attributes: s is a synthesized attribute, and i is an inherited attribute. For each of the sets of rules below, tell whether (i) the rules are consistent with an S-attributed definition (ii) the rules are consistent with an L-attributed definition, and (iii) whether the rules are consistent with any evaluation order at all?

a) $A.s = B.i + C.s$.

b) $A.s = B.i + C.s$ and $D.i = A.i + B.s$.

c) $A.s = B.s + D.s$.

! d) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$, and $D.i = B.i + C.i$.

! **Exercise 5.2.4:** This grammar generates binary numbers with a "decimal" point:

$$S \to L \;.\; L \mid L$$
$$L \to L \; B \mid B$$
$$B \to 0 \mid 1$$

Design an L-attributed SDD to compute $S.val$, the decimal-number value of an input string. For example, the translation of string `101.101` should be the decimal number 5.625. *Hint*: use an inherited attribute $L.side$ that tells which side of the decimal point a bit is on.

!! **Exercise 5.2.5:** Design an S-attributed SDD for the grammar and translation described in Exercise 5.2.4.

!! **Exercise 5.2.6:** Implement Algorithm 3.23, which converts a regular expression into a nondeterministic finite automaton, by an L-attributed SDD on a top-down parsable grammar. Assume that there is a token **char** representing any character, and that **char**.*lexval* is the character it represents. You may also assume the existence of a function *new*() that returns a new state, that is, a state never before returned by this function. Use any convenient notation to specify the transitions of the NFA.

https://hemanthrajhemu.github.io

5.3 Applications of Syntax-Directed Translation

The syntax-directed translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. Here, we consider selected examples to illustrate some representative SDD's.

The main application in this section is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree. (Chapter 6 also discusses approaches to intermediate-code generation that apply an SDD without ever constructing a tree explicitly.)

We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

The final example of this section is an L-attributed definition that deals with basic and array types.

5.3.1 Construction of Syntax Trees

As discussed in Section 2.8.2, each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression $E_1 + E_2$ has label + and two children representing the subexpressions E_1 and E_2.

We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function *Leaf*(*op*, *val*) creates a leaf object. Alternatively, if nodes are viewed as records, then *Leaf* returns a pointer to a new record for a leaf.

- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function *Node* takes two or more arguments: $Node(op, c_1, c_2, \ldots, c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, \ldots, c_k$.

Example 5.11: The S-attributed definition in Fig. 5.10 constructs syntax trees for a simple expression grammar involving only the binary operators + and −. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node*, which represents a node of the syntax tree.

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with $'+'$ for *op* and two children, $E_1.node$ and $T.node$, for the subexpressions. The second production has a similar rule.

https://hemanthrajhemu.github.io

	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new } Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new } Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new } Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new } Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

For production 3, $E \to T$, no node is created, since $E.node$ is the same as $T.node$. Similarly, no node is created for production 4, $T \to (E)$. The value of $T.node$ is the same as $E.node$, since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree.

The last two T-productions have a single terminal on the right. We use the constructor *Leaf* to create a suitable node, which becomes the value of $T.node$.

Figure 5.11 shows the construction of a syntax tree for the input $a - 4 + c$. The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of $E.node$ and $T.node$; each line points to the appropriate syntax-tree node.

At the bottom we see leaves for a, 4 and c, constructed by *Leaf*. We suppose that the lexical value **id**.*entry* points into the symbol table, and the lexical value **num**.*val* is the numerical value of a constant. These leaves, or pointers to them, become the value of $T.node$ at the three parse-tree nodes labeled T, according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for a is also the value of $E.node$ for the leftmost E in the parse tree.

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for $-$ with the third leaf.

If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown in Fig. 5.12 ends with p_5 pointing to the root of the constructed syntax tree. □

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

Example 5.12: The L-attributed definition in Fig. 5.13 performs the same translation as the S-attributed definition in Fig. 5.10. The attributes for the grammar symbols E, T, **id**, and **num** are as discussed in Example 5.11.

https://hemanthrajhemu.github.io

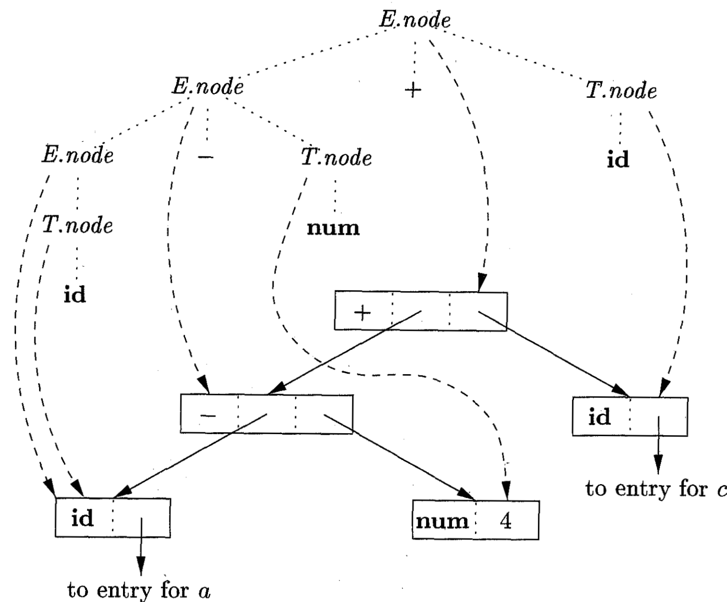


Figure 5.11: Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new } Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new } Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new } Node('-', p_1, p_2);$
4) $p_4 = \textbf{new } Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new } Node('+', p_3, p_4);$

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

The rules for building syntax trees in this example are similar to the rules for the desk calculator in Example 5.3. In the desk-calculator example, a term $x * y$ was evaluated by passing x as an inherited attribute, since x and $* y$ appeared in different portions of the parse tree. Here, the idea is to build a syntax tree for $x + y$ by passing x as an inherited attribute, since x and $+ y$ appear in different subtrees. Nonterminal E' is the counterpart of nonterminal T' in Example 5.3. Compare the dependency graph for $a - 4 + c$ in Fig. 5.14 with that for $3 * 5$ in Fig. 5.7.

Nonterminal E' has an inherited attribute *inh* and a synthesized attribute *syn*. Attribute $E'.inh$ represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for E'. At node 5 in the dependency graph in Fig. 5.14, $E'.inh$ denotes the root of the partial syntax tree for the identifier a; that is, the leaf for a. At node 6, $E'.inh$ denotes the root for the partial syntax

https://hemanthrajhemu.github.io

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh =$ **new** $Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh =$ **new** $Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**$,$ **id**$.entry)$
7)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**$,$ **num**$.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

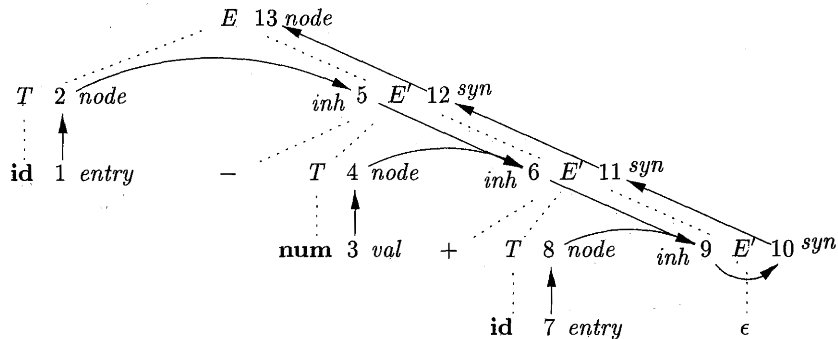


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

tree for the input $a - 4$. At node 9, $E'.inh$ denotes the syntax tree for $a - 4 + c$.

Since there is no more input, at node 9, $E'.inh$ points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of $E.node$. Specifically, the attribute value at node 10 is defined by the rule $E'.syn = E'.inh$ associated with the production $E' \rightarrow \epsilon$. The attribute value at node 11 is defined by the rule $E'.syn = E'_1.syn$ associated with production 2 in Fig. 5.13. Similar rules define the attribute values at nodes 12 and 13. □

5.3.2 The Structure of a Type

Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input; attributes can then be used to carry informa-

https://hemanthrajhemu.github.io

tion from one part of the parse tree to another. The next example shows how a mismatch in structure can be due to the design of the language, and not due to constraints imposed by the parsing method.

Example 5.13: In C, the type **int** [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression *array*(2, *array*(3, *integer*)) is represented by the tree in Fig. 5.15. The operator *array* takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled *array* with two children for a number and a type.

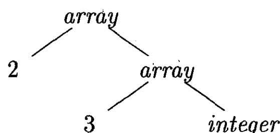


Figure 5.15: Type expression for **int**[2][3]

With the SDD in Fig. 5.16, nonterminal T generates either a basic type or an array type. Nonterminal B generates one of the basic types **int** and **float**. T generates a basic type when T derives $B\,C$ and C derives ϵ. Otherwise, C generates array components consisting of a sequence of integers, each integer surrounded by brackets.

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

The nonterminals B and T have a synthesized attribute t representing a type. The nonterminal C has two attributes: an inherited attribute b and a synthesized attribute t. The inherited b attributes pass a basic type down the tree, and the synthesized t attributes accumulate the result.

An annotated parse tree for the input string **int** [2] [3] is shown in Fig. 5.17. The corresponding type expression in Fig. 5.15 is constructed by passing the type *integer* from B, down the chain of C's through the inherited attributes b. The array type is synthesized up the chain of C's through the attributes t.

In more detail, at the root for $T \rightarrow B\ C$, nonterminal C inherits the type from B, using the inherited attribute $C.b$. At the rightmost node for C, the

https://hemanthrajhemu.github.io

production is $C \rightarrow \epsilon$, so $C.t$ equals $C.b$. The semantic rules for the production $C \rightarrow$ [**num**] C_1 form $C.t$ by applying the operator *array* to the operands **num**.*val* and $C_1.t$. □

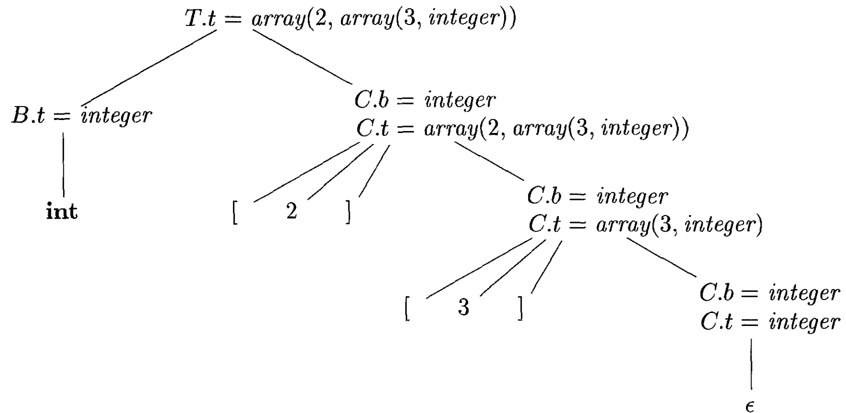


Figure 5.17: Syntax-directed translation of array types

5.3.3 Exercises for Section 5.3

Exercise 5.3.1: Below is a grammar for expressions involving operator + and integer or floating-point operands. Floating-point numbers are distinguished by having a decimal point.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \textbf{num . num} \mid \textbf{num}$$

a) Give an SDD to determine the type of each term T and expression E.

b) Extend your SDD of (a) to translate expressions into postfix notation. Use the unary operator **intToFloat** to turn an integer into an equivalent float.

! **Exercise 5.3.2:** Give an SDD to translate infix expressions with + and $*$ into equivalent expressions without redundant parentheses. For example, since both operators associate from the left, and $*$ takes precedence over +, $((a*(b+c))*(d))$ translates into $a * (b + c) * d$.

! **Exercise 5.3.3:** Give an SDD to differentiate expressions such as $x * (3 * x + x * x)$ involving the operators + and $*$, the variable x, and constants. Assume that no simplification occurs, so that, for example, $3 * x$ will be translated into $3 * 1 + 0 * x$.

https://hemanthrajhemu.github.io

5.4 Syntax-Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. All of the applications of syntax-directed definitions in Section 5.3 can be implemented using syntax-directed translation schemes.

From Section 2.3.5, a *syntax-directed translation scheme* (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called *semantic actions* and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal. An example appears in Section 5.4.3.

Typically, SDT's are implemented during parsing, without building a parse tree. In this section, we focus on the use of SDT's to implement two important classes of SDD's:

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.

2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

We shall see how, in both these cases, the semantic rules in an SDD can be converted into an SDT with actions that are executed at the right time. During parsing, an action in a production body is executed as soon as all the grammar symbols to the left of the action have been matched.

SDT's that can be implemented during parsing can be characterized by introducing distinct *marker nonterminals* in place of each embedded action; each marker M has only one production, $M \rightarrow \epsilon$. If the grammar with marker nonterminals can be parsed by a given method, then the SDT can be implemented during parsing.

5.4.1 Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called *postfix SDT's*.

Example 5.14: The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser. □

https://hemanthrajhemu.github.io

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Figure 5.18: Postfix SDT implementing the desk calculator

5.4.2 Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur. The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.

In Fig. 5.19, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols $X\,Y\,Z$ are on top of the stack; perhaps they are about to be reduced according to a production like $A \rightarrow X\,Y\,Z$. Here, we show $X.x$ as the one attribute of X, and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

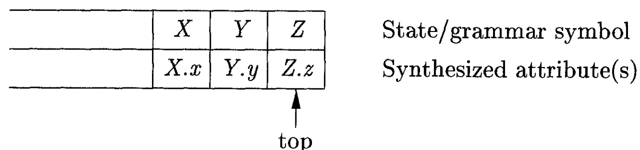


Figure 5.19: Parser stack with a field for synthesized attributes

If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head. If we reduce by a production such as $A \rightarrow X\,Y\,Z$, then we have all the attributes of X, Y, and Z available, at known positions on the stack, as in Fig. 5.19. After the action, A and its attributes are at the top of the stack, in the position of the record for X.

Example 5.15: Let us rewrite the actions of the desk-calculator SDT of Ex-

https://hemanthrajhemu.github.io

ample 5.14 so that they manipulate the parser stack explicitly. Such stack manipulation is usually done automatically by the parser.

PRODUCTION	ACTIONS
$L \to E$ **n**	{ print($stack[top-1].val$); $top = top - 1$; }
$E \to E_1 + T$	{ $stack[top-2].val = stack[top-2].val + stack[top].val$; $top = top - 2$; }
$E \to T$	
$T \to T_1 * F$	{ $stack[top-2].val = stack[top-2].val \times stack[top].val$; $top = top - 2$; }
$T \to F$	
$F \to (E)$	{ $stack[top-2].val = stack[top-1].val$; $top = top - 2$; }
$F \to$ **digit**	

Figure 5.20: Implementing the desk calculator on a bottom-up parsing stack

Suppose that the stack is kept in an array of records called *stack*, with *top* a cursor to the top of the stack. Thus, $stack[top]$ refers to the top record on the stack, $stack[top-1]$ to the record below that, and so on. Also, we assume that each record has a field called *val*, which holds the attribute of whatever grammar symbol is represented in that record. Thus, we may refer to the attribute $E.val$ that appears at the third position on the stack as $stack[top-2].val$. The entire SDT is shown in Fig. 5.20.

For instance, in the second production, $E \to E_1 + T$, we go two positions below the top to get the value of E_1, and we find the value of T at the top. The resulting sum is placed where the head E will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one. After computing $E.val$, we pop two symbols off the top of the stack, so the record where we placed $E.val$ will now be at the top of the stack.

In the third production, $E \to T$, no action is necessary, because the length of the stack does not change, and the value of $T.val$ at the stack top will simply become the value of $E.val$. The same observation applies to the productions $T \to F$ and $F \to$ **digit**. Production $F \to (E)$ is slightly different. Although the value does not change, two positions are removed from the stack during the reduction, so the value has to move to the position after the reduction.

Note that we have omitted the steps that manipulate the first field of the stack records — the field that gives the LR state or otherwise represents the grammar symbol. If we are performing an LR parse, the parsing table tells us what the new state is every time we reduce; see Algorithm 4.44. Thus, we may

https://hemanthrajhemu.github.io

simply place that state in the record for the new top of stack. □

5.4.3 SDT's With Actions Inside Productions

An action may be placed at any position within the body of a production. It is performed immediately after all symbols to its left are processed. Thus, if we have a production $B \rightarrow X \{a\} Y$, the action a is done after we have recognized X (if X is a terminal) or all the terminals derived from X (if X is a nonterminal). More precisely,

- If the parse is bottom-up, then we perform action a as soon as this occurrence of X appears on the top of the parsing stack.
- If the parse is top-down, we perform a just before we attempt to expand this occurrence of Y (if Y a nonterminal) or check for Y on the input (if Y is a terminal).

SDT's that can be implemented during parsing include postfix SDT's and a class of SDT's considered in Section 5.5 that implements L-attributed definitions. Not all SDT's can be implemented during parsing, as we shall see in the next example.

Example 5.16: As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown in Fig. 5.21.

$$
\begin{array}{llll}
1) & L & \rightarrow & E \ \mathbf{n} \\
2) & E & \rightarrow & \{ \ \text{print}('+'); \ \} \ \ E_1 + T \\
3) & E & \rightarrow & T \\
4) & T & \rightarrow & \{ \ \text{print}('*'); \ \} \ \ T_1 * F \\
5) & T & \rightarrow & F \\
6) & F & \rightarrow & (\ E\ ) \\
7) & F & \rightarrow & \mathbf{digit} \ \ \{ \ \text{print}(\mathbf{digit}.\mathit{lexval}); \ \}
\end{array}
$$

Figure 5.21: Problematic SDT for infix-to-prefix translation during parsing

Unfortunately, it is impossible to implement this SDT during either top-down or bottom-up parsing, because the parser would have to perform critical actions, like printing instances of $*$ or $+$, long before it knows whether these symbols will appear in its input.

Using marker nonterminals M_2 and M_4 for the actions in productions 2 and 4, respectively, on input 3, a shift-reduce parser (see Section 4.5.3) has conflicts between reducing by $M_2 \rightarrow \epsilon$, reducing by $M_4 \rightarrow \epsilon$, and shifting the digit. □

https://hemanthrajhemu.github.io

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.

2. Then, examine each interior node N, say one for production $A \rightarrow \alpha$. Add additional children to N for the actions in α, so the children of N from left to right have exactly the symbols and actions of α.

3. Perform a preorder traversal (see Section 2.3.4) of the tree, and as soon as a node labeled by an action is visited, perform that action.

For instance, Fig. 5.22 shows the parse tree for expression $3 * 5 + 4$ with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: $+ * 3\,5\,4$.

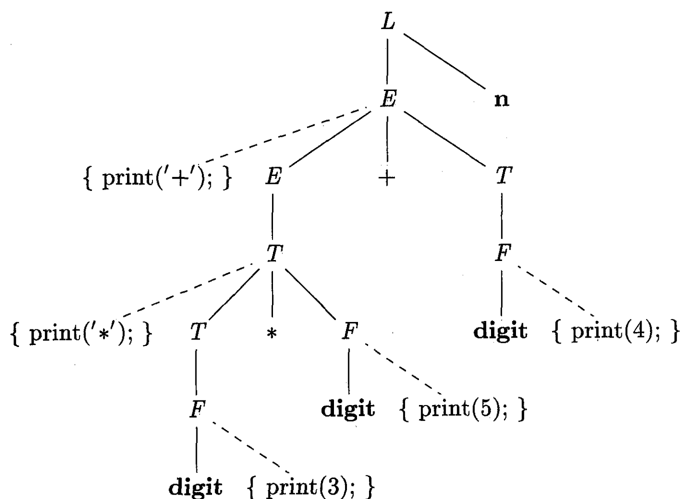


Figure 5.22: Parse tree with actions embedded

5.4.4 Eliminating Left Recursion From SDT's

Since no grammar with left recursion can be parsed deterministically top-down, we examined left-recursion elimination in Section 4.3.3. When the grammar is part of an SDT, we also need to worry about how the actions are handled.

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

- When transforming the grammar, treat the actions as if they were terminal symbols.

https://hemanthrajhemu.github.io

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up.

The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal R (for "remainder") of the first production:

$$\begin{array}{l} A \rightarrow \beta R \\ R \rightarrow \alpha R \mid \epsilon \end{array}$$

If β does not begin with A, then A no longer has a left-recursive production. In regular-definition terms, with both sets of productions, A is defined by $\beta(\alpha)^*$. See Section 4.3.3 for the handling of situations where A has more recursive or nonrecursive productions.

Example 5.17: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

$$\begin{array}{lcll} E & \rightarrow & E_1 + T & \{\ \text{print}('+');\ \} \\ E & \rightarrow & T & \end{array}$$

If we apply the standard transformation to E, the remainder of the left-recursive production is

$$\alpha \;=\; +\; T\ \{\ \text{print}('+');\ \}$$

and β, the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

$$\begin{array}{lcl} E & \rightarrow & T\ R \\ R & \rightarrow & +\ T\ \{\ \text{print}('+');\ \}\ R \\ R & \rightarrow & \epsilon \end{array}$$

□

When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions.

We shall give a general schema for the case of a single recursive production, a single nonrecursive production, and a single attribute of the left-recursive nonterminal; the generalization to many productions of each type is not hard, but is notationally cumbersome. Suppose that the two productions are

https://hemanthrajhemu.github.io

$$\begin{array}{lcl} A & \rightarrow & A_1 \; Y \; \{A.a = g(A_1.a, Y.y)\} \\ A & \rightarrow & X \; \{A.a = f(X.x)\} \end{array}$$

Here, $A.a$ is the synthesized attribute of left-recursive nonterminal A, and X and Y are single grammar symbols with synthesized attributes $X.x$ and $Y.y$, respectively. These could represent a string of several grammar symbols, each with its own attribute(s), since the schema has an arbitrary function g computing $A.a$ in the recursive production and an arbitrary function f computing $A.a$ in the second production. In each case, f and g take as arguments whatever attributes they are allowed to access if the SDD is S-attributed.

We want to turn the underlying grammar into

$$\begin{array}{lcl} A & \rightarrow & X \; R \\ R & \rightarrow & Y \; R \mid \epsilon \end{array}$$

Figure 5.23 suggests what the SDT on the new grammar must do. In (a) we see the effect of the postfix SDT on the original grammar. We apply f once, corresponding to the use of production $A \rightarrow X$, and then apply g as many times as we use the production $A \rightarrow AY$. Since R generates a "remainder" of Y's, its translation depends on the string to its left, a string of the form $XYY \cdots Y$. Each use of the production $R \rightarrow YR$ results in an application of g. For R, we use an inherited attribute $R.i$ to accumulate the result of successively applying g, starting with the value of $A.a$.

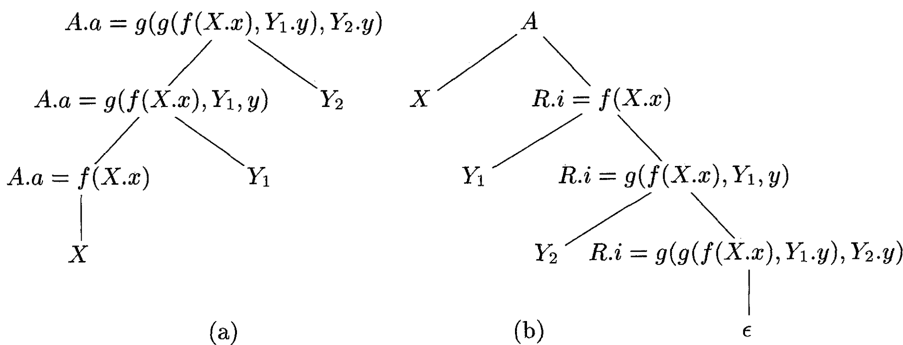


Figure 5.23: Eliminating left recursion from a postfix SDT

In addition, R has a synthesized attribute $R.s$, not shown in Fig. 5.23. This attribute is first computed when R ends its generation of Y symbols, as signaled by the use of production $R \rightarrow \epsilon$. $R.s$ is then copied up the tree, so it can become the value of $A.a$ for the entire expression $XYY \cdots Y$. The case where A generates XYY is shown in Fig. 5.23, and we see that the value of $A.a$ at the root of (a) has two uses of g. So does $R.i$ at the bottom of tree (b), and it is this value of $R.s$ that gets copied up that tree.

To accomplish this translation, we use the following SDT:

https://hemanthrajhemu.github.io

$$
\begin{array}{lcl}
A & \rightarrow & X \ \{R.i = f(X.x)\} \ R \ \{A.a = R.s\} \\
R & \rightarrow & Y \ \{R_1.i = g(R.i, Y.y\} \ R_1 \ \{R.s = R_1.s\} \\
R & \rightarrow & \epsilon \ \{R.s = R.i\}
\end{array}
$$

Notice that the inherited attribute $R.i$ is evaluated immediately before a use of R in the body, while the synthesized attributes $A.a$ and $R.s$ are evaluated at the ends of the productions. Thus, whatever values are needed to compute these attributes will be available from what has been computed to the left.

5.4.5 SDT's for L-Attributed Definitions

In Section 5.4.1, we converted S-attributed SDD's into postfix SDT's, with actions at the right ends of productions. As long as the underlying grammar is LR, postfix SDT's can be parsed and translated bottom-up.

Now, we consider the more general case of an L-attributed SDD. We shall assume that the underlying grammar can be parsed top-down, for if not it is frequently impossible to perform the translation in connection with either an LL or an LR parser. With any grammar, the technique below can be implemented by attaching actions to a parse tree and executing them during preorder traversal of the tree.

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal A immediately before that occurrence of A in the body of the production. If several inherited attributes for A depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.

2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

We shall illustrate these principles with two extended examples. The first involves typesetting. It illustrates how the techniques of compiling can be used in language processing for applications other than what we normally think of as programming languages. The second example is about the generation of intermediate code for a typical programming-language construct: a form of while-statement.

Example 5.18: This example is motivated by languages for typesetting mathematical formulas. `Eqn` is an early example of such a language; ideas from `Eqn` are still found in the TeX typesetting system, which was used to produce this book.

We shall concentrate on only the capability to define subscripts, subscripts of subscripts, and so on, ignoring superscripts, built-up fractions, and all other mathematical features. In the `Eqn` language, one writes `a sub i sub j` to set the expression a_{i_j}. A simple grammar for *boxes* (elements of text bounded by a rectangle) is

https://hemanthrajhemu.github.io

$$B \rightarrow B_1 \; B_2 \mid B_1 \textbf{ sub } B_2 \mid (\; B_1 \;) \mid \textbf{text}$$

Corresponding to these four productions, a box can be either

1. Two boxes, juxtaposed, with the first, B_1, to the left of the other, B_2.

2. A box and a subscript box. The second box appears in a smaller size, lower, and to the right of the first box.

3. A parenthesized box, for grouping of boxes and subscripts. Eqn and TEX both use curly braces for grouping, but we shall use ordinary, round parentheses to avoid confusion with the braces that surround actions in SDT's.

4. A text string, that is, any string of characters.

This grammar is ambiguous, but we can still use it to parse bottom-up if we make subscripting and juxtaposition right associative, with **sub** taking precedence over juxtaposition.

Expressions will be typeset by constructing larger boxes out of smaller ones. In Fig. 5.24, the boxes for E_1 and *.height* are about to be juxtaposed to form the box for E_1*.height*. The left box for E_1 is itself constructed from the box for E and the subscript 1. The subscript 1 is handled by shrinking its box by about 30%, lowering it, and placing it after the box for E. Although we shall treat *.height* as a text string, the rectangles within its box show how it can be constructed from boxes for the individual letters.

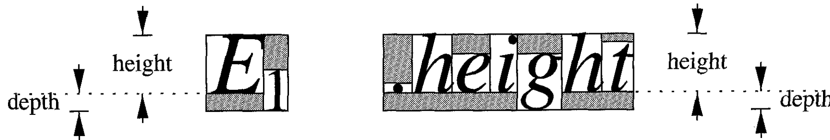


Figure 5.24: Constructing larger boxes from smaller ones

In this example, we concentrate on the vertical geometry of boxes only. The horizontal geometry — the widths of boxes — is also interesting, especially when different characters have different widths. It may not be readily apparent, but each of the distinct characters in Fig. 5.24 has a different width.

The values associated with the vertical geometry of boxes are as follows:

a) The *point size* is used to set text within a box. We shall assume that characters not in subscripts are set in 10 point type, the size of type in this book. Further, we assume that if a box has point size p, then its subscript box has the smaller point size $0.7p$. Inherited attribute $B.ps$ will represent the point size of block B. This attribute must be inherited, because the context determines by how much a given box needs to be shrunk, due to the number of levels of subscripting.

https://hemanthrajhemu.github.io

b) Each box has a *baseline*, which is a vertical position that corresponds to the bottoms of lines of text, not counting any letters, like "g" that extend below the normal baseline. In Fig. 5.24, the dotted line represents the baseline for the boxes E, *.height*, and the entire expression. The baseline for the box containing the subscript 1 is adjusted to lower the subscript.

c) A box has a *height*, which is the distance from the top of the box to the baseline. Synthesized attribute $B.ht$ gives the height of box B.

d) A box has a *depth*, which is the distance from the baseline to the bottom of the box. Synthesized attribute $B.dp$ gives the depth of box B.

The SDD in Fig. 5.25 gives rules for computing point sizes, heights, and depths. Production 1 is used to assign $B.ps$ the initial value 10.

	Production	Semantic Rules
1)	$S \rightarrow B$	$B.ps = 10$
2)	$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$ $B.dp = \max(B_1.dp, B_2.dp)$
3)	$B \rightarrow B_1$ **sub** B_2	$B_1.ps = B.ps$ $B_2.ps = 0.7 \times B.ps$ $B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4)	$B \rightarrow (\ B_1\ )$	$B_1.ps = B.ps$ $B.ht = B_1.ht$ $B.dp = B_1.dp$
5)	$B \rightarrow$ **text**	$B.ht = getHt(B.ps, \textbf{text}.lexval)$ $B.dp = getDp(B.ps, \textbf{text}.lexval)$

Figure 5.25: SDD for typesetting boxes

Production 2 handles juxtaposition. Point sizes are copied down the parse tree; that is, two sub-boxes of a box inherit the same point size from the larger box. Heights and depths are computed up the tree by taking the maximum. That is, the height of the larger box is the maximum of the heights of its two components, and similarly for the depth.

Production 3 handles subscripting and is the most subtle. In this greatly simplified example, we assume that the point size of a subscripted box is 70% of the point size of its parent. Reality is much more complex, since subscripts cannot shrink indefinitely; in practice, after a few levels, the sizes of subscripts

https://hemanthrajhemu.github.io

shrink hardly at all. Further, we assume that the baseline of a subscript box drops by 25% of the parent's point size; again, reality is more complex.

Production 4 copies attributes appropriately when parentheses are used. Finally, production 5 handles the leaves that represent text boxes. In this matter too, the true situation is complicated, so we merely show two unspecified functions *getHt* and *getDp* that examine tables created with each font to determine the maximum height and maximum depth of any characters in the text string. The string itself is presumed to be provided as the attribute *lexval* of terminal **text**.

Our last task is to turn this SDD into an SDT, following the rules for an L-attributed SDD, which Fig. 5.25 is. The appropriate SDT is shown in Fig. 5.26. For readability, since production bodies become long, we split them across lines and line up the actions. Production bodies therefore consist of the contents of all lines up to the head of the next production. □

	PRODUCTION		ACTIONS
1)	$S \rightarrow$		$\{\ B.ps = 10;\ \}$
		B	
2)	$B \rightarrow$		$\{ B_1.ps = B.ps;\ \}$
		B_1	$\{ B_2.ps = B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$ $B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow$		$\{ B_1.ps = B.ps;\ \}$
		B_1 **sub**	$\{ B_2.ps = 0.7 \times B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow$	(	$\{ B_1.ps = B.ps;\ \}$
		B_1)	$\{\ B.ht = B_1.ht;$ $B.dp = B_1.dp;\ \}$
5)	$B \rightarrow$	**text**	$\{\ B.ht = getHt(B.ps, \textbf{text}.lexval);$ $B.dp = getDp(B.ps, \textbf{text}.lexval);\ \}$

Figure 5.26: SDT for typesetting boxes

Our next example concentrates on a simple while-statement and the generation of intermediate code for this type of statement. Intermediate code will be treated as a string-valued attribute. Later, we shall explore techniques that involve the writing of pieces of a string-valued attribute as we parse, thus avoiding the copying of long strings to build even longer strings. The technique was introduced in Example 5.17, where we generated the postfix form of an infix

https://hemanthrajhemu.github.io

expression "on-the-fly," rather than computing it as an attribute. However, in our first formulation, we create a string-valued attribute by concatenation.

Example 5.19: For this example, we only need one production:

$$S \rightarrow \textbf{while} \ (\ C \ ) \ S_1$$

Here, S is the nonterminal that generates all kinds of statements, presumably including if-statements, assignment statements, and others. In this example, C stands for a conditional expression — a boolean expression that evaluates to true or false.

In this flow-of-control example, the only things we ever generate are labels. All the other intermediate-code instructions are assumed to be generated by parts of the SDT that are not shown. Specifically, we generate explicit instructions of the form **label** L, where L is an identifier, to indicate that L is the label of the instruction that follows. We assume that the intermediate code is like that introduced in Section 2.8.4.

The meaning of our while-statement is that the conditional C is evaluated. If it is true, control goes to the beginning of the code for S_1. If false, then control goes to the code that follows the while-statement's code. The code for S_1 must be designed to jump to the beginning of the code for the while-statement when finished; the jump to the beginning of the code that evaluates C is not shown in Fig. 5.27.

We use the following attributes to generate the proper intermediate code:

1. The inherited attribute $S.next$ labels the beginning of the code that must be executed after S is finished.

2. The synthesized attribute $S.code$ is the sequence of intermediate-code steps that implements a statement S and ends with a jump to $S.next$.

3. The inherited attribute $C.true$ labels the beginning of the code that must be executed if C is true.

4. The inherited attribute $C.false$ labels the beginning of the code that must be executed if C is false.

5. The synthesized attribute $C.code$ is the sequence of intermediate-code steps that implements the condition C and jumps either to $C.true$ or to $C.false$, depending on whether C is true or false.

The SDD that computes these attributes for the while-statement is shown in Fig. 5.27. A number of points merit explanation:

- The function *new* generates new labels.

- The variables $L1$ and $L2$ hold labels that we need in the code. $L1$ is the beginning of the code for the while-statement, and we need to arrange

https://hemanthrajhemu.github.io

$$
\begin{array}{ll}
S \rightarrow \textbf{while}\ (\ C\ )\ S_1 & L1 = new(); \\
 & L2 = new(); \\
 & S_1.next = L1; \\
 & C.false = S.next; \\
 & C.true = L2; \\
 & S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code
\end{array}
$$

Figure 5.27: SDD for while-statements

that S_1 jumps there after it finishes. That is why we set $S_1.next$ to $L1$. $L2$ is the beginning of the code for S_1, and it becomes the value of $C.true$, because we branch there when C is true.

- Notice that $C.false$ is set to $S.next$, because when the condition is false, we execute whatever code must follow the code for S.
- We use $\parallel$ as the symbol for concatenation of intermediate-code fragments. The value of $S.code$ thus begins with the label $L1$, then the code for condition C, another label $L2$, and the code for S_1.

This SDD is L-attributed. When we convert it into an SDT, the only remaining issue is how to handle the labels $L1$ and $L2$, which are variables, and not attributes. If we treat actions as dummy nonterminals, then such variables can be treated as the synthesized attributes of dummy nonterminals. Since $L1$ and $L2$ do not depend on any other attributes, they can be assigned to the first action in the production. The resulting SDT with embedded actions that implements this L-attributed definition is shown in Fig. 5.28. □

$$
\begin{array}{lll}
S & \rightarrow \textbf{while}\ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \} \\
 & C\ ) & \{\ S_1.next = L1;\ \} \\
 & S_1 & \{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}
\end{array}
$$

Figure 5.28: SDT for while-statements

5.4.6 Exercises for Section 5.4

Exercise 5.4.1: We mentioned in Section 5.4.2 that it is possible to deduce, from the LR state on the parsing stack, what grammar symbol is represented by the state. How would we discover this information?

Exercise 5.4.2: Rewrite the following SDT:

$$
\begin{array}{l}
A \rightarrow A\ \{a\}\ B \mid A\ B\ \{b\} \mid 0 \\
B \rightarrow B\ \{c\}\ A \mid B\ A\ \{d\} \mid 1
\end{array}
$$

https://hemanthrajhemu.github.io

so that the underlying grammar becomes non-left-recursive. Here, a, b, c, and d are actions, and 0 and 1 are terminals.

! Exercise 5.4.3: The following SDT computes the value of a string of 0's and 1's interpreted as a positive, binary integer.

$$\begin{array}{rcl} B & \rightarrow & B_1\ 0\ \{B.val = 2 \times B_1.val\} \\ & | & B_1\ 1\ \{B.val = 2 \times B_1.val + 1\} \\ & | & 1\ \{B.val = 1\} \end{array}$$

Rewrite this SDT so the underlying grammar is not left recursive, and yet the same value of $B.val$ is computed for the entire input string.

! Exercise 5.4.4: Write L-attributed SDD's analogous to that of Example 5.19 for the following productions, each of which represents a familiar flow-of-control construct, as in the programming language C. You may need to generate a three-address statement to jump to a particular label L, in which case you should generate **goto** L.

a) $S \rightarrow$ **if** (C) S_1 **else** S_2

b) $S \rightarrow$ **do** S_1 **while** (C)

c) $S \rightarrow$ '{' L '}'; $L \rightarrow L\ S \mid \epsilon$

Note that any statement in the list can have a jump from its middle to the next statement, so it is not sufficient simply to generate code for each statement in order.

Exercise 5.4.5: Convert each of your SDD's from Exercise 5.4.4 to an SDT in the manner of Example 5.19.

Exercise 5.4.6: Modify the SDD of Fig. 5.25 to include a synthesized attribute $B.le$, the length of a box. The length of the concatenation of two boxes is the sum of the lengths of each. Then add your new rules to the proper positions in the SDT of Fig. 5.26

Exercise 5.4.7: Modify the SDD of Fig. 5.25 to include superscripts denoted by operator **sup** between boxes. If box B_2 is a superscript of box B_1, then position the baseline of B_2 0.6 times the point size of B_1 above the baseline of B_1. Add the new production and rules to the SDT of Fig. 5.26.

5.5 Implementing L-Attributed SDD's

Since many translation applications can be addressed using L-attributed definitions, we shall consider their implementation in more detail in this section. The following methods do translation by traversing a parse tree:

https://hemanthrajhemu.github.io

1. *Build the parse tree and annotate.* This method works for any noncircular SDD whatsoever. We introduced annotated parse trees in Section 5.1.2.

2. *Build the parse tree, add actions, and execute the actions in preorder.* This approach works for any L-attributed definition. We discussed how to turn an L-attributed SDD into an SDT in Section 5.4.5; in particular, we discussed how to embed actions into productions based on the semantic rules of such an SDD.

In this section, we discuss the following methods for translation during parsing:

3. *Use a recursive-descent parser* with one function for each nonterminal. The function for nonterminal A receives the inherited attributes of A as arguments and returns the synthesized attributes of A.

4. *Generate code on the fly*, using a recursive-descent parser.

5. *Implement an SDT in conjunction with an LL-parser.* The attributes are kept on the parsing stack, and the rules fetch the needed attributes from known locations on the stack.

6. *Implement an SDT in conjunction with an LR-parser.* This method may be surprising, since the SDT for an L-attributed SDD typically has actions in the middle of productions, and we cannot be sure during an LR parse that we are even in that production until its entire body has been constructed. We shall see, however, that if the underlying grammar is LL, we can always handle both the parsing and translation bottom-up.

5.5.1 Translation During Recursive-Descent Parsing

A recursive-descent parser has a function A for each nonterminal A, as discussed in Section 4.4.1. We can extend the parser into a translator as follows:

a) The arguments of function A are the inherited attributes of nonterminal A.

b) The return-value of function A is the collection of synthesized attributes of nonterminal A.

In the body of function A, we need to both parse and handle attributes:

1. Decide upon the production used to expand A.

2. Check that each terminal appears on the input when it is required. We shall assume that no backtracking is needed, but the extension to recursive-descent parsing with backtracking can be done by restoring the input position upon failure, as discussed in Section 4.4.1.

https://hemanthrajhemu.github.io

3. Preserve, in local variables, the values of all attributes needed to compute inherited attributes for nonterminals in the body or synthesized attributes for the head nonterminal.

4. Call functions corresponding to nonterminals in the body of the selected production, providing them with the proper arguments. Since the underlying SDD is L-attributed, we have already computed these attributes and stored them in local variables.

Example 5.20: Let us consider the SDD and SDT of Example 5.19 for while-statements. A pseudocode rendition of the relevant parts of the function S appears in Fig. 5.29.

```
string S(label next) {
        string Scode, Ccode; /* local variables holding code fragments */
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                check ')' is next on the input, and advance;
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* other statement types */
}
```

Figure 5.29: Implementing while-statements with a recursive-descent parser

We show S as storing and returning long strings. In practice, it would be far more efficient for functions like S and C to return pointers to records that represent these strings. Then, the return-statement in function S would not physically concatenate the components shown, but rather would construct a record, or perhaps tree of records, expressing the concatenation of the strings represented by *Scode* and *Ccode*, the labels $L1$ and $L2$, and the two occurrences of the literal string `"label"`. □

Example 5.21: Now, let us take up the SDT of Fig. 5.26 for typesetting boxes. First, we address parsing, since the underlying grammar in Fig. 5.26 is ambiguous. The following transformed grammar makes juxtaposition and subscripting right associative, with **sub** taking precedence over juxtaposition:

https://hemanthrajhemu.github.io

$$
\begin{array}{lcl}
S & \rightarrow & B \\
B & \rightarrow & T\ B_1 \ \mid\ T \\
T & \rightarrow & F\ \textbf{sub}\ T_1 \ \mid\ F \\
F & \rightarrow & (\ B\ ) \ \mid\ \textbf{text}
\end{array}
$$

The two new nonterminals, T and F, are motivated by terms and factors in expressions. Here, a "factor," generated by F, is either a parenthesized box or a text string. A "term," generated by T, is a "factor" with a sequence of subscripts, and a box generated by B is a sequence of juxtaposed "terms."

The attributes of B carry over to T and F, since the new nonterminals also denote boxes; they were introduced simply to aid parsing. Thus, both T and F have an inherited attribute *ps* and synthesized attributes *ht* and *dp*, with semantic actions that can be adapted from the SDT in Fig. 5.26.

The grammar is not yet ready for top-down parsing, since the productions for B and T have common prefixes. Consider T, for instance. A top-down parser cannot choose between the two productions for T by looking one symbol ahead in the input. Fortunately, we can use a form of left-factoring, discussed in Section 4.3.4, to make the grammar ready. With SDT's, the notion of common prefix applies to actions as well. Both productions for T begin with the nonterminal F inheriting attribute *ps* from T.

The pseudocode in Fig. 5.30 for $T(ps)$ folds in the code for $F(ps)$. After left-factoring is applied to $T \rightarrow F\ \textbf{sub}\ T_1 \mid F$, there is only one call to F; the pseudocode shows the result of substituting the code for F in place of this call.

The function T will be called as $T(10.0)$ by the function for B, which we do not show. It returns a pair consisting of the height and depth of the box generated by nonterminal T; in practice, it would return a record containing the height and depth.

Function T begins by checking for a left parenthesis, in which case it must have the production $F \rightarrow (B)$ to work with. It saves whatever the B inside the parentheses returns, but if that B is not followed by a right parenthesis, then there is a syntax error, which must be handled in a manner not shown.

Otherwise, if the current input is **text**, then the function T uses *getHt* and *getDp* to determine the height and depth of this text.

T then decides whether the next box is a subscript and adjusts the point size, if so. We use the actions associated with the production $B \rightarrow B\ \textbf{sub}\ B$ in Fig. 5.26 for the height and depth of the larger box. Otherwise, we simply return what F would have returned: $(h1, d1)$. □

5.5.2 On-The-Fly Code Generation

The construction of long strings of code that are attribute values, as in Example 5.20, is undesirable for several reasons, including the time it could take to copy or move long strings. In common cases such as our running code-generation example, we can instead incrementally generate pieces of the code into an array or output file by executing actions in an SDT. The elements we need to make this technique work are:

https://hemanthrajhemu.github.io

```
(float, float) T(float ps) {
        float h1, h2, d1, d2; /* locals to hold heights and depths */
        /* start code for F(ps) */
        if ( current input == '(' ) {
                advance input;
                (h1, d1) = B(ps);
                if (current input != ')' ) syntax error: expected ')';
                advance input;
        }
        else if ( current input == text ) {
                let lexical value text.lexval be t;
                advance input;
                h1 = getHt(ps, t);
                d1 = getDp(ps, t);
        }
        else syntax error: expected text or '(';
        /* end code for F(ps) */
        if ( current input == sub ) {
                advance input;
                (h2, d2) = T(0.7 * ps);
                return (max(h1, h2 - 0.25 * ps),  max(d1, d2 + 0.25 * ps));
        }
        return (h1, d1);
}
```

Figure 5.30: Recursive-descent typesetting of boxes

1. There is, for one or more nonterminals, a *main* attribute. For convenience, we shall assume that the main attributes are all string valued. In Example 5.20, the attributes *S.code* and *C.code* are main attributes; the other attributes are not.

2. The main attributes are synthesized.

3. The rules that evaluate the main attribute(s) ensure that

   (a) The main attribute is the concatenation of main attributes of nonterminals appearing in the body of the production involved, perhaps with other elements that are not main attributes, such as the string **label** or the values of labels $L1$ and $L2$.

   (b) The main attributes of nonterminals appear in the rule in the same order as the nonterminals themselves appear in the production body.

As a consequence of the above conditions, the main attribute can be constructed by emitting the non-main-attribute elements of the concatenation. We can rely

https://hemanthrajhemu.github.io

The Type of Main Attributes

Our simplifying assumption that main attributes are of string type is really too restrictive. The true requirement is that the type of all the main attributes must have values that can be constructed by concatenation of elements. For instance, a list of objects of any type would be appropriate, as long as we represent these lists in a way that allows elements to be efficiently appended to the end of the list. Thus, if the purpose of the main attribute is to represent a sequence of intermediate-code statements, we could produce the intermediate code by writing statements to the end of an array of objects. Of course the requirements stated in Section 5.5.2 still apply to lists; for example, main attributes must be assembled from other main attributes by concatenation in order.

on the recursive calls to the functions for the nonterminals in a production body to emit the value of their main attribute incrementally.

Example 5.22: We can modify the function of Fig. 5.29 to emit elements of the main translation *S.code* instead of saving them for concatenation into a return value of *S.code*. The revised function *S* appears in Fig. 5.31.

```
void S(label next) {
    label L1, L2; /* the local labels */
    if ( current input == token while ) {
        advance input;
        check '(' is next on the input, and advance;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        check ')' is next on the input, and advance;
        print("label", L2);
        S(L1);
    }
    else /* other statement types */
}
```

Figure 5.31: On-the-fly recursive-descent code generation for while-statements

In Fig. 5.31, S and C now have no return value, since their only synthesized attributes are produced by printing. Further, the position of the print statements is significant. The order in which output is printed is: first `label` $L1$, then the code for C (which is the same as the value of *Ccode* in Fig. 5.29), then

https://hemanthrajhemu.github.io

`label` $L2$, and finally the code from the recursive call to S (which is the same as *Scode* in Fig. 5.29). Thus, the code printed by this call to S is exactly the same as the value of *Scode* that is returned in Fig. 5.29). □

Incidentally, we can make the same change to the underlying SDT: turn the construction of a main attribute into actions that emit the elements of that attribute. In Fig. 5.32 we see the SDT of Fig. 5.28 revised to generate code on the fly.

$$
\begin{array}{lll}
S & \rightarrow & \textbf{while} \; (\quad \{\; L1 = new(); \; L2 = new(); \; C.false = S.next; \\
 & & \qquad\qquad\; C.true = L2; \; print(\texttt{"label"}, L1); \;\} \\
 & & C \;) \qquad\;\; \{\; S_1.next = L1; \; print(\texttt{"label"}, L2); \;\} \\
 & & S_1
\end{array}
$$

Figure 5.32: SDT for on-the-fly code generation for while statements

5.5.3 L-Attributed SDD's and LL Parsing

Suppose that an L-attributed SDD is based on an LL-grammar and that we have converted it to an SDT with actions embedded in the productions, as described in Section 5.4.5. We can then perform the translation during LL parsing by extending the parser stack to hold actions and certain data items needed for attribute evaluation. Typically, the data items are copies of attributes.

In addition to records representing terminals and nonterminals, the parser stack will hold *action-records* representing actions to be executed and *synthesize-records* to hold the synthesized attributes for nonterminals. We use the following two principles to manage attributes on the stack:

- The inherited attributes of a nonterminal A are placed in the stack record that represents that nonterminal. The code to evaluate these attributes will usually be represented by an action-record immediately above the stack record for A; in fact, the conversion of L-attributed SDD's to SDT's ensures that the action-record will be immediately above A.

- The synthesized attributes for a nonterminal A are placed in a separate synthesize-record that is immediately below the record for A on the stack.

This strategy places records of several types on the parsing stack, trusting that these variant record types can be managed properly as subclasses of a "stack-record" class. In practice, we might combine several records into one, but the ideas are perhaps best explained by separating data used for different purposes into different records.

Action-records contain pointers to code to be executed. Actions may also appear in synthesize-records; these actions typically place copies of the synthesized attribute(s) in other records further down the stack, where the value of

https://hemanthrajhemu.github.io

that attribute will be needed after the synthesize-record and its attributes are popped off the stack.

Let us take a brief look at LL parsing to see the need to make temporary copies of attributes. From Section 4.4.4, a table-driven LL parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that $S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$, where S is the start symbol. When the parser expands by a production $A \rightarrow B\,C$, it replaces A on top of the stack by $B\,C$.

Suppose nonterminal C has an inherited attribute $C.i$. With $A \rightarrow B\,C$, the inherited attribute $C.i$ may depend not only on the inherited attributes of A, but on all the attributes of B. Thus, we may need to process B completely before $C.i$ can be evaluated. We therefore save temporary copies of all the attributes needed to evaluate $C.i$ in the action-record that evaluates $C.i$. Otherwise, when the parser replaces A on top of the stack by $B\,C$, the inherited attributes of A will have disappeared, along with its stack record.

Since the underlying SDD is L-attributed, we can be sure that the values of the inherited attributes of A are available when A rises to the top of the stack. The values will therefore be available in time to be copied into the action-record that evaluates the inherited attributes of C. Furthermore, space for the synthesized attributes of A is not a problem, since the space is in the synthesize-record for A, which remains on the stack, below B and C, when the parser expands by $A \rightarrow B\,C$.

As B is processed, we can perform actions (through a record just above B on the stack) that copy its inherited attributes for use by C, as needed, and after B is processed, the synthesize-record for B can copy its synthesized attributes for use by C, if needed. Likewise, synthesized attributes of A may need temporaries to help compute their value, and these can be copied to the synthesize-record for A as B and then C are processed. The principle that makes all this copying of attributes work is:

- All copying takes place among the records that are created during one expansion of one nonterminal. Thus, each of these records knows how far below it on the stack each other record is, and can write values into the records below safely.

The next example illustrates the implementation of inherited attributes during LL parsing by diligently copying attribute values. Shortcuts or optimizations are possible, particularly with copy rules, which simply copy the value of one attribute into another. Shortcuts are deferred until Example 5.24, which also illustrates synthesize-records.

Example 5.23: This example implements the the SDT of Fig. 5.32, which generates code on the fly for the while-production. This SDT does not have synthesized attributes, except for dummy attributes that represent labels.

Figure 5.33(a) shows the situation as we are about to use the while-production to expand S, presumably because the lookahead symbol on the input is

https://hemanthrajhemu.github.io

while. The record at the top of stack is for S, and it contains only the inherited attribute $S.next$, which we suppose has the value x. Since we are now parsing top-down, we show the stack top at the left, according to our usual convention.

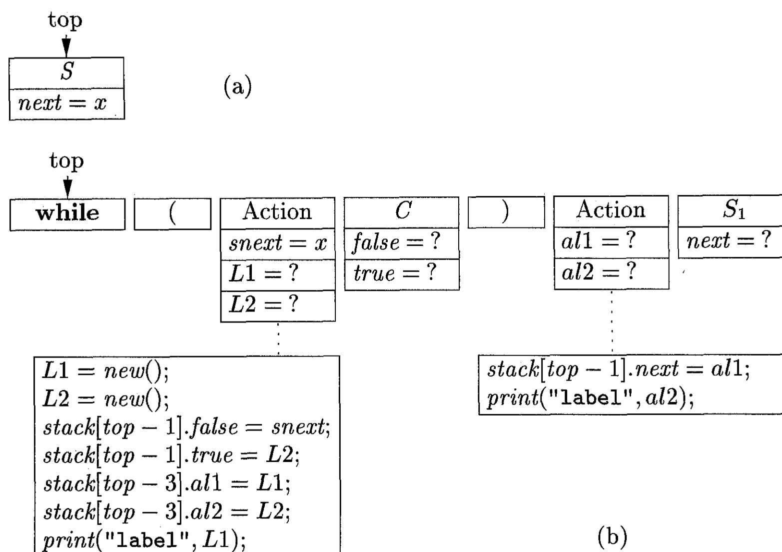


Figure 5.33: Expansion of S according to the while-statement production

Figure 5.33(b) shows the situation immediately after we have expanded S. There are action-records in front of the nonterminals C and S_1, corresponding to the actions in the underlying SDT of Fig. 5.32. The record for C has room for inherited attributes *true* and *false*, while the record for S_1 has room for attribute *next*, as all S-records must. We show values for these fields as ?, because we do not yet know their values.

The parser next recognizes **while** and (on the input and pops their records off the stack. Now, the first action is at the top, and it must be executed. This action-record has a field *snext*, which holds a copy of the inherited attribute $S.next$. When S is popped from the stack, the value of $S.next$ is copied into the field *snext* for use during the evaluation of the inherited attributes for C. The code for the first action generates new values for $L1$ and $L2$, which we shall suppose are y and z, respectively. The next step is to make z the value of $C.true$. The assignment $stack[top - 1].true = L2$ is written knowing it is only executed when this action-record is at the top of stack, so $top - 1$ refers to the record below it — the record for C.

The first action-record then copies $L1$ into field $al1$ in the second action, where it will be used to evaluate $S_1.next$. It also copies $L2$ into a field called $al2$ of the second action; this value is needed for that action-record to print its output properly. Finally, the first action-record prints `label` y to the output.

The situation after completing the first action and popping its record off

https://hemanthrajhemu.github.io

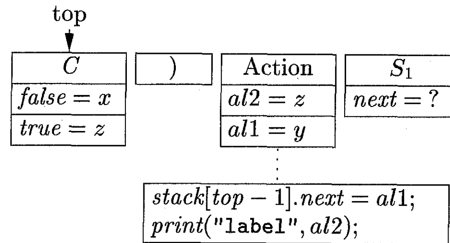


Figure 5.34: After the action above C is performed

the stack is shown in Fig. 5.34. The values of inherited attributes in the record for C have been filled in properly, as have the temporaries $al1$ and $al2$ in the second action record. At this point, C is expanded, and we presume that the code to implement its test containing jumps to labels x and z, as appropriate, is generated. When the C-record is popped from the stack, the record for) becomes top and causes the parser to check for) on its input.

With the action above S_1 at the top of the stack, its code sets $S_1.next$ and emits `label` z. When that is done, the record for S_1 becomes the top of stack, and as it is expanded, we presume it correctly generates code that implements whatever kind of statement it is and then jump to label y. □

Example 5.24: Now, let us consider the same while-statement, but with a translation that produces the output $S.code$ as a synthesized attribute, rather than by on-the-fly generation. In order to follow the explanation, it is useful to bear in mind the following invariant or inductive hypothesis, which we assume is followed for every nonterminal:

- Every nonterminal that has code associated with it leaves that code, as a string, in the synthesize-record just below it on the stack.

Assuming this statement is true, we shall handle the while-production so it maintains this statement as an invariant.

Figure 5.35(a) shows the situation just before S is expanded using the production for while-statements. At the top of the stack we see the record for S; it has a field for its inherited attribute $S.next$, as in Example 5.23. Immediately below that record is the synthesize-record for this occurrence of S. The latter has a field for $S.code$, as all synthesize-records for S must have. We also show it with some other fields for local storage and actions, since the SDT for the while production in Fig. 5.28 is surely part of a larger SDT.

Our expansion of S is based on the SDT of Fig. 5.28, and it is shown in Fig. 5.35(b). As a shortcut, during the expansion, we assume that the inherited attribute $S.next$ is assigned directly to $C.false$, rather than being placed in the first action and then copied into the record for C.

Let us examine what each record does when it becomes the top of stack. First, the **while** record causes the token **while** to be matched with the input,

https://hemanthrajhemu.github.io

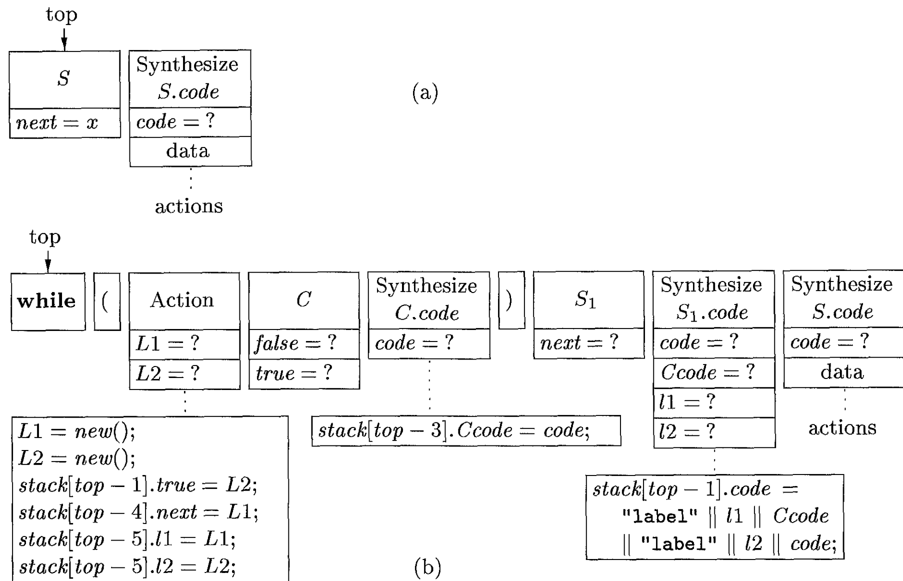


Figure 5.35: Expansion of S with synthesized attribute constructed on the stack

which it must, or else we would not have expanded S in this way. After **while** and (are popped off the stack, the code for the action-record is executed. It generates values for $L1$ and $L2$, and we take the shortcut of copying them directly to the inherited attributes that need them: $S_1.next$ and $C.true$. The last two steps of the action cause $L1$ and $L2$ to be copied into the record called "Synthesize $S_1.code$."

The synthesize-record for S_1 does double duty: not only will it hold the synthesized attribute $S_1.code$, but it will also serve as an action-record to complete the evaluation of the attributes for the entire production $S \rightarrow$ **while** (C) S_1. In particular, when it gets to the top, it will compute the synthesized attribute $S.code$ and place its value in the synthesize-record for the head S.

When C becomes the top of the stack, it has both its inherited attributes computed. By the inductive hypothesis stated above, we suppose it correctly generates code to execute its condition and jump to the proper label. We also assume that the actions performed during the expansion of C correctly place this code in the record below, as the value of synthesized attribute $C.code$.

After C is popped, the synthesize-record for $C.code$ becomes the top. Its code is needed in the synthesize-record for $S_1.code$, because that is where we concatenate all the code elements to form $S.code$. The synthesize-record for $C.code$ therefore has an action to copy $C.code$ into the synthesize-record for $S_1.code$. After doing so, the record for token) reaches the top of stack, and causes a check for) on the input. Assuming that test succeeds, the record for S_1 becomes the top of stack. By our inductive hypothesis, this nonterminal is

https://hemanthrajhemu.github.io

Can We Handle L-Attributed SDD's on LR Grammars?

In Section 5.4.1, we saw that every S-attributed SDD on an LR grammar can be implemented during a bottom-up parse. From Section 5.5.3 every L-attributed SDD on an LL grammar can be parsed top-down. Since LL grammars are a proper subset of the LR grammars, and the S-attributed SDD's are a proper subset of the L-attributed SDD's, can we handle every LR grammar and L-attributed SDD bottom-up?

We cannot, as the following intuitive argument shows. Suppose we have a production $A \rightarrow B\,C$ in an LR-grammar, and there is an inherited attribute $B.i$ that depends on inherited attributes of A. When we reduce to B, we still have not seen the input that C generates, so we cannot be sure that we have a body of production $A \rightarrow B\,C$. Thus, we cannot compute $B.i$ yet, since we are unsure whether to use the rule associated with this production.

Perhaps we could wait until we have reduced to C, and know that we must reduce $B\,C$ to A. However, even then, we do not know the inherited attributes of A, because even after reduction, we may not be sure of the production body that contains this A. We could reason that this decision, too, should be deferred, and therefore further defer the computation of $B.i$. If we keep reasoning this way, we soon realize that we cannot make any decisions until the entire input is parsed. Essentially, we have reached the strategy of "build the parse tree first and then perform the translation."

expanded, and the net effect is that its code is correctly constructed and placed in the field for *code* in the synthesize-record for S_1.

Now, all the data fields of the synthesize-record for S_1 have been filled in, so when it becomes the top of stack, the action in that record can be executed. The action causes the labels and code from $C.code$ and $S_1.code$ to be concatenated in the proper order. The resulting string is placed in the record below; that is, in the synthesize-record for S. We have now correctly computed $S.code$, and when the synthesize-record for S becomes the top, that code is available for placement in another record further down the stack, where it will eventually be assembled into a larger string of code implementing a program element of which this S is a part. □

5.5.4 Bottom-Up Parsing of L-Attributed SDD's

We can do bottom-up every translation that we can do top-down. More precisely, given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse. The "trick" has three parts:

https://hemanthrajhemu.github.io

1. Start with the SDT constructed as in Section 5.4.5, which places embedded actions before each nonterminal to compute its inherited attributes and an action at the end of the production to compute synthesized attributes.

2. Introduce into the grammar a marker nonterminal in place of each embedded action. Each such place gets a distinct marker, and there is one production for any marker M, namely $M \rightarrow \epsilon$.

3. Modify the action a if marker nonterminal M replaces it in some production $A \rightarrow \alpha \{a\} \beta$, and associate with $M \rightarrow \epsilon$ an action a' that

   (a) Copies, as inherited attributes of M, any attributes of A or symbols of α that action a needs.

   (b) Computes attributes in the same way as a, but makes those attributes be synthesized attributes of M.

   This change appears illegal, since typically the action associated with production $M \rightarrow \epsilon$ will have to access attributes belonging to grammar symbols that do not appear in this production. However, we shall implement the actions on the LR parsing stack, so the necessary attributes will always be available a known number of positions down the stack.

Example 5.25: Suppose that there is a production $A \rightarrow B\,C$ in an LL grammar, and the inherited attribute $B.i$ is computed from inherited attribute $A.i$ by some formula $B.i = f(A.i)$. That is, the fragment of an SDT we care about is

$$A \rightarrow \{B.i = f(A.i);\}\ B\ C$$

We introduce marker M with inherited attribute $M.i$ and synthesized attribute $M.s$. The former will be a copy of $A.i$ and the latter will be $B.i$. The SDT will be written

$$\begin{array}{lcl} A & \rightarrow & M\ B\ C \\ M & \rightarrow & \{M.i = A.i;\ M.s = f(M.i);\} \end{array}$$

Notice that the rule for M does not have $A.i$ available to it, but in fact we shall arrange that every inherited attribute for a nonterminal such as A appears on the stack immediately below where the reduction to A will later take place. Thus, when we reduce ϵ to M, we shall find $A.i$ immediately below it, from where it may be read. Also, the value of $M.s$, which is left on the stack along with M, is really $B.i$ and properly is found right below where the reduction to B will later occur. □

Example 5.26: Let us turn the SDT of Fig. 5.28 into an SDT that can operate with an LR parse of the revised grammar. We introduce a marker M before C and a marker N before S_1, so the underlying grammar becomes

https://hemanthrajhemu.github.io

Why Markers Work

Markers are nonterminals that derive only ϵ and that appear only once among all the bodies of all productions. We shall not give a formal proof that, when a grammar is LL, marker nonterminals can be added at any position in the body, and the resulting grammar will still be LR. The intuition, however, is as follows. If a grammar is LL, then we can determine that a string w on the input is derived from nonterminal A, in a derivation that starts with production $A \to \alpha$, by seeing only the first symbol of w (or the following symbol if $w = \epsilon$). Thus, if we parse w bottom-up, then the fact that a prefix of w must be reduced to α and then to S is known as soon as the beginning of w appears on the input. In particular, if we insert markers anywhere in α, the LR states will incorporate the fact that this marker has to be there, and will reduce ϵ to the marker at the appropriate point on the input.

$$\begin{array}{lcl} S & \to & \textbf{while} \ (\ M \ C \ ) \ N \ S_1 \\ M & \to & \epsilon \\ N & \to & \epsilon \end{array}$$

Before we discuss the actions that are associated with markers M and N, let us outline the "inductive hypothesis" about where attributes are stored.

1. Below the entire body of the while-production — that is, below **while** on the stack — will be the inherited attribute $S.next$. We may not know the nonterminal or parser state associated with this stack record, but we can be sure that it will have a field, in a fixed position of the record, that holds $S.next$ before we begin to recognize what is derived from this S.

2. Inherited attributes $C.true$ and $C.false$ will be just below the stack record for C. Since the grammar is presumed to be LL, the appearance of **while** on the input assures us that the while-production is the only one that can be recognized, so we can be sure that M will appear immediately below C on the stack, and M's record will hold the inherited attributes of C.

3. Similarly, the inherited attribute $S_1.next$ must appear immediately below S_1 on the stack, so we may place that attribute in the record for N.

4. The synthesized attribute $C.code$ will appear in the record for C. As always when we have a long string as an attribute value, we expect that in practice a pointer to (an object representing) the string will appear in the record, while the string itself is outside the stack.

5. Similarly, the synthesized attribute $S_1.code$ will appear in the record for S_1.

https://hemanthrajhemu.github.io

Let us follow the parsing process for a while-statement. Suppose that a record holding *S.next* appears on the top of the stack, and the next input is the terminal **while**. We shift this terminal onto the stack. It is then certain that the production being recognized is the while-production, so the LR parser can shift "(" and determine that its next step must be to reduce ϵ to M. The stack at this time is shown in Fig. 5.36. We also show in that figure the action that is associated with the reduction to M. We create values for $L1$ and $L2$, which live in fields of the M-record. Also in that record are fields for *C.true* and *C.false*. These attributes must be in the second and third fields of the record, for consistency with other stack records that might appear below C in other contexts and also must provide these attributes for C. The action completes by assigning values to *C.true* and *C.false*, one from the $L2$ just generated, and the other by reaching down the stack to where we know *S.next* is found.

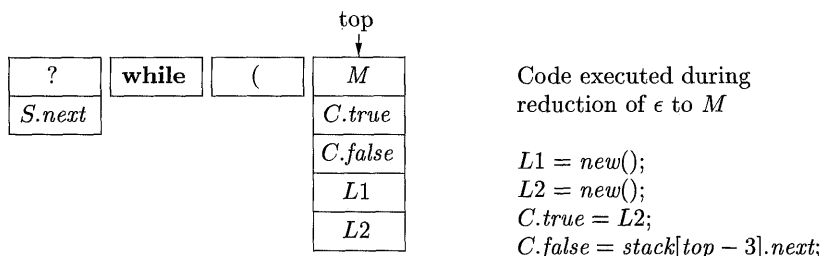


Figure 5.36: LR parsing stack after reduction of ϵ to M

We presume that the next inputs are properly reduced to C. The synthesized attribute *C.code* is therefore placed in the record for C. This change to the stack is shown in Fig. 5.37, which also incorporates the next several records that are later placed above C on the stack.

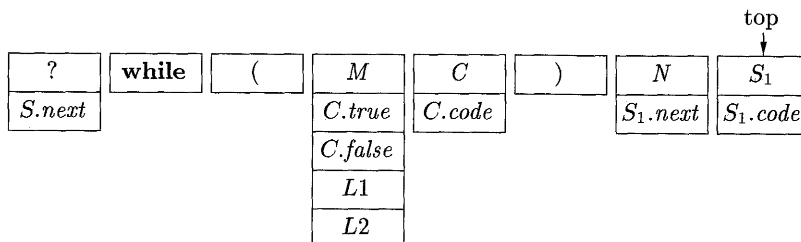


Figure 5.37: Stack just before reduction of the while-production body to S

Continuing with the recognition of the while-statement, the parser should next find ")" on the input, which it pushes onto the stack in a record of its own. At that point, the parser, which knows it is working on a while-statement because the grammar is LL, will reduce ϵ to N. The single piece of data associated with N is the inherited attribute $S_1.next$. Note that this attribute needs

https://hemanthrajhemu.github.io

to be in the record for N because that will be just below the record for S_1. The code that is executed to compute the value of $S_1.next$ is

$$S_1.next = stack[top - 3].L1;$$

This action reaches three records below N, which is at the top of stack when the code is executed, and retrieves the value of $L1$.

Next, the parser reduces some prefix of the remaining input to S, which we have consistently referred to as S_1 to distinguish it from the S at the head of the production. The value of $S_1.code$ is computed and appears in the stack record for S_1. This step takes us to the condition that is illustrated in Fig. 5.37.

At this point, the parser will reduce everything from **while** to S_1 to S. The code that is executed during this reduction is:

$$
\begin{array}{l}
tempCode = \textbf{label} \;\|\; stack[top - 4].L1 \;\|\; stack[top - 3].code \;\| \\
\quad \textbf{label} \;\|\; stack[top - 4].L2 \;\|\; stack[top].code; \\
top = top - 5; \\
stack[top].code = tempCode;
\end{array}
$$

That is, we construct the value of $S.code$ in a variable $tempCode$. That code is the usual, consisting of the two labels $L1$ and $L2$, the code for C and the code for S_1. The stack is popped, so S appears where **while** was. The value of the code for S is placed in the *code* field of that record, where it can be interpreted as the synthesized attribute $S.code$. Note that we do not show, in any of this discussion, the manipulation of LR states, which must also appear on the stack in the field that we have populated with grammar symbols. □

5.5.5 Exercises for Section 5.5

Exercise 5.5.1: Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.1.

Exercise 5.5.2: Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.2.

Exercise 5.5.3: Implement each of your SDD's of Exercise 5.4.4 with an LL parser in the style of Section 5.5.3, with code generated "on the fly."

Exercise 5.5.4: Implement each of your SDD's of Exercise 5.4.4 with an LL parser in the style of Section 5.5.3, but with code (or pointers to the code) stored on the stack.

Exercise 5.5.5: Implement each of your SDD's of Exercise 5.4.4 with an LR parser in the style of Section 5.5.4.

Exercise 5.5.6: Implement your SDD of Exercise 5.2.4 in the style of Section 5.5.1. Would an implementation in the style of Section 5.5.2 be any different?

https://hemanthrajhemu.github.io

5.6 Summary of Chapter 5

- *Inherited and Synthesized Attributes*: Syntax-directed definitions may use two kinds of attributes. A synthesized attribute at a parse-tree node is computed from attributes at its children. An inherited attribute at a node is computed from attributes at its parent and/or siblings.

- *Dependency Graphs*: Given a parse tree and an SDD, we draw edges among the attribute instances associated with each parse-tree node to denote that the value of the attribute at the head of the edge is computed in terms of the value of the attribute at the tail of the edge.

- *Cyclic Definitions*: In problematic SDD's, we find that there are some parse trees for which it is impossible to find an order in which we can compute all the attributes at all nodes. These parse trees have cycles in their associated dependency graphs. It is intractable to decide whether an SDD has such circular dependency graphs.

- *S-Attributed Definitions*: In an S-attributed SDD, all attributes are synthesized.

- *L-Attributed Definitions*: In an L-attributed SDD, attributes may be inherited or synthesized. However, inherited attributes at a parse-tree node may depend only on inherited attributes of its parent and on (any) attributes of siblings to its left.

- *Syntax Trees*: Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.

- *Implementing S-Attributed SDD's*: An S-attributed definition can be implemented by an SDT in which all actions are at the end of the production (a "postfix" SDT). The actions compute the synthesized attributes of the production head in terms of synthesized attributes of the symbols in the body. If the underlying grammar is LR, then this SDT can be implemented on the LR parser stack.

- *Eliminating Left Recursion From SDT's*: If an SDT has only side-effects (no attributes are computed), then the standard left-recursion-elimination algorithm for grammars allows us to carry the actions along as if they were terminals. When attributes are computed, we can still eliminate left recursion if the SDT is a postfix SDT.

- *Implementing L-attributed SDD's by Recursive-Descent Parsing*: If we have an L-attributed definition on a top-down parsable grammar, we can build a recursive-descent parser with no backtracking to implement the translation. Inherited attributes become arguments of the functions for their nonterminals, and synthesized attributes are returned by that function.

https://hemanthrajhemu.github.io

- *Implementing L-Attributed SDD's on an LL Grammar*: Every L-attributed definition with an underlying LL grammar can be implemented along with the parse. Records to hold the synthesized attributes for a nonterminal are placed below that nonterminal on the stack, while inherited attributes for a nonterminal are stored with that nonterminal on the stack. Action records are also placed on the stack to compute attributes at the appropriate time.

- *Implementing L-Attributed SDD's on an LL Grammar, Bottom-Up*: An L-attributed definition with an underlying LL grammar can be converted to a translation on an LR grammar and the translation performed in connection with a bottom-up parse. The grammar transformation introduces "marker" nonterminals that appear on the bottom-up parser's stack and hold inherited attributes of the nonterminal above it on the stack. Synthesized attributes are kept with their nonterminal on the stack.

5.7 References for Chapter 5

Syntax-directed definitions are a form of inductive definition in which the induction is on the syntactic structure. As such they have long been used informally in mathematics. Their application to programming languages came with the use of a grammar to structure the Algol 60 report.

The idea of a parser that calls for semantic actions can be found in Samelson and Bauer [8] and Brooker and Morris [1]. Irons [2] constructed one of the first syntax-directed compilers, using synthesized attributes. The class of L-attributed definitions comes from [6].

Inherited attributes, dependency graphs, and a test for circularity of SDD's (that is, whether or not there is some parse tree with no order in which the attributes can be computed) are from Knuth [5]. Jazayeri, Ogden, and Rounds [3] showed that testing circularity requires exponential time, as a function of the size of the SDD.

Parser generators such as Yacc [4] (see also the bibliographic notes in Chapter 4) support attribute evaluation during parsing.

The survey by Paakki [7] is a starting point for accessing the extensive literature on syntax-directed definitions and translations.

1. Brooker, R. A. and D. Morris, "A general translation program for phrase structure languages," *J. ACM* **9**:1 (1962), pp. 1–10.

2. Irons, E. T., "A syntax directed compiler for Algol 60," *Comm. ACM* **4**:1 (1961), pp. 51–55.

3. Jazayeri, M., W. F. Odgen, and W. C. Rounds, "The intrinsic exponential complexity of the circularity problem for attribute grammars," *Comm. ACM* **18**:12 (1975), pp. 697–706.

https://hemanthrajhemu.github.io

4. Johnson, S. C., "Yacc — Yet Another Compiler Compiler," Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Available at `http://dinosaur.compilertools.net/yacc/` .

5. Knuth, D.E., "Semantics of context-free languages," *Mathematical Systems Theory* **2**:2 (1968), pp. 127–145. See also *Mathematical Systems Theory* **5**:1 (1971), pp. 95–96.

6. Lewis, P. M. II, D. J. Rosenkrantz, and R. E. Stearns, "Attributed translations," *J. Computer and System Sciences* **9**:3 (1974), pp. 279–307.

7. Paakki, J., "Attribute grammar paradigms — a high-level methodology in language implementation," *Computing Surveys* **27**:2 (1995) pp. 196–255.

8. Samelson, K. and F. L. Bauer, "Sequential formula translation," *Comm. ACM* **3**:2 (1960), pp. 76–83.

https://hemanthrajhemu.github.io

Chapter 6

Intermediate-Code Generation

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just m front ends and n back ends.

This chapter deals with intermediate representations, static type checking, and intermediate code generation. For simplicity, we assume that a compiler front end is organized as in Fig. 6.1, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. We shall use the syntax-directed formalisms of Chapters 2 and 5 to specify checking and translation. Many of the translation schemes can be implemented during either bottom-up or top-down parsing, using the techniques of Chapter 5. All schemes can be implemented by creating a syntax tree and then walking the tree.

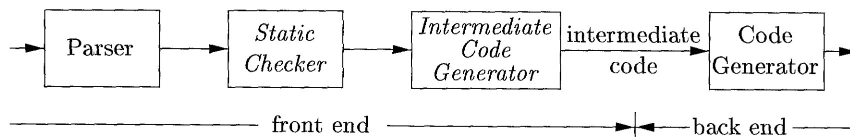


Figure 6.1: Logical structure of a compiler front end

Static checking includes *type checking*, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain

https://hemanthrajhemu.github.io

after parsing. For example, static checking assures that a break-statement in C is enclosed within a while-, for-, or switch-statement; an error is reported if such an enclosing statement does not exist.

The approach in this chapter can be used for a wide range of intermediate representations, including syntax trees and three-address code, both of which were introduced in Section 2.8. The term "three-address code" comes from instructions of the general form $x = y \ op \ z$ with three addresses: two for the operands y and z and one for the result x.

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as in Fig. 6.2. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.

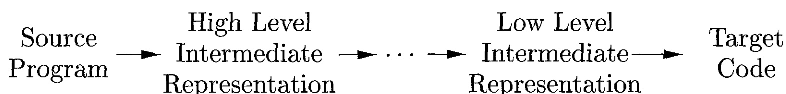


Figure 6.2: A compiler might use a sequence of intermediate representations

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high- to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial, as we shall see in Section 6.2.3. For looping statements, for example, a syntax tree represents the components of a statement, whereas three-address code contains labels and jump instructions to represent the flow of control, as in machine language.

The choice or design of an intermediate representation varies from compiler to compiler. An intermediate representation may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler. C is a programming language, yet it is often used as an intermediate form because it is flexible, it compiles into efficient machine code, and its compilers are widely available. The original C++ compiler consisted of a front end that generated C, treating a C compiler as a back end.

6.1 Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A directed acyclic graph (hereafter called a *DAG*) for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression. As we shall see in this section, DAG's can be constructed by using the same techniques that construct syntax trees.

https://hemanthrajhemu.github.io

6.1.1 Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1: Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```

The leaf for `a` has two parents, because `a` appears twice in the expression. More interestingly, the two occurrences of the common subexpression `b-c` are represented by one node, the node labeled −. That node has two parents, representing its two uses in the subexpressions `a*(b-c)` and `(b-c)*d`. Even though `b` and `c` appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression `b-c`. □

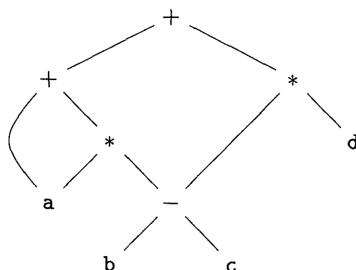


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions *Leaf* and *Node* created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, *Node*(*op*, *left*, *right*) we check whether there is already a node with label *op*, and children *left* and *right*, in that order. If so, *Node* returns the existing node; otherwise, it creates a new node.

Example 6.2: The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as

https://hemanthrajhemu.github.io

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \mathbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \mathbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \mathbf{id}$	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
6)	$T \rightarrow \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

discussed above. We assume that *entry-a* points to the symbol-table entry for a, and similarly for the other identifiers.

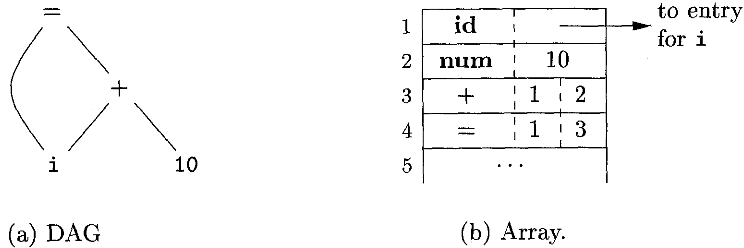
When the call to $Leaf(\mathbf{id}, entry\text{-}a)$ is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$. □

6.1.2 The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and

https://hemanthrajhemu.github.io

interior nodes have two additional fields indicating the left and right children.



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the *value number* for the node or for the expression represented by the node. For instance, in Fig. 6.6, the node labeled + has value number 3, and its left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its "value number." If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently; the next algorithm shows how.

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the *signature* of an interior node be the triple $\langle op, l, r\rangle$, where op is the label, l its left child's value number, and r its right child's value number. A unary operator may be assumed to have $r = 0$.

Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label op, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r\rangle$.

METHOD: Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l, and right child r, and return its value number. □

While Algorithm 6.3 yields the desired output, searching the entire array every time we are asked to locate one node is expensive, especially if the array holds expressions from an entire program. A more efficient approach is to use a hash table, in which the nodes are put into "buckets," each of which typically will have only a few nodes. The hash table is one of several data structures that support *dictionaries* efficiently.[1] A dictionary is an abstract data type that

[1]See Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, for a discussion of data structures supporting dictionaries.

https://hemanthrajhemu.github.io

allows us to insert and delete elements of a set, and to determine whether a given element is currently in the set. A good data structure for dictionaries, such as a hash table, performs each of these operations in time that is constant or close to constant, independent of the size of the set.

To construct a hash table for the nodes of a DAG, we need a *hash function* h that computes the index of the bucket for a signature $\langle op, l, r \rangle$, in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes. The bucket index $h(op, l, r)$ is computed deterministically from op, l, and r, so that we may repeat the calculation and always get to the same bucket index for node $\langle op, l, r \rangle$.

The buckets can be implemented as linked lists, as in Fig. 6.7. An array, indexed by hash value, holds the *bucket headers*, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node $\langle op, l, r \rangle$ can be found on the list whose header is at index $h(op, l, r)$ of the array.

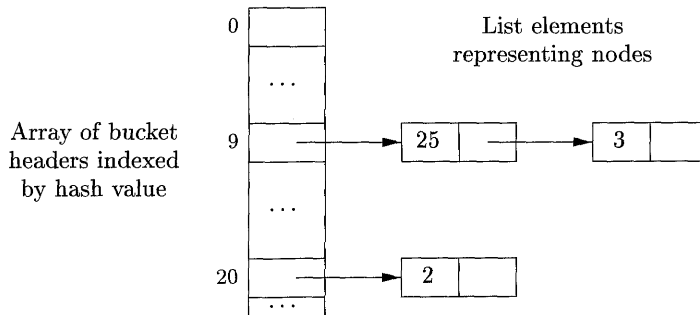


Figure 6.7: Data structure for searching buckets

Thus, given the input node op, l, and r, we compute the bucket index $h(op, l, r)$ and search the list of cells in this bucket for the given input node. Typically, there are enough buckets so that no list has more than a few cells. We may need to look at all the cells within a bucket, however, and for each value number v found in a cell, we must check whether the signature $\langle op, l, r \rangle$ of the input node matches the node with value number v in the list of cells (as in Fig. 6.7). If we find a match, we return v. If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket index $h(op, l, r)$, and return the value number in that new cell.

6.1.3 Exercises for Section 6.1

Exercise 6.1.1: Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

https://hemanthrajhemu.github.io

Exercise 6.1.2: Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming + associates from the left.

a) $a + b + (a + b)$.

b) $a + b + a + b$.

c) $a + a + \big((a + a + a + (a + a + a + a)\big)$.

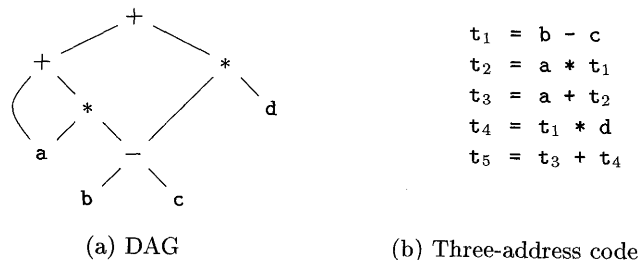
6.2 Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like `x+y*z` might be translated into the sequence of three-address instructions

```
t1 = y * z
t2 = x + t1
```

where t_1 and t_2 are compiler-generated temporary names. This unraveling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization, as discussed in Chapters 8 and 9. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

Example 6.4: Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence. □



(a) DAG (b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

https://hemanthrajhemu.github.io

6.2.1 Addresses and Instructions

Three-address code is built from two concepts: addresses and instructions. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses. Alternatively, three-address code can be implemented using records with fields for the addresses; records called quadruples and triples are discussed briefly in Section 6.2.2.

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

- *A constant.* In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered in Section 6.5.2.

- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

We now consider the common three-address instructions used in the rest of this book. Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "backpatching," discussed in Section 6.7. Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form $x = y \; op \; z$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form $x = op \; y$, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.

3. *Copy instructions* of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

https://hemanthrajhemu.github.io

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```
param x1
param x2
...
param xn
call p, n
```

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n," is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

8. Indexed copy instructions of the form x = y[i] and x[i] = y. The instruction x = y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i] = y sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form x = &y, x = $*y$, and $*x$ = y. The instruction x = &y sets the r-value of x to be the location (l-value) of y.[2] Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as `A[i][j]`, and x is a pointer name or temporary. In the instruction x = $*y$, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, $*x$ = y sets the r-value of the object pointed to by x to the r-value of y.

Example 6.5: Consider the statement

```
do i = i+1; while (a[i] < v);
```

Two possible translations of this statement are shown in Fig. 6.9. The translation in Fig. 6.9 uses a symbolic label `L`, attached to the first instruction. The

[2] From Section 2.8.3, l- and r-values are appropriate on the left and right sides of assignments, respectively.

https://hemanthrajhemu.github.io

translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication `i*8` is appropriate for an array of elements that each take 8 units of space. □

```
L:  t1 = i + 1                 100:  t1 = i + 1
    i = t1                     101:  i = t1
    t2 = i * 8                 102:  t2 = i * 8
    t3 = a [ t2 ]              103:  t3 = a [ t2 ]
    if t3 < v goto L           104:  if t3 < v goto 100
```

(a) Symbolic labels. (b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set clearly must be rich enough to implement the operations in the source language. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.

6.2.2 Quadruples

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure. In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called "quadruples," "triples," and "indirect triples."

A *quadruple* (or just "*quad*") has four fields, which we call *op*, arg_1, arg_2, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

1. Instructions with unary operators like x = `minus` y or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.

2. Operators like `param` use neither arg_2 nor *result*.

3. Conditional and unconditional jumps put the target label in *result*.

Example 6.6: Three-address code for the assignment `a = b*-c+b*-c;` appears in Fig. 6.10(a). The special operator `minus` is used to distinguish the

https://hemanthrajhemu.github.io

unary minus operator, as in `- c`, from the binary minus operator, as in `b - c`. Note that the unary-minus "three-address" statement has only two addresses, as does the copy statement $a = t_5$.

The quadruples in Fig. 6.10(b) implement the three-address code in (a). □

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

For readability, we use actual identifiers like a, b, and c in the fields arg_1, arg_2, and *result* in Fig. 6.10(b), instead of pointers to their symbol-table entries. Temporary names can either by entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class *Temp* with its own methods.

6.2.3 Triples

A *triple* has only three fields, which we call *op*, arg_1, and arg_2. Note that the *result* field in Fig. 6.10(b) is used primarily for temporary names. Using triples, we refer to the result of an operation $x \; op \; y$ by its position, rather than by an explicit temporary name. Thus, instead of the temporary t_1 in Fig. 6.10(b), a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself. In Section 6.1.2, positions or pointers to positions were called value numbers.

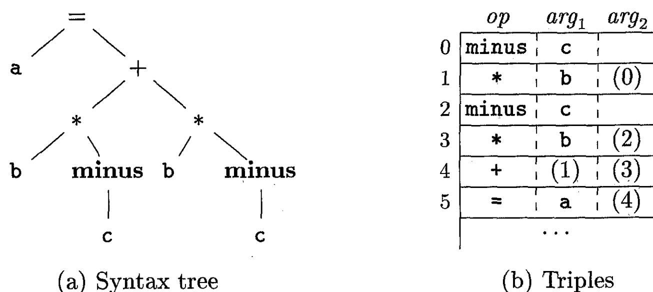
Triples are equivalent to signatures in Algorithm 6.3. Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

Example 6.7: The syntax tree and triples in Fig. 6.11 correspond to the three-address code and quadruples in Fig. 6.10. In the triple representation in Fig. 6.11(b), the copy statement $a = t_5$ is encoded in the triple representation by placing a in the arg_1 field and (4) in the arg_2 field. □

A ternary operation like $x[i] = y$ requires two entries in the triple structure; for example, we can put x and i in one triple and y in the next. Similarly, $x = y[i]$ can implemented by treating it as if it were the two instructions

https://hemanthrajhemu.github.io

> **Why Do We Need Copy Instructions?**
>
> A simple algorithm for translating expressions generates copy instructions for assignments, as in Fig. 6.10(a), where we copy t_5 into a rather than assigning $t_2 + t_4$ to a directly. Each subexpression typically gets its own, new temporary to hold its result, and only when the assignment operator = is processed do we learn where to put the value of the complete expression. A code-optimization pass, perhaps using the DAG of Section 6.1.1 as an intermediate form, can discover that t_5 can be replaced by a.



(a) Syntax tree

	op	*arg*$_1$	*arg*$_2$
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

t = y[i] and x = t, where t is a compiler-generated temporary. Note that the temporary t does not actually appear in a triple, since temporary values are referred to by their position in the triple structure.

A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with indirect triples, which we consider next.

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. For example, let us use an array *instruction* to list pointers to triples in the desired order. Then, the triples in Fig. 6.11(b) might be represented as in Fig. 6.12.

With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

https://hemanthrajhemu.github.io

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

Figure 6.12: Indirect triples representation of three-address code

6.2.4 Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term *static single-assigment.* Figure 6.13 shows the same intermediate program in three-address code and in static single-assignment form. Note that subscripts distinguish each definition of variables p and q in the SSA representation.

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

$$\begin{aligned} p_1 &= a + b \\ q_1 &= p_1 - c \\ p_2 &= q_1 * d \\ p_3 &= e - p_2 \\ q_2 &= p_3 + q_1 \end{aligned}$$

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

has two control-flow paths in which the variable x gets defined. If we use different names for x in the true part and the false part of the conditional statement, then which name should we use in the assignment y = x * a? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the ϕ-function to combine the two definitions of x:

$$\begin{aligned} &\texttt{if (flag) } x_1 = -1\texttt{; else } x_2 = 1\texttt{;} \\ &x_3 = \phi(x_1, x_2)\texttt{;} \end{aligned}$$

https://hemanthrajhemu.github.io

Here, $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true part of the conditional and the value x_2 if the control flow passes through the false part. That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

6.2.5 Exercises for Section 6.2

Exercise 6.2.1: Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

Exercise 6.2.2: Repeat Exercise 6.2.1 for the following assignment statements:

i. `a = b[i] + c[j].`

ii. `a[i] = b*c - b*d.`

iii. `x = f(y+1) + 2.`

iv. `x = *p + &y.`

! **Exercise 6.2.3:** Show how to transform a three-address code sequence into one in which each defined variable gets a unique variable name.

6.3 Types and Declarations

The applications of types can be grouped under checking and translation:

- *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator. For example, the `&&` operator in Java expects its two operands to be booleans; the result is also of type boolean.

- *Translation Applications.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

https://hemanthrajhemu.github.io

In this section, we examine types and storage layout for names declared within a procedure or a class. The actual storage for a procedure call or an object is allocated at run time, when the procedure is called or the object is created. As we examine local declarations at compile time, we can, however, lay out *relative addresses*, where the relative address of a name or a component of a data structure is an offset from the start of a data area.

6.3.1 Type Expressions

Types have structure, which we shall represent using *type expressions*: a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression. The sets of basic types and constructors depend on the language to be checked.

Example 6.8: The array type `int[2][3]` can be read as "array of 2 arrays of 3 integers each" and written as a type expression *array*(2, *array*(3, *integer*)). This type is represented by the tree in Fig. 6.14. The operator *array* takes two parameters, a number and a type. □

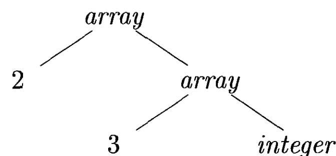


Figure 6.14: Type expression for `int[2][3]`

We shall use the following definition of type expressions:

- A basic type is a type expression. Typical basic types for a language include *boolean*, *char*, *integer*, *float*, and *void*; the latter denotes "the absence of a value."

- A type name is a type expression.

- A type expression can be formed by applying the *array* type constructor to a number and a type expression.

- A record is a data structure with named fields. A type expression can be formed by applying the *record* type constructor to the field names and their types. Record types will be implemented in Section 6.3.6 by applying the constructor *record* to a symbol table containing entries for the fields.

- A type expression can be formed by using the type constructor $\rightarrow$ for function types. We write $s \rightarrow t$ for "function from type s to type t." Function types will be useful when type checking is discussed in Section 6.5.

https://hemanthrajhemu.github.io

Type Names and Recursive Types

Once a class is defined, its name can be used as a type name in C++ or Java; for example, consider `Node` in the program fragment

```
public class Node { ··· }
...
public Node n;
```

Names can be used to define recursive types, which are needed for data structures such as linked lists. The pseudocode for a list element

```
class Cell { int info; Cell next; ··· }
```

defines the recursive type `Cell` as a class that contains a field `info` and a field `next` of type `Cell`. Similar recursive types can be defined in C using records and pointers. The techniques in this chapter carry over to recursive types.

- If s and t are type expressions, then their Cartesian product $s \times t$ is a type expression. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters). We assume that $\times$ associates to the left and that it has higher precedence than $\rightarrow$.
- Type expressions may contain variables whose values are type expressions. Compiler-generated type variables will be used in Section 6.5.4.

A convenient way to represent a type expression is to use a graph. The value-number method of Section 6.1.2, can be adapted to construct a dag for a type expression, with interior nodes for type constructors and leaves for basic types, type names, and type variables; for example, see the tree in Fig. 6.14.[3]

6.3.2 Type Equivalence

When are two type expressions equivalent? Many type-checking rules have the form, "**if** two type expressions are equal **then** return a certain type **else** error." Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions. The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

[3]Since type names denote type expressions, they can set up implicit cycles; see the box on "Type Names and Recursive Types." If edges to type names are redirected to the type expressions denoted by the names, then the resulting graph can have cycles due to recursive types.

https://hemanthrajhemu.github.io

When type expressions are represented by graphs, two types are *structurally equivalent* if and only if one of the following conditions is true:

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

If type names are treated as standing for themselves, then the first two conditions in the above definition lead to *name equivalence* of type expressions.

Name-equivalent expressions are assigned the same value number, if we use Algorithm 6.3. Structural equivalence can be tested using the unification algorithm in Section 6.5.5.

6.3.3 Declarations

We shall study types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can be handled as discussed in Example 5.10. The grammar is

$$\begin{array}{lcl} D & \rightarrow & T\ \textbf{id}\ ;\ D\ \mid\ \epsilon \\ T & \rightarrow & B\ C\ \mid\ \textbf{record}\ \text{'\{'}\ D\ \text{'\}'} \\ B & \rightarrow & \textbf{int}\ \mid\ \textbf{float} \\ C & \rightarrow & \epsilon\ \mid\ [\ \textbf{num}\ ]\ C \end{array}$$

The fragment of the above grammar that deals with basic and array types was used to illustrate inherited attributes in Section 5.3.2. The difference in this section is that we consider storage layout as well as types.

Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types. Nonterminal B generates one of the basic types **int** and **float**. Nonterminal C, for "component," generates strings of zero or more integers, each integer surrounded by brackets. An array type consists of a basic type specified by B, followed by array components specified by nonterminal C. A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

6.3.4 Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name. Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data. Run-time storage management is discussed in Chapter 7.

https://hemanthrajhemu.github.io

Address Alignment

The storage layout for data objects is strongly influenced by the addressing constraints of the target machine. For example, instructions to add integers may expect integers to be *aligned*, that is, placed at certain positions in memory such as an address divisible by 4. Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes — the next multiple of 4 — leaving 2 bytes unused. Space left unused due to alignment considerations is referred to as *padding*. When space is at a premium, a compiler may *pack* data so that no padding is left; additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

Suppose that storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. Typically, a byte is eight bits, and some number of bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of the first byte.

The *width* of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes.[4]

The translation scheme (SDT) in Fig. 6.15 computes types and their widths for basic and array types; record types will be discussed in Section 6.3.6. The SDT uses synthesized attributes *type* and *width* for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production $C \rightarrow \epsilon$. In a syntax-directed definition, t and w would be inherited attributes for C.

The body of the T-production consists of nonterminal B, an action, and nonterminal C, which appears on the next line. The action between B and C sets t to $B.type$ and w to $B.width$. If $B \rightarrow$ **int** then $B.type$ is set to *integer* and $B.width$ is set to 4, the width of an integer. Similarly, if $B \rightarrow$ **float** then $B.type$ is *float* and $B.width$ is 8, the width of a float.

The productions for C determine whether T generates a basic type or an array type. If $C \rightarrow \epsilon$, then t becomes $C.type$ and w becomes $C.width$.

Otherwise, C specifies an array component. The action for $C \rightarrow$ [**num**] C_1 forms $C.type$ by applying the type constructor *array* to the operands **num**.*value* and $C_1.type$. For instance, the result of applying *array* might be a tree structure such as Fig. 6.14.

[4]Storage allocation for pointers in C and C++ is simpler if all pointers have the same width. The reason is that the storage for a pointer may need to be allocated before we learn the type of the objects it can point to.

https://hemanthrajhemu.github.io

$$
\begin{array}{lll}
T & \rightarrow & B \\
 & & C \quad \{\ t = B.type;\ w = B.width;\ \} \\
B & \rightarrow & \textbf{int} \quad \{\ B.type = integer;\ B.width = 4;\ \} \\
B & \rightarrow & \textbf{float} \quad \{\ B.type = float;\ B.width = 8;\ \} \\
C & \rightarrow & \epsilon \quad \{\ C.type = t;\ C.width = w;\ \} \\
C & \rightarrow & [\ \textbf{num}\ ]\ C_1 \quad \{\ array(\textbf{num}.value,\ C_1.type); \\
 & & \qquad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Figure 6.15: Computing types and their widths

The width of an array is obtained by multiplying the width of an element by the number of elements in the array. If addresses of consecutive integers differ by 4, then address calculations for an array of integers will include multiplications by 4. Such multiplications provide opportunities for optimization, so it is helpful for the front end to make them explicit. In this chapter, we ignore other machine dependencies such as the alignment of data objects on word boundaries.

Example 6.9: The parse tree for the type `int[2][3]` is shown by dotted lines in Fig. 6.16. The solid lines show how the type and width are passed from B, down the chain of C's through variables t and w, and then back up the chain as synthesized attributes *type* and *width*. The variables t and w are assigned the values of $B.type$ and $B.width$, respectively, before the subtree with the C nodes is examined. The values of t and w are used at the node for $C \rightarrow \epsilon$ to start the evaluation of the synthesized attributes up the chain of C nodes. □

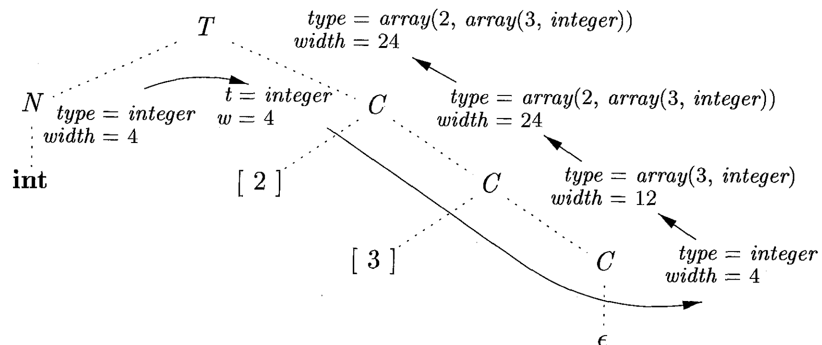


Figure 6.16: Syntax-directed translation of array types

https://hemanthrajhemu.github.io

6.3.5 Sequences of Declarations

Languages such as C and Java allow all the declarations in a single procedure to be processed as a group. The declarations may be distributed within a Java procedure, but they can still be processed when the procedure is analyzed. Therefore, we can use a variable, say *offset*, to keep track of the next available relative address.

The translation scheme of Fig. 6.17 deals with a sequence of declarations of the form T **id**, where T generates a type as in Fig. 6.15. Before the first declaration is considered, *offset* is set to 0. As each new name x is seen, x is entered into the symbol table with its relative address set to the current value of *offset*, which is then incremented by the width of the type of x.

$$
\begin{array}{lll}
P \rightarrow & & \{ \; \mathit{offset} = 0; \; \} \\
& D & \\
D \rightarrow & T \; \mathbf{id} \; ; & \{ \; \mathit{top.put}(\mathbf{id}.\mathit{lexeme}, \; T.\mathit{type}, \; \mathit{offset}); \\
& & \quad \mathit{offset} \; = \; \mathit{offset} + T.\mathit{width}; \; \} \\
& D_1 & \\
D \rightarrow & \epsilon &
\end{array}
$$

Figure 6.17: Computing the relative addresses of declared names

The semantic action within the production $D \rightarrow T\,\mathbf{id}\,;\,D_1$ creates a symbol-table entry by executing *top.put*(**id**.*lexeme*, *T.type*, *offset*). Here *top* denotes the current symbol table. The method *top.put* creates a symbol-table entry for **id**.*lexeme*, with type *T.type* and relative address *offset* in its data area.

The initialization of *offset* in Fig. 6.17 is more evident if the first production appears on one line as:

$$P \;\rightarrow\; \{ \; \mathit{offset} = 0; \; \} \;\; D \tag{6.1}$$

Nonterminals generating ϵ, called marker nonterminals, can be used to rewrite productions so that all actions appear at the ends of right sides; see Section 5.5.4. Using a marker nonterminal M, (6.1) can be restated as:

$$
\begin{array}{lcll}
P & \rightarrow & M \; D & \\
M & \rightarrow & \epsilon & \{ \; \mathit{offset} = 0; \; \}
\end{array}
$$

6.3.6 Fields in Records and Classes

The translation of declarations in Fig. 6.17 carries over to fields in records and classes. Record types can be added to the grammar in Fig. 6.15 by adding the following production

$$T \;\rightarrow\; \mathbf{record} \; '\{' \; D \; '\}'$$

https://hemanthrajhemu.github.io

The fields in this record type are specified by the sequence of declarations generated by D. The approach of Fig. 6.17 can be used to determine the types and relative addresses of fields, provided we are careful about two things:

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D.
- The offset or relative address for a field name is relative to the data area for that record.

Example 6.10: The use of a name x for a field within a record does not conflict with other uses of the name outside the record. Thus, the three uses of x in the following declarations are distinct and do not conflict with each other:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

A subsequent assignment `x = p.x+q.x;` sets variable x to the sum of the fields named x in the records p and q. Note that the relative address of x in p differs from the relative address of x in q. □

For convenience, record types will encode both the types and relative addresses of their fields, using a symbol table for the record type. A record type has the form *record*(t), where *record* is the type constructor, and t is a symbol-table object that holds information about the fields of this record type.

The translation scheme in Fig. 6.18 consists of a single production to be added to the productions for T in Fig. 6.15. This production has two semantic actions. The embedded action before D saves the existing symbol table, denoted by *top* and sets *top* to a fresh symbol table. It also saves the current *offset*, and sets *offset* to 0. The declarations generated by D will result in types and relative addresses being put in the fresh symbol table. The action after D creates a record type using *top*, before restoring the saved symbol table and offset.

$$
\begin{array}{lll}
T \rightarrow & \textbf{record}\ '\{' & \{\ \mathit{Env.push}(\mathit{top});\ \mathit{top} = \textbf{new}\ \mathit{Env}(); \\
& & \quad \mathit{Stack.push}(\mathit{offset});\ \mathit{offset} = 0;\ \} \\
& D\ '\}' & \{\ T.\mathit{type} = \mathit{record}(\mathit{top});\ T.\mathit{width} = \mathit{offset}; \\
& & \quad \mathit{top} = \mathit{Env.pop}();\ \mathit{offset} = \mathit{Stack.pop}();\ \}
\end{array}
$$

Figure 6.18: Handling of field names in records

For concreteness, the actions in Fig. 6.18 give pseudocode for a specific implementation. Let class *Env* implement symbol tables. The call *Env.push*(*top*) pushes the current symbol table denoted by *top* onto a stack. Variable *top* is then set to a new symbol table. Similarly, *offset* is pushed onto a stack called *Stack*. Variable *offset* is then set to 0.

https://hemanthrajhemu.github.io

After the declarations in D have been translated, the symbol table *top* holds the types and relative addresses of the fields in this record. Further, *offset* gives the storage needed for all the fields. The second action sets $T.type$ to *record*(*top*) and $T.width$ to *offset*. Variables *top* and *offset* are then restored to their pushed values to complete the translation of this record type.

This discussion of storage for record types carries over to classes, since no storage is reserved for methods. See Exercise 6.3.2.

6.3.7 Exercises for Section 6.3

Exercise 6.3.1: Determine the types and relative addresses for the identifiers in the following sequence of declarations:

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

! **Exercise 6.3.2:** Extend the handling of field names in Fig. 6.18 to classes and single-inheritance class hierarchies.

a) Give an implementation of class *Env* that allows linked symbol tables, so that a subclass can either redefine a field name or refer directly to a field name in a superclass.

b) Give a translation scheme that allocates a contiguous data area for the fields in a class, including inherited fields. Inherited fields must maintain the relative addresses they were assigned in the layout for the superclass.

6.4 Translation of Expressions

The rest of this chapter explores issues that arise during the translation of expressions and statements. We begin in this section with the translation of expressions into three-address code. An expression with more than one operator, like $a + b * c$, will translate into instructions with at most one operator per instruction. An array reference $A[i][j]$ will expand into a sequence of three-address instructions that calculate an address for the reference. We shall consider type checking of expressions in Section 6.5 and the use of boolean expressions to direct the flow of control through a program in Section 6.6.

6.4.1 Operations Within Expressions

The syntax-directed definition in Fig. 6.19 builds up the three-address code for an assignment statement S using attribute *code* for S and attributes *addr* and *code* for an expression E. Attributes $S.code$ and $E.code$ denote the three-address code for S and E, respectively. Attribute $E.addr$ denotes the address that will

https://hemanthrajhemu.github.io

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \parallel$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp\,()$ $E.code = E_1.code \parallel$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Figure 6.19: Three-address code for expressions

hold the value of E. Recall from Section 6.2.1 that an address can be a name, a constant, or a compiler-generated temporary.

Consider the last production, $E \rightarrow \textbf{id}$, in the syntax-directed definition in Fig. 6.19. When an expression is a single identifier, say x, then x itself holds the value of the expression. The semantic rules for this production define $E.addr$ to point to the symbol-table entry for this instance of **id**. Let *top* denote the current symbol table. Function *top.get* retrieves the entry when it is applied to the string representation **id**.*lexeme* of this instance of **id**. $E.code$ is set to the empty string.

When $E \rightarrow (\ E_1\ )$, the translation of E is the same as that of the subexpression E_1. Hence, $E.addr$ equals $E_1.addr$, and $E.code$ equals $E_1.code$.

The operators + and unary - in Fig. 6.19 are representative of the operators in a typical language. The semantic rules for $E \rightarrow E_1 + E_2$, generate code to compute the value of E from the values of E_1 and E_2. Values are computed into newly generated temporary names. If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$, then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. $E.addr$ is set to t. A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing **new** *Temp*().

For convenience, we use the notation $gen(x\ '='\ y\ '+'\ z)$ to represent the three-address instruction $x = y + z$. Expressions appearing in place of variables like x, y, and z are evaluated when passed to *gen*, and quoted strings like $'='$ are taken literally.[5] Other three-address instructions will be built up similarly

[5]In syntax-directed definitions, *gen* builds an instruction and returns it. In translation schemes, *gen* builds an instruction and incrementally emits it by putting it into the stream

https://hemanthrajhemu.github.io

by applying *gen* to a combination of expressions and strings.

When we translate the production $E \rightarrow E_1 + E_2$, the semantic rules in Fig. 6.19 build up $E.code$ by concatenating $E_1.code$, $E_2.code$, and an instruction that adds the values of E_1 and E_2. The instruction puts the result of the addition into a new temporary name for E, denoted by $E.addr$.

The translation of $E \rightarrow - E_1$ is similar. The rules create a new temporary for E and generate an instruction to perform the unary minus operation.

Finally, the production $S \rightarrow \textbf{id} = E;$ generates instructions that assign the value of expression E to the identifier **id**. The semantic rule for this production uses function *top.get* to determine the address of the identifier represented by **id**, as in the rules for $E \rightarrow \textbf{id}$. $S.code$ consists of the instructions to compute the value of E into an address given by $E.addr$, followed by an assignment to the address *top.get*(**id**.*lexeme*) for this instance of **id**.

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

$$
\begin{aligned}
& t_1 = \texttt{minus c} \\
& t_2 = \texttt{b} + t_1 \\
& \texttt{a} = t_2
\end{aligned}
$$

□

6.4.2 Incremental Translation

Code attributes can be long strings, so they are usually generated incrementally, as discussed in Section 5.5.2. Thus, instead of building up $E.code$ as in Fig. 6.19, we can arrange to generate only the new three-address instructions, as in the translation scheme of Fig. 6.20. In the incremental approach, *gen* not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far. The sequence may either be retained in memory for further processing, or it may be output incrementally.

The translation scheme in Fig. 6.20 generates the same code as the syntax-directed definition in Fig. 6.19. With the incremental approach, the *code* attribute is not used, since there is a single sequence of instructions that is created by successive calls to *gen*. For example, the semantic rule for $E \rightarrow E_1 + E_2$ in Fig. 6.20 simply calls *gen* to generate an add instruction; the instructions to compute E_1 into $E_1.addr$ and E_2 into $E_2.addr$ have already been generated.

The approach of Fig. 6.20 can also be used to build a syntax tree. The new semantic action for $E \rightarrow E_1 + E_2$ creates a node by using a constructor, as in

$$E \rightarrow E_1 + E_2 \quad \{ \; E.addr = \textbf{new}\ Node('+', E_1.addr, E_2.addr); \; \}$$

Here, attribute *addr* represents the address of a node rather than a variable or constant.

of generated instructions.

https://hemanthrajhemu.github.io

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E \;; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
E & \rightarrow & E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \;\; gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
 & | & - \; E_1 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \;\; gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \} \\
 & | & (\; E_1 \;) & \{ \; E.addr = E_1.addr; \; \} \\
 & | & \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \}
\end{array}
$$

Figure 6.20: Generating three-address code for expressions incrementally

6.4.3 Addressing Array Elements

Array elements can be accessed quickly if they are stored in a block of consecutive locations. In C and Java, array elements are numbered $0, 1, \ldots, n-1$, for an array with n elements. If the width of each array element is w, then the ith element of array A begins in location

$$base + i \times w \tag{6.2}$$

where *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of $A[0]$.

The formula (6.2) generalizes to two or more dimensions. In two dimensions, we write $A[i_1][i_2]$ in C and Java for element i_2 in row i_1. Let w_1 be the width of a row and let w_2 be the width of an element in a row. The relative address of $A[i_1][i_2]$ can then be calculated by the formula

$$base + i_1 \times w_1 + i_2 \times w_2 \tag{6.3}$$

In k dimensions, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \tag{6.4}$$

where w_j, for $1 \leq j \leq k$, is the generalization of w_1 and w_2 in (6.3).

Alternatively, the relative address of an array reference can be calculated in terms of the numbers of elements n_j along dimension j of the array and the width $w = w_k$ of a single element of the array. In two dimensions (i.e., $k = 2$ and $w = w_2$), the location for $A[i_1][i_2]$ is given by

$$base + (i_1 \times n_2 + i_2) \times w \tag{6.5}$$

In k dimensions, the following formula calculates the same address as (6.4):

$$base + ((\cdots(i_1 \times n_2 + i_2) \times n_3 + i_3) \cdots) \times n_k + i_k) \times w \tag{6.6}$$

https://hemanthrajhemu.github.io

More generally, array elements need not be numbered starting at 0. In a one-dimensional array, the array elements are numbered $low, low + 1, \ldots, high$ and $base$ is the relative address of $A[low]$. Formula (6.2) for the address of $A[i]$ is replaced by:

$$base + (i - low) \times w \tag{6.7}$$

The expressions (6.2) and (6.7) can be both be rewritten as $i \times w + c$, where the subexpression $c = base - low \times w$ can be precalculated at compile time. Note that $c = base$ when low is 0. We assume that c is saved in the symbol table entry for A, so the relative address of $A[i]$ is obtained by simply adding $i \times w$ to c.

Compile-time precalculation can also be applied to address calculations for elements of multidimensional arrays; see Exercise 6.4.5. However, there is one situation where we cannot use compile-time precalculation: when the array's size is dynamic. If we do not know the values of low and $high$ (or their generalizations in many dimensions) at compile time, then we cannot compute constants such as c. Then, formulas like (6.7) must be evaluated as they are written, when the program executes.

The above address calculations are based on row-major layout for arrays, which is used in C and Java. A two-dimensional array is normally stored in one of two forms, either *row-major* (row-by-row) or *column-major* (column-by-column). Figure 6.21 shows the layout of a 2×3 array A in (a) row-major form and (b) column-major form. Column-major form is used in the Fortran family of languages.

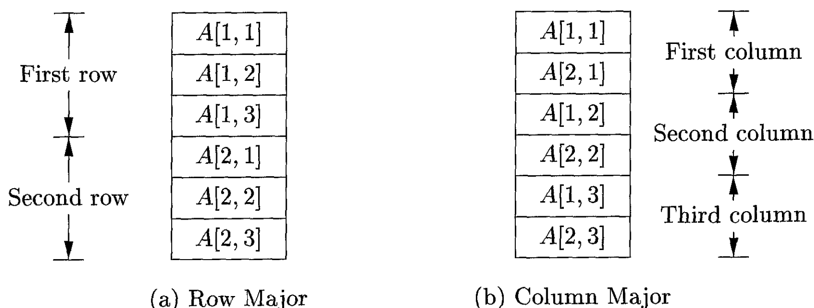


Figure 6.21: Layouts for a two-dimensional array.

We can generalize row- or column-major form to many dimensions. The generalization of row-major form is to store the elements in such a way that, as we scan down a block of storage, the rightmost subscripts appear to vary fastest, like the numbers on an odometer. Column-major form generalizes to the opposite arrangement, with the leftmost subscripts varying fastest.

https://hemanthrajhemu.github.io

6.4.4 Translation of Array References

The chief problem in generating code for array references is to relate the address-calculation formulas in Section 6.4.3 to a grammar for array references. Let nonterminal L generate an array name followed by a sequence of index expressions:

$$L \rightarrow L \; [\; E\;] \;\mid\; \textbf{id} \; [\; E\;]$$

As in C and Java, assume that the lowest-numbered array element is 0. Let us calculate addresses based on widths, using the formula (6.4), rather than on numbers of elements, as in (6.6). The translation scheme in Fig. 6.22 generates three-address code for expressions with array references. It consists of the productions and semantic actions from Fig. 6.20, together with productions involving nonterminal L.

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E \; ; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\
 & \mid & L = E \; ; & \{ \; gen(L.addr.base \; '[' \; L.addr \; ']' \; '=' \; E.addr); \; \} \\
E & \rightarrow & E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \;\; gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\
 & \mid & \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \} \\
 & \mid & L & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \;\; gen(E.addr \; '=' \; L.array.base \; '[' \; L.addr \; ']'); \; \} \\
L & \rightarrow & \textbf{id} \; [\; E \;] & \{ \; L.array = top.get(\textbf{id}.lexeme); \\
 & & & \;\; L.type = L.array.type.elem; \\
 & & & \;\; L.addr = \textbf{new} \; Temp\,(); \\
 & & & \;\; gen(L.addr \; '=' \; E.addr \; '*' \; L.type.width); \; \} \\
 & \mid & L_1 \; [\; E \;] & \{ \; L.array = L_1.array; \\
 & & & \;\; L.type = L_1.type.elem; \\
 & & & \;\; t = \textbf{new} \; Temp\,(); \\
 & & & \;\; L.addr = \textbf{new} \; Temp\,(); \\
 & & & \;\; gen(t \; '=' \; E.addr \; '*' \; L.type.width); \; \} \\
 & & & \;\; gen(L.addr \; '=' \; L_1.addr \; '+' \; t); \; \}
\end{array}
$$

Figure 6.22: Semantic actions for array references

Nonterminal L has three synthesized attributes:

1. $L.addr$ denotes a temporary that is used while computing the offset for the array reference by summing the terms $i_j \times w_j$ in (6.4).

https://hemanthrajhemu.github.io

2. $L.array$ is a pointer to the symbol-table entry for the array name. The base address of the array, say, $L.array.base$ is used to determine the actual l-value of an array reference after all the index expressions are analyzed.

3. $L.type$ is the type of the subarray generated by L. For any type t, we assume that its width is given by $t.width$. We use types as attributes, rather than widths, since types are needed anyway for type checking. For any array type t, suppose that $t.elem$ gives the element type.

The production $S \rightarrow \mathbf{id} = E$; represents an assignment to a nonarray variable, which is handled as usual. The semantic action for $S \rightarrow L = E$; generates an indexed copy instruction to assign the value denoted by expression E to the location denoted by the array reference L. Recall that attribute $L.array$ gives the symbol-table entry for the array. The array's base address — the address of its 0th element — is given by $L.array.base$. Attribute $L.addr$ denotes the temporary that holds the offset for the array reference generated by L. The location for the array reference is therefore $L.array.base[L.addr]$. The generated instruction copies the r-value from address $E.addr$ into the location for L.

Productions $E \rightarrow E_1 + E_2$ and $E \rightarrow \mathbf{id}$ are the same as before. The semantic action for the new production $E \rightarrow L$ generates code to copy the value from the location denoted by L into a new temporary. This location is $L.array.base[L.addr]$, as discussed above for the production $S \rightarrow L = E$;. Again, attribute $L.array$ gives the array name, and $L.array.base$ gives its base address. Attribute $L.addr$ denotes the temporary that holds the offset. The code for the array reference places the r-value at the location designated by the base and offset into a new temporary denoted by $E.addr$.

Example 6.12: Let `a` denote a 2×3 array of integers, and let `c`, `i`, and `j` all denote integers. Then, the type of `a` is $array(2, array(3, integer))$. Its width w is 24, assuming that the width of an integer is 4. The type of `a[i]` is $array(3, integer)$, of width $w_1 = 12$. The type of `a[i][j]` is $integer$.

An annotated parse tree for the expression `c + a[i][j]` is shown in Fig. 6.23. The expression is translated into the sequence of three-address instructions in Fig. 6.24. As usual, we have used the name of each identifier to refer to its symbol-table entry. □

6.4.5 Exercises for Section 6.4

Exercise 6.4.1: Add to the translation of Fig. 6.19 rules for the following productions:

a) $E \rightarrow E_1 * E_2$.

b) $E \rightarrow + E_1$ (unary plus).

Exercise 6.4.2: Repeat Exercise 6.4.1 for the incremental translation of Fig. 6.20.

https://hemanthrajhemu.github.io

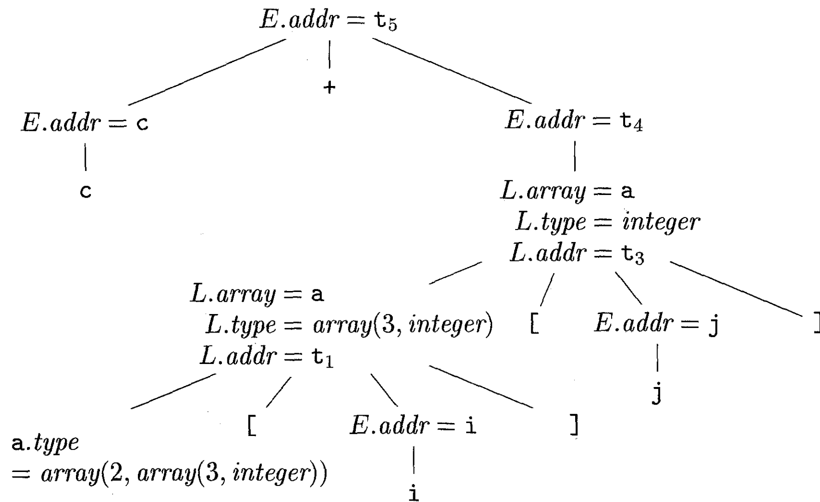


Figure 6.23: Annotated parse tree for `c+a[i][j]`

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Figure 6.24: Three-address code for expression `c+a[i][j]`

Exercise 6.4.3: Use the translation of Fig. 6.22 to translate the following assignments:

a) `x = a[i] + b[j].`

b) `x = a[i][j] + b[i][j].`

! c) `x = a[b[i][j]][c[k]].`

! **Exercise 6.4.4:** Revise the translation of Fig. 6.22 for array references of the Fortran style, that is, $\mathbf{id}[E_1, E_2, \ldots, E_n]$ for an n-dimensional array.

Exercise 6.4.5: Generalize formula (6.7) to multidimensional arrays, and indicate what values can be stored in the symbol table and used to compute offsets. Consider the following cases:

a) An array A of two dimensions, in row-major form. The first dimension has indexes running from l_1 to h_1, and the second dimension has indexes from l_2 to h_2. The width of a single array element is w.

https://hemanthrajhemu.github.io

Symbolic Type Widths

The intermediate code should be relatively independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine. However, as we have described the calculation of type widths, an assumption regarding how basic types is built into the translation scheme. For instance, Example 6.12 assumes that each element of an integer array takes four bytes. Some intermediate codes, e.g., P-code for Pascal, leave it to the code generator to fill in the size of array elements, so the intermediate code is independent of the size of a machine word. We could have done the same in our translation scheme if we replaced 4 (as the width of an integer) by a symbolic constant.

b) The same as (a), but with the array stored in column-major form.

! c) An array A of k dimensions, stored in row-major form, with elements of size w. The jth dimension has indexes running from l_j to h_j.

! d) The same as (c) but with the array stored in column-major form.

Exercise 6.4.6: An integer array $A[i, j]$ has index i ranging from 1 to 10 and index j ranging from 1 to 20. Integers take 4 bytes each. Suppose array A is stored starting at byte 0. Find the location of:

a) $A[4, 5]$ b)$A[10, 8]$ c) $A[3, 17]$.

Exercise 6.4.7: Repeat Exercise 6.4.6 if A is stored in column-major order.

Exercise 6.4.8: A real array $A[i, j, k]$ has index i ranging from 1 to 4, index j ranging from 0 to 4, and index k ranging from 5 to 10. Reals take 8 bytes each. Suppose array A is stored starting at byte 0. Find the location of:

a) $A[3, 4, 5]$ b)$A[1, 2, 7]$ c) $A[4, 3, 9]$.

Exercise 6.4.9: Repeat Exercise 6.4.8 if A is stored in column-major order.

6.5 Type Checking

To do *type checking* a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language.

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an

https://hemanthrajhemu.github.io

element along with the value of the element. A *sound* type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

Besides their use for compiling, ideas from type checking have been used to improve the security of systems that allow software modules to be imported and executed. Java programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes. Imported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

6.5.1 Rules for Type Checking

Type checking can take on two forms: synthesis and inference. *Type synthesis* builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2. A typical rule for type synthesis has the form

$$\begin{array}{l}\textbf{if } f \text{ has type } s \rightarrow t \textbf{ and } x \text{ has type } s, \\ \textbf{then} \text{ expression } f(x) \text{ has type } t\end{array} \tag{6.8}$$

Here, f and x denote expressions, and $s \rightarrow t$ denotes a function from s to t. This rule for functions with one argument carries over to functions with several arguments. The rule (6.8) can be adapted for $E_1 + E_2$ by viewing it as a function application $add(E_1, E_2)$.[6]

Type inference determines the type of a language construct from the way it is used. Looking ahead to the examples in Section 6.5.4, let *null* be a function that tests whether a list is empty. Then, from the usage $null(x)$, we can tell that x must be a list. The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.

Variables representing type expressions allow us to talk about unknown types. We shall use Greek letters $\alpha, \beta, \cdots$ for type variables in type expressions.

A typical rule for type inference has the form

$$\begin{array}{l}\textbf{if } f(x) \text{ is an expression,} \\ \textbf{then} \text{ for some } \alpha \text{ and } \beta,\ f \text{ has type } \alpha \rightarrow \beta \textbf{ and } x \text{ has type } \alpha\end{array} \tag{6.9}$$

Type inference is needed for languages like ML, which check types, but do not require names to be declared.

---

[6]We shall use the term "synthesis" even if some context information is used to determine types. With overloaded functions, where the same name is given to more than one function, the context of $E_1 + E_2$ may also need to be considered in some languages.

https://hemanthrajhemu.github.io

In this section, we consider type checking of expressions. The rules for checking statements are similar to those for expressions. For example, we treat the conditional statement "**if** (E) S;" as if it were the application of a function *if* to E and S. Let the special type *void* denote the absence of a value. Then function *if* expects to be applied to a *boolean* and a *void*; the result of the application is a *void*.

6.5.2 Type Conversions

Consider expressions like $x + i$, where x is of type float and i is of type integer. Since the representation of integers and floating-point numbers is different within a computer and different machine instructions are used for operations on integers and floats, the compiler may need to convert one of the operands of $+$ to ensure that both operands are of the same type when the addition occurs.

Suppose that integers are converted to floats when necessary, using a unary operator `(float)`. For example, the integer 2 is converted to a float in the code for the expression `2 * 3.14`:

$$\begin{array}{l} \mathtt{t_1\ =\ (float)\ 2} \\ \mathtt{t_2\ =\ t_1\ *\ 3.14} \end{array}$$

We can extend such examples to consider integer and float versions of the operators; for example, `int*` for integer operands and `float*` for floats.

Type synthesis will be illustrated by extending the scheme in Section 6.4.2 for translating expressions. We introduce another attribute $E.type$, whose value is either *integer* or *float*. The rule associated with $E \rightarrow E_1 + E_2$ builds on the pseudocode

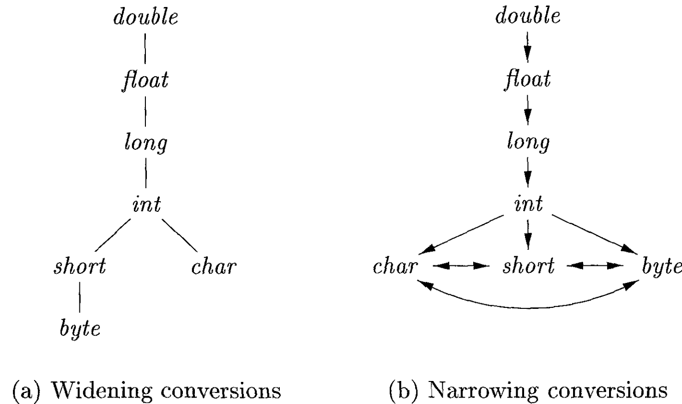
> **if** ($E_1.type = integer$ **and** $E_2.type = integer$) $E.type = integer$;
> **else if** ($E_1.type = float$ **and** $E_2.type = integer$) $\cdots$
> $\cdots$

As the number of types subject to conversion increases, the number of cases increases rapidly. Therefore with large numbers of types, careful organization of the semantic actions becomes important.

Type conversion rules vary from language to language. The rules for Java in Fig. 6.25 distinguish between *widening* conversions, which are intended to preserve information, and *narrowing* conversions, which can lose information. The widening rules are given by the hierarchy in Fig. 6.25(a): any type lower in the hierarchy can be widened to a higher type. Thus, a *char* can be widened to an *int* or to a *float*, but a *char* cannot be widened to a *short*. The narrowing rules are illustrated by the graph in Fig. 6.25(b): a type s can be narrowed to a type t if there is a path from s to t. Note that *char*, *short*, and *byte* are pairwise convertible to each other.

Conversion from one type to another is said to be *implicit* if it is done automatically by the compiler. Implicit type conversions, also called *coercions*,

https://hemanthrajhemu.github.io



(a) Widening conversions (b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

are limited in many languages to widening conversions. Conversion is said to be *explicit* if the programmer must write something to cause the conversion. Explicit conversions are also called *casts*.

The semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result. Pseudocode for *widen*, assuming that the only types are *integer* and *float*, appears in Fig. 6.26.

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Figure 6.26: Pseudocode for function *widen*

https://hemanthrajhemu.github.io

The semantic action for $E \to E_1 + E_2$ in Fig. 6.27 illustrates how type conversions can be added to the scheme in Fig. 6.20 for translating expressions. In the semantic action, temporary variable a_1 is either $E_1.addr$, if the type of E_1 does not need to be converted to the type of E, or a new temporary variable returned by *widen* if this conversion is necessary. Similarly, a_2 is either $E_2.addr$ or a new temporary holding the type-converted value of E_2. Neither conversion is needed if both types are *integer* or both are *float*. In general, however, we could find that the only way to add values of two different types is to convert them both to a third type.

$$
\begin{array}{lll}
E \;\to\; E_1 + E_2 & \{ & E.type \;=\; max(E_1.type, E_2.type); \\
 & & a_1 \;=\; widen(E_1.addr, E_1.type, E.type); \\
 & & a_2 \;=\; widen(E_2.addr, E_2.type, E.type); \\
 & & E.addr = \mathbf{new}\ Temp\,(); \\
 & & gen(E.addr\ '='\ a_1\ '+'\ a_2); \ \}
\end{array}
$$

Figure 6.27: Introducing type conversions into expression evaluation

6.5.3 Overloading of Functions and Operators

An *overloaded* symbol has different meanings depending on its context. Overloading is *resolved* when a unique meaning is determined for each occurrence of a name. In this section, we restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in Java.

Example 6.13: The + operator in Java denotes either string concatenation or addition, depending on the types of its operands. User-defined functions can be overloaded as well, as in

```
void err() { ··· }
void err(String s) { ··· }
```

Note that we can choose between these two versions of a function `err` by looking at their arguments. □

The following is a type-synthesis rule for overloaded functions:

$$
\begin{array}{l}
\textbf{if } f \text{ can have type } s_i \to t_i, \text{ for } 1 \leq i \leq n, \text{ where } s_i \neq s_j \text{ for } i \neq j \\
\textbf{and } x \text{ has type } s_k, \text{ for some } 1 \leq k \leq n \\
\textbf{then } \text{expression } f(x) \text{ has type } t_k
\end{array}
\qquad (6.10)
$$

The value-number method of Section 6.1.2 can be applied to type expressions to resolve overloading based on argument types, efficiently. In a DAG representing a type expression, we assign an integer index, called a value number, to each node. Using Algorithm 6.3, we construct a signature for a node,

https://hemanthrajhemu.github.io

consisting of its label and the value numbers of its children, in order from left to right. The signature for a function consists of the function name and the types of its arguments. The assumption that we can resolve overloading based on the types of arguments is equivalent to saying that we can resolve overloading based on signatures.

It is not always possible to resolve overloading by looking only at the arguments of a function. In Ada, instead of a single type, a subexpression standing alone may have a set of possible types for which the context must provide sufficient information to narrow the choice down to a single type (see Exercise 6.5.2).

6.5.4 Type Inference and Polymorphic Functions

Type inference is useful for a language like ML, which is strongly typed, but does not require names to be declared before they are used. Type inference ensures that names are used consistently.

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types. In this section, we consider *parametric polymorphism*, where the polymorphism is characterized by parameters or type variables. The running example is the ML program in Fig. 6.28, which defines a function *length*. The type of *length* can be described as, "for any type α, *length* maps a list of elements of type α to an integer."

fun *length*(x) $=$
        if *null*(x) **then** 0 **else** *length*(*tl*(x)) $+ 1$;

Figure 6.28: ML program for the length of a list

Example 6.14: In Fig. 6.28, the keyword **fun** introduces a function definition; functions can be recursive. The program fragment defines function *length* with one parameter x. The body of the function consists of a conditional expression. The predefined function *null* tests whether a list is empty, and the predefined function *tl* (short for "tail") returns the remainder of a list after the first element is removed.

The function *length* determines the length or number of elements of a list x. All elements of a list must have the same type, but *length* can be applied to lists whose elements are of any one type. In the following expression, *length* is applied to two different types of lists (list elements are enclosed within "[" and "]"):

$$length(["\texttt{sun}", "\texttt{mon}", "\texttt{tue}"]) + length([10, 9, 8, 7]) \quad (6.11)$$

The list of strings has length 3 and the list of integers has length 4, so expression (6.11) evaluates to 7. □

https://hemanthrajhemu.github.io

Using the symbol $\forall$ (read as "for any type") and the type constructor *list*, the type of *length* can be written as

$$\forall \alpha.\ list(\alpha) \rightarrow integer \tag{6.12}$$

The $\forall$ symbol is the *universal quantifier*, and the type variable to which it is applied is said to be *bound* by it. Bound variables can be renamed at will, provided all occurrences of the variable are renamed. Thus, the type expression

$$\forall \beta.\ list(\beta) \rightarrow integer$$

is equivalent to (6.12). A type expression with a $\forall$ symbol in it will be referred to informally as a "polymorphic type."

Each time a polymorphic function is applied, its bound type variables can denote a different type. During type checking, at each use of a polymorphic type we replace the bound variables by fresh variables and remove the universal quantifiers.

The next example informally infers a type for *length*, implicitly using type inference rules like (6.9), which is repeated here:

> **if** $f(x)$ is an expression,
> **then** for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α

Example 6.15: The abstract syntax tree in Fig. 6.29 represents the definition of *length* in Fig. 6.28. The root of the tree, labeled **fun**, represents the function definition. The remaining nonleaf nodes can be viewed as function applications. The node labeled + represents the application of the operator + to a pair of children. Similarly, the node labeled **if** represents the application of an operator **if** to a triple formed by its children (for type checking, it does not matter that either the **then** or the **else** part will be evaluated, but not both).

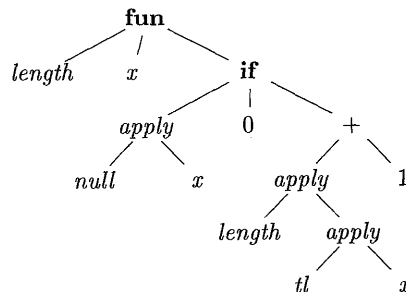


Figure 6.29: Abstract syntax tree for the function definition in Fig. 6.28

From the body of function *length*, we can infer its type. Consider the children of the node labeled **if**, from left to right. Since *null* expects to be applied to lists, x must be a list. Let us use variable α as a placeholder for the type of the list elements; that is, x has type "list of α."

https://hemanthrajhemu.github.io

Substitutions, Instances, and Unification

If t is a type expression and S is a substitution (a mapping from type variables to type expressions), then we write $S(t)$ for the result of consistently replacing all occurrences of each type variable α in t by $S(\alpha)$. $S(t)$ is called an *instance* of t. For example, $list(integer)$ is an instance of $list(\alpha)$, since it is the result of substituting *integer* for α in $list(\alpha)$. Note, however, that $integer \rightarrow float$ is not an instance of $\alpha \rightarrow \alpha$, since a substitution must replace all occurrences of α by the same type expression.

Substitution S is a *unifier* of type expressions t_1 and t_2 if $S(t_1) = S(t_2)$. S is the *most general unifier* of t_1 and t_2 if for any other unifier of t_1 and t_2, say S', it is the case that for any t, $S'(t)$ is an instance of $S(t)$. In words, S' imposes more constraints on t than S does.

If $null(x)$ is true, then $length(x)$ is 0. Thus, the type of *length* must be "function from list of α to integer." This inferred type is consistent with the usage of *length* in the else part, $length(tl(x)) + 1$. □

Since variables can appear in type expressions, we have to re-examine the notion of equivalence of types. Suppose E_1 of type $s \rightarrow s'$ is applied to E_2 of type t. Instead of simply determining the equality of s and t, we must "unify" them. Informally, we determine whether s and t can be made structurally equivalent by replacing the type variables in s and t by type expressions.

A *substitution* is a mapping from type variables to type expressions. We write $S(t)$ for the result of applying the substitution S to the variables in type expression t; see the box on "Substitutions, Instances, and Unification." Two type expressions t_1 and t_2 *unify* if there exists some substitution S such that $S(t_1) = S(t_2)$. In practice, we are interested in the most general unifier, which is a substitution that imposes the fewest constraints on the variables in the expressions. See Section 6.5.5 for a unification algorithm.

Algorithm 6.16: Type inference for polymorphic functions.

INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.

OUTPUT: Inferred types for the names in the program.

METHOD: For simplicity, we shall deal with unary functions only. The type of a function $f(x_1, x_2)$ with two parameters can be represented by a type expression $s_1 \times s_2 \rightarrow t$, where s_1 and s_2 are the types of x_1 and x_2, respectively, and t is the type of the result $f(x_1, x_2)$. An expression $f(a, b)$ can be checked by matching the type of a with s_1 and the type of b with s_2.

https://hemanthrajhemu.github.io

Check the function definitions and the expression in the input sequence. Use the inferred type of a function if it is subsequently used in an expression.

- For a function definition **fun** $\mathbf{id}_1(\mathbf{id}_2) = E$, create fresh type variables α and β. Associate the type $\alpha \rightarrow \beta$ with the function $\mathbf{id}_1$, and the type α with the parameter $\mathbf{id}_2$. Then, infer a type for expression E. Suppose α denotes type s and β denotes type t after type inference for E. The inferred type of function $\mathbf{id}_1$ is $s \rightarrow t$. Bind any type variables that remain unconstrained in $s \rightarrow t$ by $\forall$ quantifiers.

- For a function application $E_1(E_2)$, infer types for E_1 and E_2. Since E_1 is used as a function, its type must have the form $s \rightarrow s'$. (Technically, the type of E_1 must unify with $\beta \rightarrow \gamma$, where β and γ are new type variables). Let t be the inferred type of E_1. Unify s and t. If unification fails, the expression has a type error. Otherwise, the inferred type of $E_1(E_2)$ is s'.

- For each occurrence of a polymorphic function, replace the bound variables in its type by distinct fresh variables and remove the $\forall$ quantifiers. The resulting type expression is the inferred type of this occurrence.

- For a name that is encountered for the first time, introduce a fresh variable for its type.

□

Example 6.17: In Fig. 6.30, we infer a type for function *length*. The root of the syntax tree in Fig. 6.29 is for a function definition, so we introduce variables β and γ, associate the type $\beta \rightarrow \gamma$ with function *length*, and the type β with x; see lines 1-2 of Fig. 6.30.

At the right child of the root, we view **if** as a polymorphic function that is applied to a triple, consisting of a boolean and two expressions that represent the **then** and **else** parts. Its type is $\forall \alpha.\ boolean \times \alpha \times \alpha \rightarrow \alpha$.

Each application of a polymorphic function can be to a different type, so we make up a fresh variable α_i (where i is from "if") and remove the $\forall$; see line 3 of Fig. 6.30. The type of the left child of **if** must unify with *boolean*, and the types of its other two children must unify with α_i.

The predefined function *null* has type $\forall \alpha.\ list(\alpha) \rightarrow boolean$. We use a fresh type variable α_n (where n is for "null") in place of the bound variable α; see line 4. From the application of *null* to x, we infer that the type β of x must match $list(\alpha_n)$; see line 5.

At the first child of **if**, the type *boolean* for $null(x)$ matches the type expected by **if**. At the second child, the type α_i unifies with *integer*; see line 6.

Now, consider the subexpression $length(tl(x)) + 1$. We make up a fresh variable α_t (where t is for "tail") for the bound variable α in the type of *tl*; see line 8. From the application $tl(x)$, we infer $list(\alpha_t) = \beta = list(\alpha_n)$; see line 9.

Since $length(tl(x))$ is an operand of +, its type γ must unify with *integer*; see line 10. It follows that the type of *length* is $list(\alpha_n) \rightarrow integer$. After the

https://hemanthrajhemu.github.io

LINE	EXPRESSION : TYPE	UNIFY
1)	*length* : $\beta \to \gamma$	
2)	x : β	
3)	**if** : $boolean \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	*null* : $list(\alpha_n) \to boolean$	
5)	$null(x)$: $boolean$	$list(\alpha_n) = \beta$
6)	0 : $integer$	$\alpha_i = integer$
7)	+ : $integer \times integer \to integer$	
8)	*tl* : $list(\alpha_t) \to list(\alpha_t)$	
9)	$tl(x)$: $list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x))$: γ	$\gamma = integer$
11)	1 : $integer$	
12)	$length(tl(x)) + 1$: $integer$	
13)	**if**$(\ \cdots\ )$: $integer$	

Figure 6.30: Inferring a type for the function *length* of Fig. 6.28

function definition is checked, the type variable α_n remains in the type of *length*. Since no assumptions were made about α_n, any type can be substituted for it when the function is used. We therefore make it a bound variable and write

$$\forall \alpha_n.\ list(\alpha_n) \to integer$$

for the type of *length*. □

6.5.5 An Algorithm for Unification

Informally, unification is the problem of determining whether two expressions s and t can be made identical by substituting expressions for the variables in s and t. Testing equality of expressions is a special case of unification; if s and t have constants but no variables, then s and t unify if and only if they are identical. The unification algorithm in this section extends to graphs with cycles, so it can be used to test structural equivalence of circular types.[7]

We shall implement a graph-theoretic formulation of unification, where types are represented by graphs. Type variables are represented by leaves and type constructors are represented by interior nodes. Nodes are grouped into equivalence classes; if two nodes are in the same equivalence class, then the type expressions they represent must unify. Thus, all interior nodes in the same class must be for the same type constructor, and their corresponding children must be equivalent.

Example 6.18: Consider the two type expressions

[7] In some applications, it is an error to unify a variable with an expression containing that variable. Algorithm 6.19 permits such substitutions.

https://hemanthrajhemu.github.io

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

The following substitution S is the most general unifier for these expressions

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

This substitution maps the two type expressions to the following expression

$$((\alpha_1 \rightarrow \alpha_2) \;\times\; list(\alpha_1)) \rightarrow list(\alpha_2)$$

The two expressions are represented by the two nodes labeled $\rightarrow$: 1 in Fig. 6.31. The integers at the nodes indicate the equivalence classes that the nodes belong to after the nodes numbered 1 are unified. □

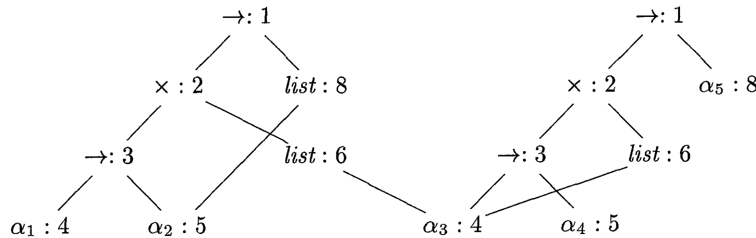


Figure 6.31: Equivalence classes after unification

Algorithm 6.19: Unification of a pair of nodes in a type graph.

INPUT: A graph representing a type and a pair of nodes m and n to be unified.

OUTPUT: Boolean value true if the expressions represented by the nodes m and n unify; false, otherwise.

METHOD: A node is implemented by a record with fields for a binary operator and pointers to the left and right children. The sets of equivalent nodes are maintained using the *set* field. One node in each equivalence class is chosen to be the unique representative of the equivalence class by making its *set* field contain a null pointer. The *set* fields of the remaining nodes in the equivalence class will point (possibly indirectly through other nodes in the set) to the representative. Initially, each node n is in an equivalence class by itself, with n as its own representative node.

The unification algorithm, shown in Fig. 6.32, uses the following two operations on nodes:

https://hemanthrajhemu.github.io

```
boolean unify(Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s1 and s2 and
                        t is an op-node with children t1 and t2) {
                union(s, t);
                return unify(s1, t1) and unify(s2, t2);
        }
        else if s or t represents a variable {
                union(s, t);
                return true;
        }
        else return false;
}
```

Figure 6.32: Unification algorithm.

- *find*(n) returns the representative node of the equivalence class currently containing node n.

- *union*(m, n) merges the equivalence classes containing nodes m and n. If one of the representatives for the equivalence classes of m and n is a non-variable node, *union* makes that nonvariable node be the representative for the merged equivalence class; otherwise, *union* makes one or the other of the original representatives be the new representative. This asymmetry in the specification of *union* is important because a variable cannot be used as the representative for an equivalence class for an expression containing a type constructor or basic type. Otherwise, two inequivalent expressions may be unified through that variable.

The *union* operation on sets is implemented by simply changing the *set* field of the representative of one equivalence class so that it points to the representative of the other. To find the equivalence class that a node belongs to, we follow the *set* pointers of nodes until the representative (the node with a null pointer in the set field) is reached.

Note that the algorithm in Fig. 6.32 uses $s = find(m)$ and $t = find(n)$ rather than m and n, respectively. The representative nodes s and t are equal if m and n are in the same equivalence class. If s and t represent the same basic type, the call *unify*(m, n) returns true. If s and t are both interior nodes for a binary type constructor, we merge their equivalence classes on speculation and recursively check that their respective children are equivalent. By merging first, we decrease the number of equivalence classes before recursively checking the children, so the algorithm terminates.

https://hemanthrajhemu.github.io

The substitution of an expression for a variable is implemented by adding the leaf for the variable to the equivalence class containing the node for that expression. Suppose either m or n is a leaf for a variable. Suppose also that this leaf has been put into an equivalence class with a node representing an expression with a type constructor or a basic type. Then *find* will return a representative that reflects that type constructor or basic type, so that a variable cannot be unified with two different expressions. □

Example 6.20: Suppose that the two expressions in Example 6.18 are represented by the initial graph in Fig. 6.33, where each node is in its own equivalence class. When Algorithm 6.19 is applied to compute *unify*(1, 9), it notes that nodes 1 and 9 both represent the same operator. It therefore merges 1 and 9 into the same equivalence class and calls *unify*(2, 10) and *unify*(8, 14). The result of computing *unify*(1, 9) is the graph previously shown in Fig. 6.31. □

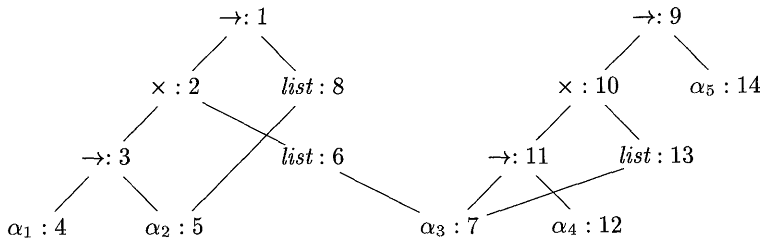


Figure 6.33: Initial graph with each node in its own equivalence class

If Algorithm 6.19 returns true, we can construct a substitution S that acts as the unifier, as follows. For each variable α, *find*(α) gives the node n that is the representative of the equivalence class of α. The expression represented by n is $S(\alpha)$. For example, in Fig. 6.31, we see that the representative for α_3 is node 4, which represents α_1. The representative for α_5 is node 8, which represents $list(\alpha_2)$. The resulting substitution S is as in Example 6.18.

6.5.6 Exercises for Section 6.5

Exercise 6.5.1: Assuming that function *widen* in Fig. 6.26 can handle any of the types in the hierarchy of Fig. 6.25(a), translate the expressions below. Assume that c and d are characters, s and t are short integers, i and j are integers, and x is a float.

a) `x = s + c.`

b) `i = s + c.`

c) `x = (s + c) * (t + d).`

https://hemanthrajhemu.github.io

Exercise 6.5.2: As in Ada, suppose that each expression must have a unique type, but that from a subexpression, by itself, all we can deduce is a set of possible types. That is, the application of function E_1 to argument E_2, represented by $E \rightarrow E_1 \,(\, E_2\,)$, has the associated rule

$$E.type \;=\; \{\; t \mid \text{ for some } s \text{ in } E_2.type,\; s \rightarrow t \text{ is in } E_1.type\;\}$$

Describe an SDD that determines a unique type for each subexpression by using an attribute *type* to synthesize a set of possible types bottom-up, and, once the unique type of the overall expression is determined, proceeds top-down to determine attribute *unique* for the type of each subexpression.

6.6 Control Flow

The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions. In programming languages, boolean expressions are often used to

1. *Alter the flow of control.* Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in **if** (E) S, the expression E must be true if statement S is reached.

2. *Compute logical values.* A boolean expression can represent *true* or *false* as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

The intended use of boolean expressions is determined by its syntactic context. For example, an expression following the keyword **if** is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value. Such syntactic contexts can be specified in a number of ways: we may use two different nonterminals, use inherited attributes, or set a flag during parsing. Alternatively we may build a syntax tree and invoke different procedures for the two different uses of boolean expressions.

This section concentrates on the use of boolean expressions to alter the flow of control. For clarity, we introduce a new nonterminal B for this purpose. In Section 6.6.6, we consider how a compiler can allow boolean expressions to represent logical values.

6.6.1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote &&, ||, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **rel** E_2, where E_1 and

https://hemanthrajhemu.github.io

E_2 are arithmetic expressions. In this section, we consider boolean expressions generated by the following grammar:

$$B \rightarrow B \texttt{ || } B \mid B \texttt{ \&\& } B \mid \texttt{ ! } B \mid (B) \mid E \textbf{ rel } E \mid \textbf{true} \mid \textbf{false}$$

We use the attribute **rel**.*op* to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by **rel**. As is customary, we assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.

Given the expression B_1 || B_2, if we determine that B_1 is true, then we can conclude that the entire expression is true without having to evaluate B_2. Similarly, given B_1&&B_2, if B_1 is false, then the entire expression is false.

The semantic definition of the programming language determines whether all parts of a boolean expression must be evaluated. If the language definition permits (or requires) portions of a boolean expression to go unevaluated, then the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value. Thus, in an expression such as B_1 || B_2, neither B_1 nor B_2 is necessarily evaluated fully. If either B_1 or B_2 is an expression with side effects (e.g., it contains a function that changes a global variable), then an unexpected answer may be obtained.

6.6.2 Short-Circuit Code

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 6.21: The statement

```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

might be translated into the code of Fig. 6.34. In this translation, the boolean expression is true if control reaches label L_2. If the expression is false, control goes immediately to L_1, skipping L_2 and the assignment `x = 0`. □

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.34: Jumping code

https://hemanthrajhemu.github.io

6.6.3 Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:

$$\begin{array}{lcl} S & \rightarrow & \textbf{if} \ (\ B\ )\ S_1 \\ S & \rightarrow & \textbf{if} \ (\ B\ )\ S_1 \ \textbf{else} \ S_2 \\ S & \rightarrow & \textbf{while} \ (\ B\ )\ S_1 \end{array}$$

In these productions, nonterminal B represents a boolean expression and nonterminal S represents a statement.

This grammar generalizes the running example of while expressions that we introduced in Example 5.19. As in that example, both B and S have a synthesized attribute *code*, which gives the translation into three-address instructions. For simplicity, we build up the translations $B.code$ and $S.code$ as strings, using syntax-directed definitions. The semantic rules defining the *code* attributes could be implemented instead by building up syntax trees and then emitting code during a tree traversal, or by any of the approaches outlined in Section 5.5.

The translation of **if** (B) S_1 consists of $B.code$ followed by $S_1.code$, as illustrated in Fig. 6.35(a). Within $B.code$ are jumps based on the value of B. If B is true, control flows to the first instruction of $S_1.code$, and if B is false, control flows to the instruction immediately following $S_1.code$.

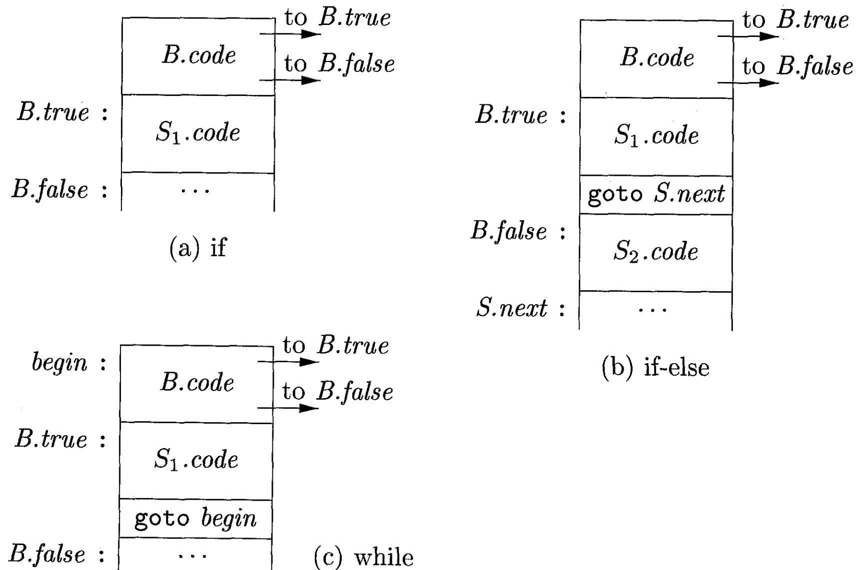


Figure 6.35: Code for if-, if-else-, and while-statements

The labels for the jumps in $B.code$ and $S.code$ are managed using inherited attributes. With a boolean expression B, we associate two labels: $B.true$, the

https://hemanthrajhemu.github.io

label to which control flows if B is true, and $B.false$, the label to which control flows if B is false. With a statement S, we associate an inherited attribute $S.next$ denoting a label for the instruction immediately after the code for S. In some cases, the instruction immediately following $S.code$ is a jump to some label L. A jump to a jump to L from within $S.code$ is avoided using $S.next$.

The syntax-directed definition in Fig. 6.36-6.37 produces three-address code for boolean expressions in the context of if-, if-else-, and while-statements.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}'\ S.next)$ $\|\| \; label(B.false) \;\|\|\; S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

Figure 6.36: Syntax-directed definition for flow-of-control statements.

We assume that $newlabel()$ creates a new label each time it is called, and that $label(L)$ attaches label L to the next three-address instruction to be generated.[8]

[8] If implemented literally, the semantic rules will generate lots of labels and may attach more than one label to a three-address instruction. The backpatching approach of Section 6.7

https://hemanthrajhemu.github.io

A program consists of a statement generated by $P \rightarrow S$. The semantic rules associated with this production initialize *S.next* to a new label. *P.code* consists of *S.code* followed by the new label *S.next*. Token **assign** in the production $S \rightarrow$ **assign** is a placeholder for assignment statements. The translation of assignments is as discussed in Section 6.4; for this discussion of control flow, *S.code* is simply **assign**.*code*.

In translating $S \rightarrow$ **if** (B) S_1, the semantic rules in Fig. 6.36 create a new label *B.true* and attach it to the first three-address instruction generated for the statement S_1, as illustrated in Fig. 6.35(a). Thus, jumps to *B.true* within the code for B will go to the code for S_1. Further, by setting *B.false* to *S.next*, we ensure that control will skip the code for S_1 if B evaluates to false.

In translating the if-else-statement $S \rightarrow$ **if** (B) S_1 **else** S_2, the code for the boolean expression B has jumps out of it to the first instruction of the code for S_1 if B is true, and to the first instruction of the code for S_2 if B is false, as illustrated in Fig. 6.35(b). Further, control flows from both S_1 and S_2 to the three-address instruction immediately following the code for S — its label is given by the inherited attribute *S.next*. An explicit `goto` *S.next* appears after the code for S_1 to skip over the code for S_2. No goto is needed after S_2, since S_2*.next* is the same as *S.next*.

The code for $S \rightarrow$ **while** (B) S_1 is formed from *B.code* and S_1*.code* as shown in Fig. 6.35(c). We use a local variable *begin* to hold a new label attached to the first instruction for this while-statement, which is also the first instruction for B. We use a variable rather than an attribute, because *begin* is local to the semantic rules for this production. The inherited label *S.next* marks the instruction that control must flow to if B is false; hence, *B.false* is set to be *S.next*. A new label *B.true* is attached to the first instruction for S_1; the code for B generates a jump to this label if B is true. After the code for S_1 we place the instruction `goto` *begin*, which causes a jump back to the beginning of the code for the boolean expression. Note that S_1*.next* is set to this label *begin*, so jumps from within S_1*.code* can go directly to *begin*.

The code for $S \rightarrow S_1$ S_2 consists of the code for S_1 followed by the code for S_2. The semantic rules manage the labels; the first instruction after the code for S_1 is the beginning of the code for S_2; and the instruction after the code for S_2 is also the instruction after the code for S.

We discuss the translation of flow-of-control statements further in Section 6.7. There we shall see an alternative method, called "backpatching," which emits code for statements in one pass.

6.6.4 Control-Flow Translation of Boolean Expressions

The semantic rules for boolean expressions in Fig. 6.37 complement the semantic rules for statements in Fig. 6.36. As in the code layout of Fig. 6.35, a boolean expression B is translated into three-address instructions that evaluate B using

---

creates labels only when they are needed. Alternatively, unnecessary labels can be eliminated during a subsequent optimization phase.

https://hemanthrajhemu.github.io

conditional and unconditional jumps to one of two labels: $B.true$ if B is true, and $B.false$ if B is false.

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \;\texttt{||}\; B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.false) \;\|\|\; B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$
$B \rightarrow \;!\; B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \;\textbf{rel}\; E_2$	$B.code = E_1.code \;\|\|\; E_2.code$ $\|\| \; gen(\texttt{'if'}\; E_1.addr\; \textbf{rel}.op\; E_2.addr\; \texttt{'goto'}\; B.true)$ $\|\| \; gen(\texttt{'goto'}\; B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen(\texttt{'goto'}\; B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen(\texttt{'goto'}\; B.false)$

Figure 6.37: Generating three-address code for booleans

The fourth production in Fig. 6.37, $B \rightarrow E_1$ **rel** E_2, is translated directly into a comparison three-address instruction with jumps to the appropriate places. For instance, B of the form $a < b$ translates into:

```
if a < b goto B.true
goto B.false
```

The remaining productions for B are translated as follows:

1. Suppose B is of the form B_1 || B_2. If B_1 is true, then we immediately know that B itself is true, so $B_1.true$ is the same as $B.true$. If B_1 is false, then B_2 must be evaluated, so we make $B_1.false$ be the label of the first instruction in the code for B_2. The true and false exits of B_2 are the same as the true and false exits of B, respectively.

https://hemanthrajhemu.github.io

2. The translation of B_1 && B_2 is similar.

3. No code is needed for an expression B of the form $!B_1$: just interchange the true and false exits of B to get the true and false exits of B_1.

4. The constants **true** and **false** translate into jumps to $B.true$ and $B.false$, respectively.

Example 6.22: Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;
```
(6.13)

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Figure 6.38: Control-flow translation of a simple if-statement

The statement (6.13) constitutes a program generated by $P \rightarrow S$ from Fig. 6.36. The semantic rules for the production generate a new label L_1 for the instruction after the code for S. Statement S has the form **if** (B) S_1, where S_1 is `x = 0;`, so the rules in Fig. 6.36 generate a new label L_2 and attach it to the first (and only, in this case) instruction in $S_1.code$, which is `x = 0`.

Since `||` has lower precedence than &&, the boolean expression in (6.13) has the form B_1 `||` B_2, where B_1 is $x < 100$. Following the rules in Fig. 6.37, $B_1.true$ is L_2, the label of the assignment `x = 0;`. $B_1.false$ is a new label L_3, attached to the first instruction in the code for B_2.

Note that the code generated is not optimal, in that the translation has three more instructions (goto's) than the code in Example 6.21. The instruction `goto` L_3 is redundant, since L_3 is the label of the very next instruction. The two `goto` L_1 instructions can be eliminated by using `ifFalse` instead of `if` instructions, as in Example 6.21. □

6.6.5 Avoiding Redundant Gotos

In Example 6.22, the comparison $x > 200$ translates into the code fragment:

https://hemanthrajhemu.github.io

```
        if x > 200 goto L4
        goto L1
L4:     ...
```

Instead, consider the instruction:

```
        ifFalse x > 200 goto L1
L4:     ...
```

This `ifFalse` instruction takes advantage of the natural flow from one instruction to the next in sequence, so control simply "falls through" to label L_4 if $x > 200$ is false, thereby avoiding a jump.

In the code layouts for if- and while-statements in Fig. 6.35, the code for statement S_1 immediately follows the code for the boolean expression B. By using a special label *fall* (i.e., "don't generate any jump"), we can adapt the semantic rules in Fig. 6.36 and 6.37 to allow control to fall through from the code for B to the code for S_1. The new rules for $S \rightarrow$ **if** (B) S_1 in Fig. 6.36 set $B.true$ to *fall*:

$$\begin{aligned} B.true &= fall \\ B.false &= S_1.next \ = \ S.next \\ S.code &= B.code \ || \ S_1.code \end{aligned}$$

Similarly, the rules for if-else- and while-statements also set $B.true$ to *fall*.

We now adapt the semantic rules for boolean expressions to allow control to fall through whenever possible. The new rules for $B \rightarrow E_1$ **rel** E_2 in Fig. 6.39 generate two instructions, as in Fig. 6.37, if both $B.true$ and $B.false$ are explicit labels; that is, neither equals *fall*. Otherwise, if $B.true$ is an explicit label, then $B.false$ must be *fall*, so they generate an `if` instruction that lets control fall through if the condition is false. Conversely, if $B.false$ is an explicit label, then they generate an `ifFalse` instruction. In the remaining case, both $B.true$ and $B.false$ are *fall*, so no jump in generated.[9]

In the new rules for $B \rightarrow B_1$ `||` B_2 in Fig. 6.40, note that the meaning of label *fall* for B is different from its meaning for B_1. Suppose $B.true$ is *fall*; i.e, control falls through B, if B evaluates to true. Although B evaluates to true if B_1 does, $B_1.true$ must ensure that control jumps over the code for B_2 to get to the next instruction after B.

On the other hand, if B_1 evaluates to false, the truth-value of B is determined by the value of B_2, so the rules in Fig. 6.40 ensure that $B_1.false$ corresponds to control falling through from B_1 to the code for B_2.

The semantic rules are for $B \rightarrow B_1$ && B_2 are similar to those in Fig. 6.40. We leave them as an exercise.

Example 6.23: With the new rules using the special label *fall*, the program (6.13) from Example 6.21

[9] In C and Java, expressions may contain assignments within them, so code must be generated for the subexpressions E_1 and E_2, even if both $B.true$ and $B.false$ are *fall*. If desired, dead code can be eliminated during an optimization phase.

https://hemanthrajhemu.github.io

$$
\begin{aligned}
test &= E_1.addr \ \mathbf{rel}.op \ E_2.addr \\
s &= \mathbf{if}\ B.true \neq fall \ \mathbf{and}\ B.false \neq fall \ \mathbf{then} \\
&\qquad gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \ || \ gen(\texttt{'goto'}\ B.false) \\
&\quad \mathbf{else\ if}\ B.true \neq fall \ \mathbf{then}\ gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \\
&\quad \mathbf{else\ if}\ B.false \neq fall \ \mathbf{then}\ gen(\texttt{'ifFalse'}\ test\ \texttt{'goto'}\ B.false) \\
&\quad \mathbf{else}\ '\,' \\
B.code &= E_1.code \ || \ E_2.code \ || \ s
\end{aligned}
$$

Figure 6.39: Semantic rules for $B \rightarrow E_1$ **rel** E_2

$$
\begin{aligned}
B_1.true &= \mathbf{if}\ B.true \neq fall \ \mathbf{then}\ B.true \ \mathbf{else}\ newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \mathbf{if}\ B.true \neq fall \ \mathbf{then}\ B_1.code \ || \ B_2.code \\
&\quad \mathbf{else}\ B_1.code \ || \ B_2.code \ || \ label(B_1.true)
\end{aligned}
$$

Figure 6.40: Semantic rules for $B \rightarrow B_1$ `||` B_2

```
if( x < 100 || x > 200 && x != y ) x = 0;
```

translates into the code of Fig. 6.41.

```
        if x < 100 goto L2
        ifFalse x > 200 goto L1
        ifFalse x != y goto L1
L2:     x = 0
L1:
```

Figure 6.41: If-statement translated using the fall-through technique

As in Example 6.22, the rules for $P \rightarrow S$ create label L_1. The difference from Example 6.22 is that the inherited attribute $B.true$ is *fall* when the semantic rules for $B \rightarrow B_1$ `||` B_2 are applied ($B.false$ is L_1). The rules in Fig. 6.40 create a new label L_2 to allow a jump over the code for B_2 if B_1 evaluates to true. Thus, $B_1.true$ is L_2 and $B_1.false$ is *fall*, since B_2 must be evaluated if B_1 is false.

The production $B \rightarrow E_1$ **rel** E_2 that generates $x < 100$ is therefore reached with $B.true = L_2$ and $B.false = fall$. With these inherited labels, the rules in Fig. 6.39 therefore generate a single instruction `if x < 100 goto L2`. □

https://hemanthrajhemu.github.io

6.6.6 Boolean Values and Jumping Code

The focus in this section has been on the use of boolean expressions to alter the flow of control in statements. A boolean expression may also be evaluated for its value, as in assignment statements such as `x = true;` or `x = a<b;`.

A clean way of handling both roles of boolean expressions is to first build a syntax tree for expressions, using either of the following approaches:

1. *Use two passes.* Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.

2. *Use one pass for statements, but two passes for expressions.* With this approach, we would translate E in **while** (E) S_1 before S_1 is examined. The translation of E, however, would be done by building its syntax tree and then walking the tree.

The following grammar has a single nonterminal E for expressions:

$$S \rightarrow \textbf{id} = E \; ; \mid \textbf{if} \; (\; E \;) \; S \mid \textbf{while} \; (\; E \;) \; S \mid S \; S$$
$$E \rightarrow E \; || \; E \mid E \; \&\& \; E \mid E \; \textbf{rel} \; E \mid E + E \mid (E) \mid \textbf{id} \mid \textbf{true} \mid \textbf{false}$$

Nonterminal E governs the flow of control in $S \rightarrow$ **while** (E) S_1. The same nonterminal E denotes a value in $S \rightarrow$ **id** $= E$; and $E \rightarrow E + E$.

We can handle these two roles of expressions by using separate code-generation functions. Suppose that attribute $E.n$ denotes the syntax-tree node for an expression E and that nodes are objects. Let method *jump* generate jumping code at an expression node, and let method *rvalue* generate code to compute the value of the node into a temporary.

When E appears in $S \rightarrow$ **while** (E) S_1, method *jump* is called at node $E.n$. The implementation of *jump* is based on the rules for boolean expressions in Fig. 6.37. Specifically, jumping code is generated by calling $E.n.jump(t, f)$, where t is a new label for the first instruction of $S_1.code$ and f is the label $S.next$.

When E appears in $S \rightarrow$ **id** $= E$;, method *rvalue* is called at node $E.n$. If E has the form $E_1 + E_2$, the method call $E.n.rvalue()$ generates code as discussed in Section 6.4. If E has the form $E_1 \; \&\& \; E_2$, we first generate jumping code for E and then assign true or false to a new temporary `t` at the true and false exits, respectively, from the jumping code.

For example, the assignment `x = a<b && c<d` can be implemented by the code in Fig. 6.42.

6.6.7 Exercises for Section 6.6

Exercise 6.6.1: Add rules to the syntax-directed definition of Fig. 6.36 for the following control-flow constructs:

a) A repeat-statment **repeat** S **while** B.

https://hemanthrajhemu.github.io

```
        ifFalse a < b goto L1
        ifFalse c > d goto L1
        t = true
        goto L2
L1:     t = false
L2:     x = t
```

Figure 6.42: Translating a boolean assignment by computing the value of a temporary

! b) A for-loop **for** (S_1; B; S_2) S_3.

Exercise 6.6.2: Modern machines try to execute many instructions at the same time, including branching instructions. Thus, there is a severe cost if the machine speculatively follows one branch, when control actually goes another way (all the speculative work is thrown away). It is therefore desirable to minimize the number of branches. Notice that the implementation of a while-loop in Fig. 6.35(c) has two branches per interation: one to enter the body from the condition B and the other to jump back to the code for B. As a result, it is usually preferable to implement **while** (B) S as if it were **if** (B) { **repeat** S **until** !(B) }. Show what the code layout looks like for this translation, and revise the rule for while-loops in Fig. 6.36.

! Exercise 6.6.3: Suppose that there were an "exclusive-or" operator (true if and only if exactly one of its two arguments is true) in C. Write the rule for this operator in the style of Fig. 6.37.

Exercise 6.6.4: Translate the following expressions using the goto-avoiding translation scheme of Section 6.6.5:

a) `if (a==b && c==d || e==f) x == 1;`

b) `if (a==b || c==d || e==f) x == 1;`

c) `if (a==b && c==d && e==f) x == 1;`

Exercise 6.6.5: Give a translation scheme based on the syntax-directed definition in Figs. 6.36 and 6.37.

Exercise 6.6.6: Adapt the semantic rules in Figs. 6.36 and 6.37 to allow control to fall through, using rules like the ones in Figs. 6.39 and 6.40.

! Exercise 6.6.7: The semantic rules for statements in Exercise 6.6.6 generate unnecessary labels. Modify the rules for statements in Fig. 6.36 to create labels as needed, using a special label *deferred* to mean that a label has not yet been created. Your rules must generate code similar to that in Example 6.21.

https://hemanthrajhemu.github.io

!! **Exercise 6.6.8:** Section 6.6.5 talks about using fall-through code to minimize the number of jumps in the generated intermediate code. However, it does not take advantage of the option to replace a condition by its complement, e.g., replace `if a < b goto` L_1`; goto` L_2 by `if b >= a goto` L_2`; goto` L_1. Develop a SDD that does take advantage of this option when needed.

6.7 Backpatching

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression B in `if (` B `)` S contains a jump, for when B is false, to the instruction following the code for S. In a one-pass translation, B must be translated before S is examined. What then is the target of the `goto` that jumps over the code for S? In Section 6.6 we addressed this problem by passing labels as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses.

This section takes a complementary approach, called *backpatching*, in which lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

6.7.1 One-Pass Code Generation Using Backpatching

Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass. The translations we generate will be of the same form as those in Section 6.6, except for how we manage labels.

In this section, synthesized attributes *truelist* and *falselist* of nonterminal B are used to manage labels in jumping code for boolean expressions. In particular, $B.truelist$ will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true. $B.falselist$ likewise is the list of instructions that eventually get the label to which control goes when B is false. As code is generated for B, jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by $B.truelist$ and $B.falselist$, as appropriate. Similarly, a statement S has a synthesized attribute $S.nextlist$, denoting a list of jumps to the instruction immediately following the code for S.

For specificity, we generate instructions into an instruction array, and labels will be indices into this array. To manipulate lists of jumps, we use three functions:

1. *makelist*(i) creates a new list containing only i, an index into the array of instructions; *makelist* returns a pointer to the newly created list.

https://hemanthrajhemu.github.io

2. $merge(p_1, p_2)$ concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.

3. $backpatch(p, i)$ inserts i as the target label for each of the instructions on the list pointed to by p.

6.7.2 Backpatching for Boolean Expressions

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker nonterminal M in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

$$B \to B_1 \;||\; M\, B_2 \mid B_1 \;\&\&\; M\; B_2 \mid\; !\; B_1 \mid (\, B_1\,) \mid E_1 \;\textbf{rel}\; E_2 \mid \textbf{true} \mid \textbf{false}$$
$$M \to \epsilon$$

The translation scheme is in Fig. 6.43.

1)	$B \to B_1 \;||\; M\; B_2$	{ $backpatch(B_1.falselist, M.instr)$; $B.truelist = merge(B_1.truelist, B_2.truelist)$; $B.falselist = B_2.falselist$; }
2)	$B \to B_1 \;\&\&\; M\; B_2$	{ $backpatch(B_1.truelist, M.instr)$; $B.truelist = B_2.truelist$; $B.falselist = merge(B_1.falselist, B_2.falselist)$; }
3)	$B \to\; !\; B_1$	{ $B.truelist = B_1.falselist$; $B.falselist = B_1.truelist$; }
4)	$B \to (\; B_1\;)$	{ $B.truelist = B_1.truelist$; $B.falselist = B_1.falselist$; }
5)	$B \to E_1$ **rel** E_2	{ $B.truelist = makelist(nextinstr)$; $B.falselist = makelist(nextinstr + 1)$; $emit($'`if`' $E_1.addr$ **rel**$.op$ $E_2.addr$ '`goto _`'$)$; $emit($'`goto _`'$)$; }
6)	$B \to$ **true**	{ $B.truelist = makelist(nextinstr)$; $emit($'`goto _`'$)$; }
7)	$B \to$ **false**	{ $B.falselist = makelist(nextinstr)$; $emit($'`goto _`'$)$; }
8)	$M \to \epsilon$	{ $M.instr = nextinstr$; }

Figure 6.43: Translation scheme for boolean expressions

Consider semantic action (1) for the production $B \to B_1 \;||\; M\, B_2$. If B_1 is true, then B is also true, so the jumps on $B_1.truelist$ become part of $B.truelist$. If B_1 is false, however, we must next test B_2, so the target for the jumps

https://hemanthrajhemu.github.io

B_1.*falselist* must be the beginning of the code generated for B_2. This target is obtained using the marker nonterminal M. That nonterminal produces, as a synthesized attribute $M.instr$, the index of the next instruction, just before B_2 code starts being generated.

To obtain that instruction index, we associate with the production $M \rightarrow \epsilon$ the semantic action

$$\{ M.instr = nextinstr; \}$$

The variable *nextinstr* holds the index of the next instruction to follow. This value will be backpatched onto the B_1.*falselist* (i.e., each instruction on the list B_1.*falselist* will receive $M.instr$ as its target label) when we have seen the remainder of the production $B \rightarrow B_1 \; || \; M \; B_2$.

Semantic action (2) for $B \rightarrow B_1 \; \&\& \; M \; B_2$ is similar to (1). Action (3) for $B \rightarrow \; !B$ swaps the true and false lists. Action (4) ignores parentheses.

For simplicity, semantic action (5) generates two instructions, a conditional goto and an unconditional one. Neither has its target filled in. These instructions are put on new lists, pointed to by B.*truelist* and B.*falselist*, respectively.

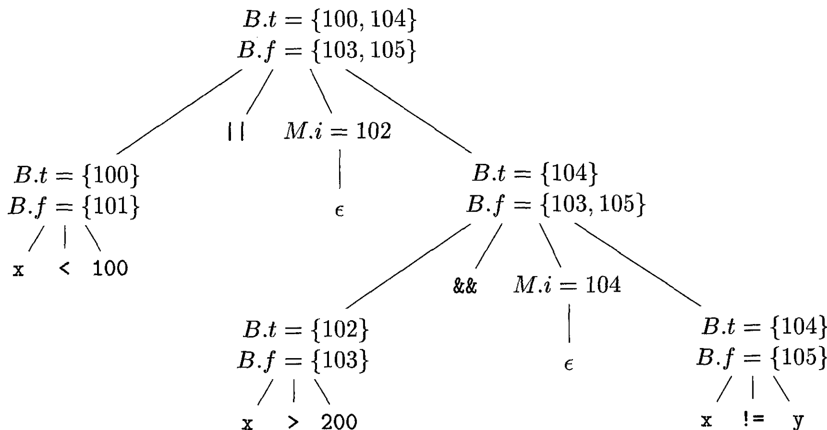


Figure 6.44: Annotated parse tree for $x < 100 \; || \; x > 200 \; \&\& \; x \; != \; y$

Example 6.24: Consider again the expression

$$x < 100 \; || \; x > 200 \; \&\& \; x \; != \; y$$

An annotated parse tree is shown in Fig. 6.44; for readability, attributes *truelist*, *falselist*, and *instr* are represented by their initial letters. The actions are performed during a depth-first traversal of the tree. Since all actions appear at the ends of right sides, they can be performed in conjunction with reductions during a bottom-up parse. In response to the reduction of $x < 100$ to B by production (5), the two instructions

https://hemanthrajhemu.github.io

```
100:   if x < 100 goto _
101:   goto _
```

are generated. (We arbitrarily start instruction numbers at 100.) The marker nonterminal M in the production

$$B \to B_1 \,||\, M\, B_2$$

records the value of *nextinstr*, which at this time is 102. The reduction of $x > 200$ to B by production (5) generates the instructions

```
102:   if x > 200 goto _
103:   goto _
```

The subexpression $x > 200$ corresponds to B_1 in the production

$$B \to B_1 \;\&\&\; M\; B_2$$

The marker nonterminal M records the current value of *nextinstr*, which is now 104. Reducing $x \mathrel{!=} y$ into B by production (5) generates

```
104:   if x != y goto _
105:   goto _
```

We now reduce by $B \to B_1 \;\&\&\; M\; B_2$. The corresponding semantic action calls *backpatch*(B_1.*truelist*, M.*instr*) to bind the true exit of B_1 to the first instruction of B_2. Since B_1.*truelist* is $\{102\}$ and M.*instr* is 104, this call to *backpatch* fills in 104 in instruction 102. The six instructions generated so far are thus as shown in Fig. 6.45(a).

The semantic action associated with the final reduction by $B \to B_1 \,||\, M\, B_2$ calls *backpatch*($\{101\}$,102) which leaves the instructions as in Fig. 6.45(b).

The entire expression is true if and only if the gotos of instructions 100 or 104 are reached, and is false if and only if the gotos of instructions 103 or 105 are reached. These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression. □

6.7.3 Flow-of-Control Statements

We now use backpatching to translate flow-of-control statements in one pass. Consider statements generated by the following grammar:

$$\begin{array}{lcl} S & \to & \textbf{if}\,(\,B\,)\,S \;\mid\; \textbf{if}\,(\,B\,)\,S\;\textbf{else}\;S \;\mid\; \textbf{while}\,(\,B\,)\,S \;\mid\; \{\,L\,\} \mid A\;; \\ L & \to & L\,S \;\mid\; S \end{array}$$

Here S denotes a statement, L a statement list, A an assignment-statement, and B a boolean expression. Note that there must be other productions, such as

https://hemanthrajhemu.github.io

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(a) After backpatching 104 into instruction 102.

```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(b) After backpatching 102 into instruction 101.

Figure 6.45: Steps in the backpatch process

those for assignment-statements. The productions given, however, are sufficient to illustrate the techniques used to translate flow-of-control statements.

The code layout for if-, if-else-, and while-statements is the same as in Section 6.6. We make the tacit assumption that the code sequence in the instruction array reflects the natural flow of control from one instruction to the next. If not, then explicit jumps must be inserted to implement the natural sequential flow of control.

The translation scheme in Fig. 6.46 maintains lists of jumps that are filled in when their targets are found. As in Fig. 6.43, boolean expressions generated by nonterminal B have two lists of jumps, $B.truelist$ and $B.falselist$, corresponding to the true and false exits from the code for B, respectively. Statements generated by nonterminals S and L have a list of unfilled jumps, given by attribute *nextlist*, that must eventually be completed by backpatching. $S.nextlist$ is a list of all conditional and unconditional jumps to the instruction following the code for statement S in execution order. $L.nextlist$ is defined similarly.

Consider the semantic action (3) in Fig. 6.46. The code layout for production $S \rightarrow \textbf{while} (B) S_1$ is as in Fig. 6.35(c). The two occurrences of the marker nonterminal M in the production

$$S \rightarrow \textbf{while } M_1 \text{ (} B \text{) } M_2 \ S_1$$

record the instruction numbers of the beginning of the code for B and the beginning of the code for S_1. The corresponding labels in Fig. 6.35(c) are *begin* and $B.true$, respectively.

https://hemanthrajhemu.github.io

1) $S \rightarrow \textbf{if} \, (\, B \,) \, M \, S_1$ $\{$ $backpatch(B.truelist, \; M.instr);$
$S.nextlist \; = \; merge(B.falselist, \; S_1.nextlist); \}$

2) $S \rightarrow \; \textbf{if} \, (\, B \,) \, M_1 \, S_1 \, N \, \textbf{else} \, M_2 \, S_2$
$\{ \; backpatch(B.truelist, \; M_1.instr);$
$backpatch(B.falselist, \; M_2.instr);$
$temp \; = \; merge(S_1.nextlist, \; N.nextlist);$
$S.nextlist \; = \; merge(temp, \; S_2.nextlist); \}$

3) $S \rightarrow \; \textbf{while} \; M_1 \; (\, B \,) \; M_2 \, S_1$
$\{ \; backpatch(S_1.nextlist, \; M_1.instr);$
$backpatch(B.truelist, \; M_2.instr);$
$S.nextlist \; = \; B.falselist;$
$emit(\text{'goto'} \; M_1.instr); \}$

4) $S \rightarrow \{ \; L \; \}$ $\{ \; S.nextlist \; = \; L.nextlist; \}$

5) $S \rightarrow A \; ;$ $\{ \; S.nextlist \; = \; \textbf{null}; \}$

6) $M \rightarrow \epsilon$ $\{ \; M.instr \; = \; nextinstr; \}$

7) $N \rightarrow \epsilon$ $\{ \; N.nextlist \; = \; makelist(nextinstr);$
$emit(\text{'goto _'}); \}$

8) $L \rightarrow L_1 \; M \; S$ $\{ \; backpatch(L_1.nextlist, \; M.instr);$
$L.nextlist \; = \; S.nextlist; \}$

9) $L \rightarrow S$ $\{ \; L.nextlist \; = \; S.nextlist; \}$

Figure 6.46: Translation of statements

Again, the only production for M is $M \rightarrow \epsilon$. Action (6) in Fig. 6.46 sets attribute $M.instr$ to the number of the next instruction. After the body S_1 of the while-statement is executed, control flows to the beginning. Therefore, when we reduce **while** M_1 (B) $M_2 \, S_1$ to S, we backpatch $S_1.nextlist$ to make all targets on that list be $M_1.instr$. An explicit jump to the beginning of the code for B is appended after the code for S_1 because control may also "fall out the bottom." $B.truelist$ is backpatched to go to the beginning of S_1 by making jumps on $B.truelist$ go to $M_2.instr$.

A more compelling argument for using $S.nextlist$ and $L.nextlist$ comes when code is generated for the conditional statement **if** (B) S_1 **else** S_2. If control "falls out the bottom" of S_1, as when S_1 is an assignment, we must include at the end of the code for S_1 a jump over the code for S_2. We use another marker nonterminal to generate this jump after S_1. Let nonterminal N be this

https://hemanthrajhemu.github.io

marker with production $N \rightarrow \epsilon$. N has attribute $N.nextlist$, which will be a list consisting of the instruction number of the jump goto _ that is generated by the semantic action (7) for N.

Semantic action (2) in Fig. 6.46 deals with if-else-statements with the syntax

$$S \rightarrow \textbf{if} \; (\; B \;) \; M_1 \; S_1 \; N \; \textbf{else} \; M_2 \; S_2$$

We backpatch the jumps when B is true to the instruction $M_1.instr$; the latter is the beginning of the code for S_1. Similarly, we backpatch jumps when B is false to go to the beginning of the code for S_2. The list $S.nextlist$ includes all jumps out of S_1 and S_2, as well as the jump generated by N. (Variable *temp* is a temporary that is used only for merging lists.)

Semantic actions (8) and (9) handle sequences of statements. In

$$L \rightarrow L_1 \, M \, S$$

the instruction following the code for L_1 in order of execution is the beginning of S. Thus the $L_1.nextlist$ list is backpatched to the beginning of the code for S, which is given by $M.instr$. In $L \rightarrow S$, $L.nextlist$ is the same as $S.nextlist$.

Note that no new instructions are generated anywhere in these semantic rules, except for rules (3) and (7). All other code is generated by the semantic actions associated with assignment-statements and expressions. The flow of control causes the proper backpatching so that the assignments and boolean expression evaluations will connect properly.

6.7.4 Break-, Continue-, and Goto-Statements

The most elementary programming language construct for changing the flow of control in a program is the goto-statement. In C, a statement like `goto L` sends control to the statement labeled `L` — there must be precisely one statement with label `L` in this scope. Goto-statements can be implemented by maintaining a list of unfilled jumps for each label and then backpatching the target when it is known.

Java does away with goto-statements. However, Java does permit disciplined jumps called break-statements, which send control out of an enclosing construct, and continue-statements, which trigger the next iteration of an enclosing loop. The following excerpt from a lexical analyzer illustrates simple break- and continue-statements:

```
1)  for ( ; ; readch() ) {
2)      if( peek == ' ' || peek == '\t' ) continue;
3)      else if( peek == '\n' ) line = line + 1;
4)      else break;
5)  }
```

Control jumps from the break-statement on line 4 to the next statement after the enclosing for loop. Control jumps from the continue-statement on line 2 to code to evaluate *readch*() and then to the if-statement on line 2.

https://hemanthrajhemu.github.io

If S is the enclosing construct, then a break-statement is a jump to the first instruction after the code for S. We can generate code for the break by (1) keeping track of the enclosing statement S, (2) generating an unfilled jump for the break-statement, and (3) putting this unfilled jump on $S.nextlist$, where *nextlist* is as discussed in Section 6.7.3.

In a two-pass front end that builds syntax trees, $S.nextlist$ can be implemented as a field in the node for S. We can keep track of S by using the symbol table to map a special identifier **break** to the node for the enclosing statement S. This approach will also handle labeled break-statements in Java, since the symbol table can be used to map the label to the syntax-tree node for the enclosing construct.

Alternatively, instead of using the symbol table to access the node for S, we can put a pointer to $S.nextlist$ in the symbol table. Now, when a break-statement is reached, we generate an unfilled jump, look up *nextlist* through the symbol table, and add the jump to the list, where it will be backpatched as discussed in Section 6.7.3.

Continue-statements can be handled in a manner analogous to the break-statement. The main difference between the two is that the target of the generated jump is different.

6.7.5 Exercises for Section 6.7

Exercise 6.7.1: Using the translation of Fig. 6.43, translate each of the following expressions. Show the true and false lists for each subexpression. You may assume the address of the first instruction generated is 100.

a) `a==b && (c==d || e==f)`

b) `(a==b || c==d) || e==f`

c) `(a==b && c==d) && e==f`

Exercise 6.7.2: In Fig. 6.47(a) is the outline of a program, and Fig. 6.47(b) sketches the structure of the generated three-address code, using the backpatching translation of Fig. 6.46. Here, i_1 through i_8 are the labels of the generated instructions that begin each of the "Code" sections. When we implement this translation, we maintain, for each boolean expression E, two lists of places in the code for E, which we denote by $E.true$ and $E.false$. The places on list $E.true$ are those places where we eventually put the label of the statement to which control must flow whenever E is true; $E.false$ similarly lists the places where we put the label that control flows to when E is found to be false. Also, we maintain for each statement S, a list of places where we must put the label to which control flows when S is finished. Give the value (one of i_1 through i_8) that eventually replaces each place on each of the following lists:

(a) $E_3.false$ (b) $S_2.next$ (c) $E_4.false$ (d) $S_1.next$ (e) $E_2.true$

https://hemanthrajhemu.github.io

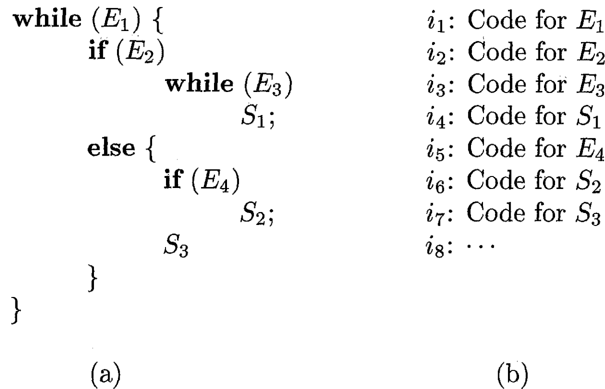


Figure 6.47: Control-flow structure of program for Exercise 6.7.2

Exercise 6.7.3: When performing the translation of Fig. 6.47 using the scheme of Fig. 6.46, we create lists *S.next* for each statement, starting with the assignment-statements S_1, S_2, and S_3, and proceeding to progressively larger if-statements, if-else-statements, while-statements, and statement blocks. There are five constructed statements of this type in Fig. 6.47:

S_4: **while** (E_3) S_1.

S_5: **if** (E_4) S_2.

S_6: The block consisting of S_5 and S_3.

S_7: The statement **if** S_4 **else** S_6.

S_8: The entire program.

For each of these constructed statements, there is a rule that allows us to construct $S_i.next$ in terms of other $S_j.next$ lists, and the lists $E_k.true$ and $E_k.false$ for the expressions in the program. Give the rules for

(a) $S_4.next$ (b) $S_5.next$ (c) $S_6.next$ (d) $S_7.next$ (e) $S_8.next$

6.8 Switch-Statements

The "switch" or "case" statement is available in a variety of languages. Our switch-statement syntax is shown in Fig. 6.48. There is a selector expression E, which is to be evaluated, followed by n constant values $V_1, V_2, \cdots, V_n$ that the expression might take, perhaps including a *default* "value," which always matches the expression if no other value does.

https://hemanthrajhemu.github.io

```
switch ( E ) {
        case V1: S1
        case V2: S2
                ...
        case Vn-1: Sn-1
        default: Sn
}
```

Figure 6.48: Switch-statement syntax

6.8.1 Translation of Switch-Statements

The intended translation of a switch is code to:

1. Evaluate the expression E.

2. Find the value V_j in the list of cases that is the same as the value of the expression. Recall that the default value matches the expression if none of the values explicitly mentioned in cases does.

3. Execute the statement S_j associated with the value found.

Step (2) is an n-way branch, which can be implemented in one of several ways. If the number of cases is small, say 10 at most, then it is reasonable to use a sequence of conditional jumps, each of which tests for an individual value and transfers to the code for the corresponding statement.

A compact way to implement this sequence of conditional jumps is to create a table of pairs, each pair consisting of a value and a label for the corresponding statement's code. The value of the expression itself, paired with the label for the default statement is placed at the end of the table at run time. A simple loop generated by the compiler compares the value of the expression with each value in the table, being assured that if no other match is found, the last (default) entry is sure to match.

If the number of values exceeds 10 or so, it is more efficient to construct a hash table for the values, with the labels of the various statements as entries. If no entry for the value possessed by the switch expression is found, a jump to the default statement is generated.

There is a common special case that can be implemented even more efficiently than by an n-way branch. If the values all lie in some small range, say min to max, and the number of different values is a reasonable fraction of $max - min$, then we can construct an array of $max - min$ "buckets," where bucket $j - min$ contains the label of the statement with value j; any bucket that would otherwise remain unfilled contains the default label.

To perform the switch, evaluate the expression to obtain the value j; check that it is in the range min to max and transfer indirectly to the table entry at offset $j - min$. For example, if the expression is of type character, a table of,

https://hemanthrajhemu.github.io

say, 128 entries (depending on the character set) may be created and transferred through with no range testing.

6.8.2 Syntax-Directed Translation of Switch-Statements

The intermediate code in Fig. 6.49 is a convenient translation of the switch-statement in Fig. 6.48. The tests all appear at the end so that a simple code generator can recognize the multiway branch and generate efficient code for it, using the most appropriate implementation suggested at the beginning of this section.

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Figure 6.49: Translation of a switch-statement

The more straightforward sequence shown in Fig. 6.50 would require the compiler to do extensive analysis to find the most efficient implementation. Note that it is inconvenient in a one-pass compiler to place the branching statements at the beginning, because the compiler could not then emit code for each of the statements S_i as it saw them.

To translate into the form of Fig. 6.49, when we see the keyword **switch**, we generate two new labels `test` and `next`, and a new temporary t. Then, as we parse the expression E, we generate code to evaluate E into t. After processing E, we generate the jump `goto test`.

Then, as we see each **case** keyword, we create a new label L_i and enter it into the symbol table. We place in a queue, used only to store cases, a value-label pair consisting of the value V_i of the case constant and L_i (or a pointer to the symbol-table entry for L_i). We process each statement **case** V_i: S_i by emitting the label L_i attached to the code for S_i, followed by the jump `goto next`.

https://hemanthrajhemu.github.io

```
          code to evaluate E into t
          if t != V1 goto L1
          code for S1
          goto next
L1:       if t != V2 goto L2
          code for S2
          goto next
L2:
          ...
Ln-2:     if t != Vn-1 goto Ln-1
          code for Sn-1
          goto next
Ln-1:     code for Sn
next:
```

Figure 6.50: Another translation of a switch statement

When the end of the switch is found, we are ready to generate the code for the n-way branch. Reading the queue of value-label pairs, we can generate a sequence of three-address statements of the form shown in Fig. 6.51. There, t is the temporary holding the value of the selector expression E, and L_n is the label for the default statement.

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
label next
```

Figure 6.51: Case three-address-code instructions used to translate a switch-statement

The `case` t V_i L_i instruction is a synonym for `if` t = V_i `goto` L_i in Fig. 6.49, but the `case` instruction is easier for the final code generator to detect as a candidate for special treatment. At the code-generation phase, these sequences of `case` statements can be translated into an n-way branch of the most efficient type, depending on how many there are and whether the values fall into a small range.

6.8.3 Exercises for Section 6.8

! **Exercise 6.8.1:** In order to translate a switch-statement into a sequence of `case`-statements as in Fig. 6.51, the translator needs to create the list of value-

https://hemanthrajhemu.github.io

label pairs, as it processes the source code for the switch. We can do so, using an additional translation that accumulates just the pairs. Sketch a syntax-direction definition that produces the list of pairs, while also emitting code for the statements S_i that are the actions for each case.

6.9 Intermediate Code for Procedures

Procedures and their implementation will be discussed at length in Chapter 7, along with the run-time management of storage for names. We use the term function in this section for a procedure that returns a value. We briefly discuss function declarations and three-address code for function calls. In three-address code, a function call is unraveled into the evaluation of parameters in preparation for a call, followed by the call itself. For simplicity, we assume that parameters are passed by value; parameter-passing methods are discussed in Section 1.6.6.

Example 6.25: Suppose that `a` is an array of integers, and that `f` is a function from integers to integers. Then, the assignment

```
n = f(a[i]);
```

might translate into the following three-address code:

```
1)  t1 = i * 4
2)  t2 = a [ t1 ]
3)  param t2
4)  t3 = call f, 1
5)  n = t3
```

The first two lines compute the value of the expression `a[i]` into temporary t_2, as discussed in Section 6.4. Line 3 makes t_2 an actual parameter for the call on line 4 of `f` with one parameter. Line 5 assigns the value returned by the function call to t_3. Line 6 assigns the returned value to `n`. □

The productions in Fig. 6.52 allow function definitions and function calls. (The syntax generates unwanted commas after the last parameter, but is good enough for illustrating translation.) Nonterminals D and T generate declarations and types, respectively, as in Section 6.3. A function definition generated by D consists of keyword **define**, a return type, the function name, formal parameters in parentheses and a function body consisting of a statement. Nonterminal F generates zero or more formal parameters, where a formal parameter consists of a type followed by an identifier. Nonterminals S and E generate statements and expressions, respectively. The production for S adds a statement that returns the value of an expression. The production for E adds function calls, with actual parameters generated by A. An actual parameter is an expression.

https://hemanthrajhemu.github.io

$$
\begin{array}{lcl}
D & \rightarrow & \textbf{define}\ T\ \textbf{id}\ (\ F\ )\ \{\ S\ \} \\
F & \rightarrow & \epsilon\ \mid\ T\ \textbf{id}\ ,\ F \\
S & \rightarrow & \textbf{return}\ E\ ; \\
E & \rightarrow & \textbf{id}\ (\ A\ ) \\
A & \rightarrow & \epsilon\ \mid\ E\ ,\ A
\end{array}
$$

Figure 6.52: Adding functions to the source language

Function definitions and function calls can be translated using concepts that have already been introduced in this chapter.

- *Function types.* The type of a function must encode the return type and the types of the formal parameters. Let *void* be a special type that represents no parameter or no return type. The type of a function *pop*() that returns an integer is therefore "function from *void* to *integer*." Function types can be represented by using a constructor *fun* applied to the return type and an ordered list of types for the parameters.

- *Symbol tables.* Let s be the top symbol table when the function definition is reached. The function name is entered into s for use in the rest of the program. The formal parameters of a function can be handled in analogy with field names in a record (see Fig. 6.18. In the production for D, after seeing **define** and the function name, we push s and set up a new symbol table

$$Env.push(top); \quad top = \textbf{new}\ Env(top);$$

Call the new symbol table, t. Note that *top* is passed as a parameter in **new** *Env*(*top*), so the new symbol table t can be linked to the previous one, s. The new table t is used to translate the function body. We revert to the previous symbol table s after the function body is translated.

- *Type checking.* Within expressions, a function is treated like any other operator. The discussion of type checking in Section 6.5.2 therefore carries over, including the rules for coercions. For example, if f is a function with a parameter of type real, then the integer 2 is coerced to a real in the call $f(2)$.

- *Function calls.* When generating three-address instructions for a function call $\textbf{id}(E, E, \ldots, E)$, it is sufficient to generate the three-address instructions for evaluating or reducing the parameters E to addresses, followed by a `param` instruction for each parameter. If we do not want to mix the parameter-evaluating instructions with the *param* instructions, the attribute $E.addr$ for each expression E can be saved in a data structure

https://hemanthrajhemu.github.io

such as a queue. Once all the expressions are translated, the `param` instructions can be generated as the queue is emptied.

The procedure is such an important and frequently used programming construct that it is imperative for a compiler to good code for procedure calls and returns. The run-time routines that handle procedure parameter passing, calls, and returns are part of the run-time support package. Mechanisms for run-time support are discussed in Chapter 7.

6.10 Summary of Chapter 6

The techniques in this chapter can be combined to build a simple compiler front end, like the one in Appendix A. The front end can be built incrementally:

- *Pick an intermediate representation*: An intermediate representation is typically some combination of a graphical notation and three-address code. As in syntax trees, a node in a graphical notation represents a construct; the children of a node represent its subconstructs. Three address code takes its name from instructions of the form $x = y \textbf{ op } z$, with at most one operator per instruction. There are additional instructions for control flow.

- *Translate expressions*: Expressions with built-up operations can be unwound into a sequence of individual operations by attaching actions to each production of the form $E \rightarrow E_1 \textbf{ op } E_2$. The action either creates a node for E with the nodes for E_1 and E_2 as children, or it generates a three-address instruction that applies **op** to the addresses for E_1 and E_2 and puts the result into a new temporary name, which becomes the address for E.

- *Check types*: The type of an expression $E_1 \textbf{ op } E_2$ is determined by the operator **op** and the types of E_1 and E_2. A coercion is an implicit type conversion, such as from *integer* to *float*. Intermediate code contains explicit type conversions to ensure an exact match between operand types and the types expected by an operator.

- *Use a symbol table to implement declarations*: A declaration specifies the type of a name. The width of a type is the amount of storage needed for a name with that type. Using widths, the relative address of a name at run time can be computed as an offset from the start of a data area. The type and relative address of a name are put into the symbol table due to a declaration, so the translator can subsequently get them when the name appears in an expression.

- *Flatten arrays*: For quick access, array elements are stored in consecutive locations. Arrays of arrays are flattened so they can be treated as a one-

https://hemanthrajhemu.github.io

dimensional array of individual elements. The type of an array is used to calculate the address of an array element relative to the base of the array.

- *Generate jumping code for boolean expressions*: In short-circuit or jumping code, the value of a boolean expression is implicit in the position reached in the code. Jumping code is useful because a boolean expression B is typically used for control flow, as in **if** (B) S. Boolean values can be computed by jumping to `t = true` or `t = false`, as appropriate, where `t` is a temporary name. Using labels for jumps, a boolean expression can be translated by inheriting labels corresponding to its true and false exits. The constants *true* and *false* translate into a jump to the true and false exits, respectively.

- *Implement statements using control flow*: Statements can be translated by inheriting a label *next*, where *next* marks the first instruction after the code for this statement. The conditional $S \rightarrow$ **if** (B) S_1 can be translated by attaching a new label marking the beginning of the code for S_1 and passing the new label and $S.next$ for the true and false exits, respectively, of B.

- *Alternatively, use backpatching*: Backpatching is a technique for generating code for boolean expressions and statements in one pass. The idea is to maintain lists of incomplete jumps, where all the jump instructions on a list have the same target. When the target becomes known, all the instructions on its list are completed by filling in the target.

- *Implement records*: Field names in a record or class can be treated as a sequence of declarations. A record type encodes the types and relative addresses of the fields. A symbol table object can be used for this purpose.

6.11 References for Chapter 6

Most of the techniques in this chapter stem from the flurry of design and implementation activity around Algol 60. Syntax-directed translation into intermediate code was well established by the time Pascal [11] and C [6, 9] were created.

UNCOL (for Universal Compiler Oriented Language) is a mythical universal intermediate language, sought since the mid 1950's. Given an UNCOL, compilers could be constructed by hooking a front end for a given source language with a back end for a given target language [10]. The bootstrapping techniques given in the report [10] are routinely used to retarget compilers.

The UNCOL ideal of mixing and matching front ends with back ends has been approached in a number of ways. A retargetable compiler consists of one front end that can be put together with several back ends to implement a given language on several machines. Neliac was an early example of a language with a retargetable compiler [5] written in its own language. Another approach is to

https://hemanthrajhemu.github.io

retrofit a front end for a new language onto an existing compiler. Feldman [2] describes the addition of a Fortran 77 front end to the C compilers [6] and [9]. GCC, the GNU Compiler Collection [3], supports front ends for C, C++, Objective-C, Fortran, Java, and Ada.

Value numbers and their implementation by hashing are from Ershov [1].

The use of type information to improve the security of Java bytecodes is described by Gosling [4].

Type inference by using unification to solve sets of equations has been rediscovered several times; its application to ML is described by Milner [7]. See Pierce [8] for a comprehensive treatment of types.

1. Ershov, A. P., "On programming of arithmetic operations," *Comm. ACM* **1**:8 (1958), pp. 3–6. See also *Comm. ACM* **1**:9 (1958), p. 16.

2. Feldman, S. I., "Implementation of a portable Fortran 77 compiler using modern tools," *ACM SIGPLAN Notices* **14**:8 (1979), pp. 98–106

3. GCC home page `http://gcc.gnu.org/`, Free Software Foundation.

4. Gosling, J., "Java intermediate bytecodes," *Proc. ACM SIGPLAN Workshop on Intermediate Representations* (1995), pp. 111–118.

5. Huskey, H. D., M. H. Halstead, and R. McArthur, "Neliac — a dialect of Algol," *Comm. ACM* **3**:8 (1960), pp. 463–468.

6. Johnson, S. C., "A tour through the portable C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

7. Milner, R., "A theory of type polymorphism in programming," *J. Computer and System Sciences* **17**:3 (1978), pp. 348–375.

8. Pierce, B. C., *Types and Programming Languages*, MIT Press, Cambridge, Mass., 2002.

9. Ritchie, D. M., "A tour through the UNIX C compiler," Bell Telephone Laboratories, Inc., Murray Hill, N. J., 1979.

10. Strong, J., J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel, "The problem of programming communication with changing machines: a proposed solution," *Comm. ACM* **1**:8 (1958), pp. 12–18. Part 2: **1**:9 (1958), pp. 9–15. Report of the Share Ad-Hoc committee on Universal Languages.

11. Wirth, N. "The design of a Pascal compiler," *Software—Practice and Experience* **1**:4 (1971), pp. 309–333.

https://hemanthrajhemu.github.io

Chapter 8

Code Generation

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig. 8.1.

The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the *back end*, may make multiple passes over the IR before generating the target program. Code optimization is discussed in detail in Chapter 9. The techniques presented in this chapter can be used whether or not an optimization phase occurs before code generation.

A code generator has three primary tasks: instruction selection, register

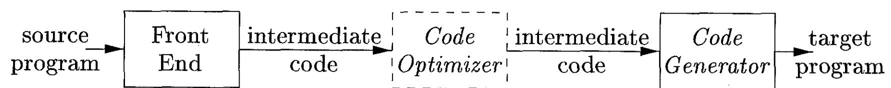


Figure 8.1: Position of code generator

https://hemanthrajhemu.github.io

allocation and assignment, and instruction ordering. The importance of these tasks is outlined in Section 8.1. Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

This chapter presents algorithms that code generators can use to translate the IR into a sequence of target language instructions for simple register machines. The algorithms will be illustrated by using the machine model in Section 8.2. Chapter 10 covers the problem of code generation for complex modern machines that support a great deal of parallelism within a single instruction.

After discussing the broad issues in the design of a code generator, we show what kind of target code a compiler needs to generate to support the abstractions embodied in a typical source language. In Section 8.3, we outline implementations of static and stack allocation of data areas, and show how names in the IR can be converted into addresses in the target code.

Many code generators partition IR instructions into "basic blocks," which consist of sequences of instructions that are always executed together. The partitioning of the IR into basic blocks is the subject of Section 8.4. The following section presents simple local transformations that can be used to transform basic blocks into modified basic blocks from which more efficient code can be generated. These transformations are a rudimentary form of code optimization, although the deeper theory of code optimization will not be taken up until Chapter 9. An example of a useful, local transformation is the discovery of common subexpressions at the level of intermediate code and the resultant replacement of arithmetic operations by simpler copy operations.

Section 8.6 presents a simple code-generation algorithm that generates code for each statement in turn, keeping operands in registers as long as possible. The output of this kind of code generator can be readily improved by peephole optimization techniques such as those discussed in the following Section 8.7.

The remaining sections explore instruction selection and register allocation.

8.1 Issues in the Design of a Code Generator

While the details are dependent on the specifics of the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are encountered in the design of almost all code generators.

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that a code generator might face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

https://hemanthrajhemu.github.io

8.1.1 Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's. Many of the algorithms in this chapter are couched in terms of the representations considered in Chapter 6: three-address code, trees, and DAG's. The techniques we discuss can be applied, however, to the other intermediate representations as well.

In this chapter, we assume that the front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers. We also assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type-conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

8.1.2 The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.

However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The inter-

https://hemanthrajhemu.github.io

preter provides software compatibility across multiple platforms, a major factor in the success of Java.

To overcome the high performance penalty of interpretation, which can be on the order of a factor of 10, *just-in-time* (JIT) Java compilers have been created. These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine. Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine, bypassing the Java bytecodes entirely.

Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.

Producing a relocatable machine-language program (often called an *object module*) as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

In this chapter, we shall use a very simple RISC-like computer as our target machine. We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines. For readability, we use assembly code as the target language . As long as addresses can be calculated from offsets and other information stored in the symbol table, the code generator can produce relocatable or absolute addresses for names just as easily as symbolic addresses.

8.1.3 Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs

https://hemanthrajhemu.github.io

further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form `x = y + z`, where `x`, `y`, and `z` are statically allocated, can be translated into the code sequence

```
LD  R0, y         // R0 = y          (load y into register R0)
ADD R0, R0, z     // R0 = R0 + z     (add z to R0)
ST  x, R0         // x = R0          (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD  R0, b        // R0 = b
ADD R0, R0, c    // R0 = R0 + c
ST  a, R0        // a = R0
LD  R0, a        // R0 = a
ADD R0, R0, e    // R0 = R0 + e
ST  d, R0        // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if `a` is not subsequently used.

The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

For example, if the target machine has an "increment" instruction (`INC`), then the three-address statement `a = a + 1` may be implemented more efficiently by the single instruction `INC a`, rather than by a more obvious sequence that loads `a` into a register, adds one to the register, and then stores the result back into `a`:

https://hemanthrajhemu.github.io

```
LD  R0, a       // R0 = a
ADD R0, R0, #1  // R0 = R0 + 1
ST  a, R0       // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine-code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

In Section 8.9 we shall see that instruction selection can be modeled as a tree-pattern matching process in which we represent the IR and the machine instructions as trees. We then attempt to "tile" an IR tree with a set of subtrees that correspond to machine instructions. If we associate a cost with each machine-instruction subtree, we can use dynamic programming to generate optimal code sequences. Dynamic programming is discussed in Section 8.11.

8.1.4 Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two subproblems:

1. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.

2. *Register assignment*, during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Example 8.1: Certain machines require *register-pairs* (an even and next odd-numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

```
M x, y
```

where x, the multiplicand, is the even register of an even/odd register pair and y, the multiplier, is the odd register. The product occupies the entire even/odd register pair. The division instruction is of the form

https://hemanthrajhemu.github.io

```
D x, y
```

where the dividend occupies an even/odd register pair whose even register is `x`; the divisor is `y`. After division, the even register holds the remainder and the odd register the quotient.

Now, consider the two three-address code sequences in Fig. 8.2 in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 8.3.

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d

   (a)                (b)
```

Figure 8.2: Two three-address code sequences

```
L    R1,a          L    R0, a
A    R1,b          A    R0, b
M    R0,c          A    R0, c
D    R0,d          SRDA R0, 32
ST   R1,t          D    R0, d
                   ST   R1, t

   (a)                (b)
```

Figure 8.3: Optimal machine-code sequences

R*i* stands for register *i*. `SRDA` stands for Shift-Right-Double-Arithmetic and `SRDA R0,32` shifts the dividend into `R1` and clears `R0` so all bits equal its sign bit. `L`, `ST`, and `A` stand for load, store, and add, respectively. Note that the optimal choice for the register into which `a` is to be loaded depends on what will ultimately happen to `t`. □

Strategies for register allocation and assignment are discussed in Section 8.8. Section 8.10 shows that for certain classes of machines we can construct code sequences that evaluate expressions using as few registers as possible.

8.1.5 Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid

https://hemanthrajhemu.github.io

the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator. In Chapter 10, we shall study code scheduling for pipelined machines that can execute several operations in a single clock cycle.

8.2 The Target Language

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine. In this chapter, we shall use as a target language assembly code for a simple computer that is representative of many register machines. However, the code-generation techniques presented in this chapter can be used on many other classes of machines as well.

8.2.1 A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with n general-purpose registers, `R0`, `R1`, ..., `R`$n-1$. A full-fledged assembly language would have scores of instructions. To avoid hiding the concepts in a myriad of details, we shall use a very limited set of instructions and assume that all operands are integers. Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction. We assume the following kinds of instructions are available:

- *Load* operations: The instruction `LD` *dst, addr* loads the value in location *addr* into location *dst*. This instruction denotes the assignment *dst* = *addr*. The most common form of this instruction is `LD` r, x which loads the value in location x into register r. An instruction of the form `LD` r_1, r_2 is a *register-to-register copy* in which the contents of register r_2 are copied into register r_1.

- *Store* operations: The instruction `ST` x, r stores the value in register r into the location x. This instruction denotes the assignment $x = r$.

- *Computation* operations of the form *OP dst*, src_1, src_2, where *OP* is a operator like `ADD` or `SUB`, and *dst*, src_1, and src_2 are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP* to the values in locations src_1 and src_2, and place the result of this operation in location *dst*. For example, `SUB` r_1, r_2, r_3 computes $r_1 = r_2 - r_3$. Any value formerly stored in r_1 is lost, but if r_1 is r_2 or r_3, the old value is read first. Unary operators that take only one operand do not have a src_2.

https://hemanthrajhemu.github.io

- *Unconditional jumps*: The instruction `BR` L causes control to branch to the machine instruction with label L. (`BR` stands for *branch*.)

- *Conditional jumps* of the form `B`*cond* r, L, where r is a register, L is a label, and *cond* stands for any of the common tests on values in the register r. For example, `BLTZ` r, L causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a variable name x referring to the memory location that is reserved for x (that is, the l-value of x).

- A location can also be an indexed address of the form $a(r)$, where a is a variable and r is a register. The memory location denoted by $a(r)$ is computed by taking the l-value of a and adding to it the value in register r. For example, the instruction `LD R1, a(R2)` has the effect of setting `R1` $= contents(\texttt{a} + contents(\texttt{R2}))$, where $contents(x)$ denotes the contents of the register or memory location represented by x. This addressing mode is useful for accessing arrays, where a is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array a.

- A memory location can be an integer indexed by a register. For example, `LD R1, 100(R2)` has the effect of setting `R1` $= contents(100 + contents(\texttt{R2}))$, that is, of loading into `R1` the value in the memory location obtained by adding 100 to the contents of register `R2`. This feature is useful for following pointers, as we shall see in the example below.

- We also allow two indirect addressing modes: `*`r means the memory location found in the location represented by the contents of register r and `*100(`r`)` means the memory location found in the location obtained by adding 100 to the contents of r. For example, `LD R1, *100(R2)` has the effect of setting `R1` $= contents(contents(100 + contents(\texttt{R2})))$, that is, of loading into `R1` the value in the memory location stored in the memory location obtained by adding 100 to the contents of register `R2`.

- Finally, we allow an immediate constant addressing mode. The constant is prefixed by `#`. The instruction `LD R1, #100` loads the integer 100 into register `R1`, and `ADD R1, R1, #100` adds the integer 100 into register `R1`.

Comments at the end of instructions are preceded by `//`.

Example 8.2: The three-address statement `x = y - z` can be implemented by the machine instructions:

https://hemanthrajhemu.github.io

```
LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST  x, R1           // x = R1
```

We can do better, perhaps. One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible. For example, y and/or z may have been computed in a register, and if so we can avoid the LD step(s). Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed.

Suppose a is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of a are indexed starting at 0. We may execute the three-address instruction b = a[i] by the machine instructions:

```
LD  R1, i           // R1 = i
MUL R1, R1, 8       // R1 = R1 * 8
LD  R2, a(R1)       // R2 = contents(a + contents(R1))
ST  b, R2           // b = R2
```

That is, the second step computes $8i$, and the third step places in register R2 the value in the ith element of a — the one found in the location that is $8i$ bytes past the base address of the array a.

Similarly, the assignment into the array a represented by three-address instruction a[j] = c is implemented by:

```
LD  R1, c           // R1 = c
LD  R2, j           // R2 = j
MUL R2, R2, 8       // R2 = R2 * 8
ST  a(R2), R1       // contents(a +  contents(R2)) = R1
```

To implement a simple pointer indirection, such as the three-address statement x = *p, we can use machine instructions like:

```
LD  R1, p           // R1 = p
LD  R2, 0(R1)       // R2 = contents(0 + contents(R1))
ST  x, R2           // x = R2
```

The assignment through a pointer *p = y is similarly implemented in machine code by:

```
LD  R1, p           // R1 = p
LD  R2, y           // R2 = y
ST  0(R1), R2       // contents(0 + contents(R1)) = R2
```

Finally, consider a conditional-jump three-address instruction like

```
if x < y goto L
```

https://hemanthrajhemu.github.io

The machine-code equivalent would be something like:

```
LD   R1, x         // R1 = x
LD   R2, y         // R2 = y
SUB  R1, R1, R2    // R1 = R1 - R2
BLTZ R1, M         // if R1 < 0 jump to M
```

Here, `M` is the label that represents the first machine instruction generated from the three-address instruction that has label `L`. As for any three-address instruction, we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored. □

8.2.2 Program and Instruction Costs

We often associate a cost with compiling and running a program. Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given source program is an undecidable problem in general, and many of the subproblems involved are NP-hard. As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

For the remainder of this chapter, we shall assume each target-language instruction has an associated cost. For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction. Some examples:

- The instruction `LD R0, R1` copies the contents of register `R1` into register `R0`. This instruction has a cost of one because no additional memory words are required.

- The instruction `LD R0, M` loads the contents of memory location `M` into register `R0`. The cost is two since the address of memory location `M` is in the word following the instruction.

- The instruction `LD R1, *100(R2)` loads into register `R1` the value given by *contents*(*contents*(100 + *contents*(R2))). The cost is three because the constant 100 is stored in the word following the instruction.

https://hemanthrajhemu.github.io

In this chapter we assume the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input. Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs. We shall see that in some situations we can actually generate optimal code for expressions on certain classes of register machines.

8.2.3 Exercises for Section 8.2

Exercise 8.2.1: Generate code for the following three-address statements assuming all variables are stored in memory locations.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) The two statements

```
x = b * c
y = a + x
```

Exercise 8.2.2: Generate code for the following three-address statements assuming a and b are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

https://hemanthrajhemu.github.io

Exercise 8.2.3: Generate code for the following three-address sequence assuming that p and q are in memory locations:

```
y = *q
q = q + 4
*p = y
p = p + 4
```

Exercise 8.2.4: Generate code for the following sequence assuming that x, y, and z are in memory locations:

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

Exercise 8.2.5: Generate code for the following sequence assuming hat n is in a memory location:

```
    s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

Exercise 8.2.6: Determine the costs of the following instruction sequences:

a)
```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

b)
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

c)
```
LD  R0, c
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

d)
```
LD R0, p
LD R1, 0(R0)
ST x, R1
```

https://hemanthrajhemu.github.io

e)
```
LD R0, p
LD R1, x
ST 0(R0), R1
```

f)
```
LD   R0, x
LD   R1, y
SUB  R0, R0, R1
BLTZ *R3, R0
```

8.3 Addresses in the Target Code

In this section, we show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation. In Section 7.1, we described how each executing program runs in its own logical address space that was partitioned into four code and data areas:

1. A statically determined area *Code* that holds the executable target code. The size of the target code can be determined at compile time.

2. A statically determined data area *Static* for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.

3. A dynamically managed area *Heap* for holding data objects that are allocated and freed during program execution. The size of the *Heap* cannot be determined at compile time.

4. A dynamically managed area *Stack* for holding activation records as they are created and destroyed during procedure calls and returns. Like the *Heap*, the size of the *Stack* cannot be determined at compile time.

8.3.1 Static Allocation

To illustrate code generation for simplified procedure calls and returns, we shall focus on the following three-address statements:

- `call` *callee*
- `return`
- `halt`
- `action`, which is a placeholder for other three-address statements.

The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table. We shall first illustrate how to store the return address in an activation record on a procedure

https://hemanthrajhemu.github.io

call and how to return control to it after the procedure call. For convenience, we assume the first location in the activation holds the return address.

Let us first consider the code needed to implement the simplest case, static allocation. Here, a `call` *callee* statement in the intermediate code can be implemented by a sequence of two target-machine instructions:

```
ST   callee.staticArea, #here + 20
BR   callee.codeArea
```

The ST instruction saves the return address at the beginning of the activation record for *callee*, and the BR transfers control to the target code for the called procedure *callee*. The attribute before *callee.staticArea* is a constant that gives the address of the beginning of the activation record for *callee*, and the attribute *callee.codeArea* is a constant referring to the address of the first instruction of the called procedure *callee* in the *Code* area of the run-time memory.

The operand #*here* + 20 in the ST instruction is the literal return address; it is the address of the instruction following the BR instruction. We assume that #*here* is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of 5 words or 20 bytes.

The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is HALT, which returns control to the operating system. A `return` *callee* statement can be implemented by a simple jump instruction

```
BR   *callee.staticArea
```

which transfers control to the address saved at the beginning of the activation record for *callee*.

Example 8.3: Suppose we have the following three-address code:

```
                // code for c
action1
call p
action2
halt
                // code for p
action3
return
```

Figure 8.4 shows the target program for this three-address code. We use the pseudoinstruction ACTION to represent the sequence of machine instructions to execute the statement `action`, which represents three-address code that is not relevant for this discussion. We arbitrarily start the code for procedure `c` at address 100 and for procedure `p` at address 200. We that assume each ACTION instruction takes 20 bytes. We further assume that the activation records for these procedures are statically allocated starting at locations 300 and 364, respectively.

The instructions starting at address 100 implement the statements

https://hemanthrajhemu.github.io

```
action₁; call p; action₂; halt
```

of the first procedure c. Execution therefore starts with the instruction $\mathtt{ACTION}_1$ at address 100. The ST instruction at address 120 saves the return address 140 in the machine-status field, which is the first word in the activation record of p. The BR instruction at address 132 transfers control the first instruction in the target code of the called procedure p.

```
                            // code for c
100:  ACTION1               // code for action1
120:  ST 364, #140          // save return address 140 in location 364
132:  BR 200                // call p
140:  ACTION2
160:  HALT                  // return to operating system
      ...
                            // code for p
200:  ACTION3
220:  BR *364               // return to address saved in location 364
      ...
                            // 300-363 hold activation record for c
300:                        // return address
304:                        // local data for c
      ...
                            // 364-451 hold activation record for p
364:                        // return address
368:                        // local data for p
```

Figure 8.4: Target code for static allocation

After executing $\mathtt{ACTION}_3$, the jump instruction at location 220 is executed. Since location 140 was saved at address 364 by the call sequence above, *364 represents 140 when the BR statement at address 220 is executed. Therefore, when procedure p terminates, control returns to address 140 and execution of procedure c resumes. □

8.3.2 Stack Allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, however, the position of an activation record for a procedure is not known until run time. This position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register. The indexed address mode of our target machine is convenient for this purpose.

Relative addresses in an activation record can be taken as offsets from any known position in the activation record, as we saw in Chapter 7. For conve-

https://hemanthrajhemu.github.io

nience, we shall use positive offsets by maintaining in a register SP a pointer to the beginning of the activation record on top of the stack. When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure. After control returns to the caller, we decrement SP, thereby deallocating the activation record of the called procedure.

The code for the first procedure initializes the stack by setting SP to the start of the stack area in memory:

```
LD   SP, #stackStart              // initialize the stack
code for the first procedure
HALT                              // terminate execution
```

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

```
ADD  SP, SP, #caller.recordSize   // increment stack pointer
ST   *SP, #here + 16              // save return address
BR   callee.codeArea              // return to caller
```

The operand *#caller.recordSize* represents the size of an activation record, so the ADD instruction makes SP point to the next activation record. The operand *#here* + 16 in the ST instruction is the address of the instruction following BR; it is saved in the address pointed to by SP.

The return sequence consists of two parts. The called procedure transfers control to the return address using

```
BR  *0(SP)                        // return to caller
```

The reason for using *0(SP) in the BR instruction is that we need two levels of indirection: 0(SP) is the address of the first word in the activation record and *0(SP) is the return address saved there.

The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value. That is, after the subtraction SP points to the beginning of the activation record of the caller:

```
SUB  SP, SP, #caller.recordSize   // decrement stack pointer
```

Chapter 7 contains a broader discussion of calling sequences and the trade-offs in the division of labor between the calling and called procedures.

Example 8.4: The program in Fig. 8.5 is an abstraction of the quicksort program in the previous chapter. Procedure q is recursive, so more than one activation of q can be alive at the same time.

Suppose that the sizes of the activation records for procedures m, p, and q have been determined to be *msize*, *psize*, and *qsize*, respectively. The first word in each activation record will hold a return address. We arbitrarily assume that the code for these procedures starts at addresses 100, 200, and 300, respectively,

https://hemanthrajhemu.github.io

```
                              // code for m
action1
call q
action2
halt
                              // code for p
action3
return
                              // code for q
action4
call p
action5
call q
action6
call q
return
```

Figure 8.5: Code for Example 8.4

and that the stack starts at address 600. The target program is shown in Figure 8.6.

We assume that $\texttt{ACTION}_4$ contains a conditional jump to the address 456 of the return sequence from q; otherwise, the recursive procedure q is condemned to call itself forever.

If *msize*, *psize*, and *qsize* are 20, 40, and 60, respectively, the first instruction at address 100 initializes the SP to 600, the starting address of the stack. SP holds 620 just before control transfers from m to q, because *msize* is 20. Subsequently, when q calls p, the instruction at address 320 increments SP to 680, where the activation record for p begins; SP reverts to 620 after control returns to q. If the next two recursive calls of q return immediately, the maximum value of SP during this execution 680. Note, however, that the last stack location used is 739, since the activation record of q starting at location 680 extends for 60 bytes. □

8.3.3 Run-Time Addresses for Names

The storage-allocation strategy and the layout of local data in an activation record for a procedure determine how the storage for names is accessed. In Chapter 6, we assumed that a name in a three-address statement is really a pointer to a symbol-table entry for that name. This approach has a significant advantage; it makes the compiler more portable, since the front end need not be changed even when the compiler is moved to a different machine where a different run-time organization is needed. On the other hand, generating the specific sequence of access steps while generating intermediate code can be of

https://hemanthrajhemu.github.io

```
                                 // code for m
100:  LD SP, #600                // initialize the stack
108:  ACTION1                    // code for action1
128:  ADD SP, SP, #msize         // call sequence begins
136:  ST *SP, #152               // push return address
144:  BR 300                     // call q
152:  SUB SP, SP, #msize         // restore SP
160:  ACTION12
180:  HALT
      ...
                                 // code for p
200:  ACTION3
220:  BR *0(SP)                  // return
      ...
                                 // code for q
300:  ACTION4                    // contains a conditional jump to 456
320:  ADD SP, SP, #qsize
328:  ST *SP, #344               // push return address
336:  BR 200                     // call p
344:  SUB SP, SP, #qsize
352:  ACTION5
372:  ADD SP, SP, #qsize
380:  BR *SP, #396               // push return address
388:  BR 300                     // call q
396:  SUB SP, SP, #qsize
404:  ACTION6
424:  ADD SP, SP, #qsize
432:  ST *SP, #440               // push return address
440:  BR 300                     // call q
448:  SUB SP, SP, #qsize
456:  BR *0(SP)                  // return
      ...
600:                             // stack starts here
```

Figure 8.6: Target code for stack allocation

https://hemanthrajhemu.github.io

significant advantage in an optimizing compiler, since it lets the optimizer take advantage of details it would not see in the simple three-address statement.

In either case, names must eventually be replaced by code to access storage locations. We thus consider some elaborations of the simple three-address copy statement `x = 0`. After the declarations in a procedure are processed, suppose the symbol-table entry for `x` contains a relative address 12 for `x`. For consider the case in which `x` is in a statically allocated area beginning at address *static*. Then the actual run-time address of `x` is $static + 12$. Although the compiler can eventually determine the value of $static + 12$ at compile time, the position of the static area may not be known when intermediate code to access the name is generated. In that case, it makes sense to generate three-address code to "compute" $static + 12$, with the understanding that this computation will be carried out during the code generation phase, or possibly by the loader, before the program runs. The assignment `x = 0` then translates into

```
static[12] = 0
```

If the static area starts at address 100, the target code for this statement is

```
LD 112, #0
```

8.3.4 Exercises for Section 8.3

Exercise 8.3.1: Generate code for the following three-address statements assuming stack allocation where register `SP` points to the top of the stack.

```
call p
call q
return
call r
return
return
```

Exercise 8.3.2: Generate code for the following three-address statements assuming stack allocation where register `SP` points to the top of the stack.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) The two statements

```
x = b * c
y = a + x
```

https://hemanthrajhemu.github.io

Exercise 8.3.3: Generate code for the following three-address statements again assuming stack allocation and assuming `a` and `b` are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

8.4 Basic Blocks and Flow Graphs

This section introduces a graph representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code-generation algorithm. Code generation benefits from context. We can do a better job of register allocation if we know how values are defined and used, as we shall see in Section 8.8. We can do a better job of instruction selection by looking at sequences of three-address statements, as we shall see in Section 8.9.

The representation is constructed as follows:

1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that

   (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.

   (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.

2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.

https://hemanthrajhemu.github.io

The Effect of Interrupts

The notion that control, once it reaches the beginning of a basic block is certain to continue through to the end requires a bit of thought. There are many reasons why an interrupt, not reflected explicitly in the code, could cause control to leave the block, perhaps never to return. For example, an instruction like `x = y/z` appears not to affect control flow, but if z is 0 it could actually cause the program to abort.

We shall not worry about such possibilities. The reason is as follows. The purpose of constructing basic blocks is to optimize the code. Generally, when an interrupt occurs, either it will be handled and control will come back to the instruction that caused the interrupt, as if control had never deviated, or the program will halt with an error. In the latter case, it doesn't matter how we optimized the code, even if we depended on control reaching the end of the basic block, because the program didn't produce its intended result anyway.

Starting in Chapter 9, we discuss transformations on flow graphs that turn the original intermediate code into "optimized" intermediate code from which better target code can be generated. The "optimized" intermediate code is turned into machine code using the code-generation techniques in this chapter.

8.4.1 Basic Blocks

Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm 8.5: Partitioning three-address instructions into basic blocks.

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are *leaders*, that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.

https://hemanthrajhemu.github.io

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program. □

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Figure 8.7: Intermediate code to set a 10×10 matrix to an identity matrix

Example 8.6: The intermediate code in Fig. 8.7 turns a 10×10 matrix **a** into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 8.8. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix **a** is stored in row-major form.

for i from 1 to 10 **do**
    for j from 1 to 10 **do**
        $a[i, j] = 0.0;$
for i from 1 to 10 **do**
    $a[i, i] = 1.0;$

Figure 8.8: Source code for Fig. 8.7

https://hemanthrajhemu.github.io

First, instruction 1 is a leader by rule (1) of Algorithm 8.5. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17. □

8.4.2 Next-Use Information

Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

The *use* of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can flow from statement i to j along a path that has no intervening assignments to x, then we say statement j *uses* the value of x computed at statement i. We further say that x is *live* at statement i.

We wish to determine for each three-address statement $x = y + z$ what the next uses of x, y, and z are. For the present, we do not concern ourselves with uses outside the basic block containing this three-address statement.

Our algorithm to determine liveness and next-use information makes a backward pass over each basic block. We store the information in the symbol table. We can easily scan a stream of three-address statements to find the ends of basic blocks as in Algorithm 8.5. Since procedures can have arbitrary side effects, we assume for convenience that each procedure call starts a new basic block.

Algorithm 8.7: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all nontemporary variables in B as being live on exit.

OUTPUT: At each statement i: $x = y + z$ in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: $x = y + z$ in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and y.

https://hemanthrajhemu.github.io

2. In the symbol table, set x to "not live" and "no next use."
3. In the symbol table, set y and z to "live" and the next uses of y and z to i.

Here we have used $+$ as a symbol representing any operator. If the three-address statement i is of the form $x = + y$ or $x = y$, the steps are the same as above, ignoring z. Note that the order of steps (2) and (3) may not be interchanged because x may be y or z. □

8.4.3 Flow Graphs

Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a *predecessor* of C, and C is a *successor* of B.

Often we add two nodes, called the *entry* and *exit*, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

Example 8.8: The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9. The entry points to basic block B_1, since B_1 contains the first instruction of the program. The only successor of B_1 is B_2, because B_1 does not end in an unconditional jump, and the leader of B_2 immediately follows the end of B_1.

Block B_3 has two successors. One is itself, because the leader of B_3, instruction 3, is the target of the conditional jump at the end of B_3, instruction 9. The other successor is B_4, because control can fall through the conditional jump at the end of B_3 and next enter the leader of B_4.

Only B_6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B_6. □

https://hemanthrajhemu.github.io

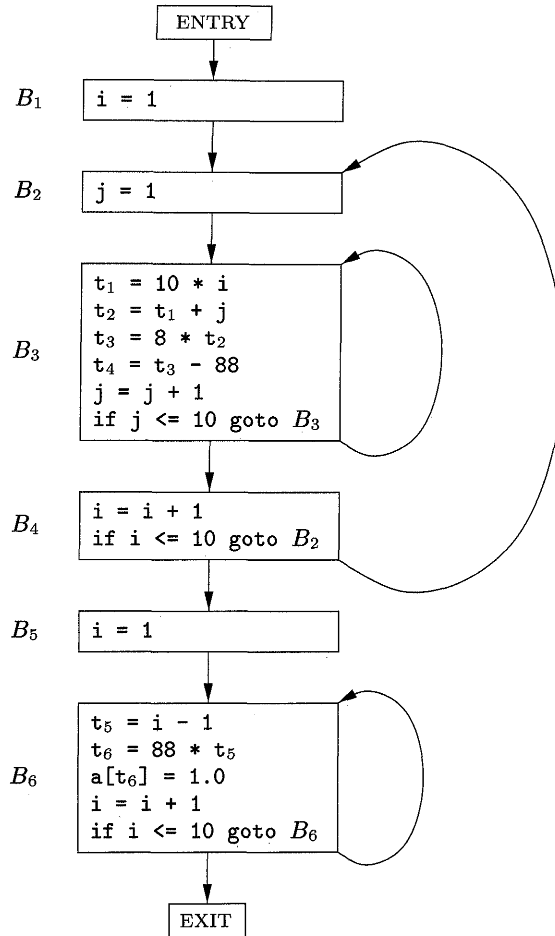


Figure 8.9: Flow graph from Fig. 8.7

8.4.4 Representation of Flow Graphs

First, note from Fig. 8.9 that in the flow graph, it is normal to replace the jumps to instruction numbers or labels by jumps to basic blocks. Recall that every conditional or unconditional jump is to the leader of some basic block, and it is to this block that the jump will now refer. The reason for this change is that after constructing the flow graph, it is common to make substantial changes to the instructions in the various basic blocks. If jumps were to instructions, we would have to fix the targets of the jumps every time one of the target instructions was changed.

Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) need their own representation. We might represent the content of a node by a

https://hemanthrajhemu.github.io

pointer to the leader in the array of three-address instructions, together with a count of the number of instructions or a second pointer to the last instruction. However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

8.4.5 Loops

Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a *loop* if

1. There is a node in L called the *loop entry* with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.

2. Every node in L has a nonempty path, completely within L, to the entry of L.

Example 8.9: The flow graph of Fig. 8.9 has three loops:

1. B_3 by itself.

2. B_6 by itself.

3. $\{B_2, B_3, B_4\}$.

The first two are single nodes with an edge to the node itself. For instance, B_3 forms a loop with B_3 as its entry. Note that the second requirement for a loop is that there be a nonempty path from B_3 to itself. Thus, a single node like B_2, which does not have an edge $B_2 \rightarrow B_2$, is not a loop, since there is no nonempty path from B_2 to itself within $\{B_2\}$.

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1, that is not in L. Further, each of the three nodes has a nonempty path to B_2 staying within L. For instance, B_2 has the path $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$. □

8.4.6 Exercises for Section 8.4

Exercise 8.4.1: Figure 8.10 is a simple matrix-multiplication program.

a) Translate the program into three-address statements of the type we have been using in this section. Assume the matrix entries are numbers that require 8 bytes, and that matrices are stored in row-major order.

https://hemanthrajhemu.github.io

b) Construct the flow graph for your code from (a).

c) Identify the loops in your flow graph from (b).

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        c[i][j] = 0.0;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Figure 8.10: A matrix-multiplication algorithm

Exercise 8.4.2: Figure 8.11 is code to count the number of primes from 2 to n, using the sieve method on a suitably large array a. That is, $a[i]$ is TRUE at the end only if there is no prime $\sqrt{i}$ or less that evenly divides i. We initialize all $a[i]$ to TRUE and then set $a[j]$ to FALSE if we find a divisor of j.

a) Translate the program into three-address statements of the type we have been using in this section. Assume integers require 4 bytes.

b) Construct the flow graph for your code from (a).

c) Identify the loops in your flow graph from (b).

```
for (i=2; i<=n; i++)
    a[i] = TRUE;
count = 0;
s = sqrt(n);
for (i=2; i<=s; i++)
    if (a[i]) /* i has been found to be a prime */ {
        count++;
        for (j=2*i; j<=n; j = j+i)
            a[j] = FALSE; /* no multiple of i is a prime */
    }
```

Figure 8.11: Code to sieve for primes

https://hemanthrajhemu.github.io

8.5 Optimization of Basic Blocks

We can often obtain a substantial improvement in the running time of code merely by performing *local* optimization within each basic block by itself. More thorough *global* optimization, which looks at how information flows among the basic blocks of a program, is covered in later chapters, starting with Chapter 9. It is a complex subject, with many different techniques to consider.

8.5.1 The DAG Representation of Basic Blocks

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph). In Section 6.1.1, we introduced the DAG as a representation for single expressions. The idea extends naturally to the collection of expressions that are created within one basic block. We construct a DAG for a basic blockas follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.

3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.

4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis, discussed in Section 9.2.5.

The DAG representation of a basic block lets us perform several code-improving transformations on the code represented by the block.

a) We can eliminate *local common subexpressions*, that is, instructions that compute a value that has already been computed.

b) We can eliminate *dead code*, that is, instructions that compute a value that is never used.

c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.

d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

https://hemanthrajhemu.github.io

8.5.2 Finding Local Common Subexpressions

Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator. If so, N computes the same value as M and may be used in its place. This technique was introduced as the "value-number" method of detecting common subexpressions in Section 6.1.1.

Example 8.10: A DAG for the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

is shown in Fig. 8.12. When we construct the node for the third statement `c = b + c`, we know that the use of `b` in `b + c` refers to the node of Fig. 8.12 labeled `-`, because that is the most recent definition of `b`. Thus, we do not confuse the values computed at statements one and three.

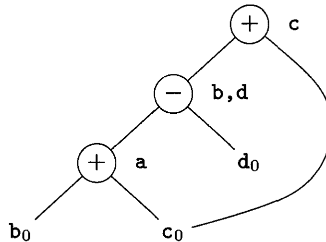


Figure 8.12: DAG for basic block in Example 8.10

However, the node corresponding to the fourth statement `d = a - d` has the operator `-` and the nodes with attached variables `a` and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add `d` to the list of definitions for the node labeled $-$. □

It might appear that, since there are only three nonleaf nodes in the DAG of Fig. 8.12, the basic block in Example 8.10 can be replaced by a block with only three statements. In fact, if `b` is not live on exit from the block, then we do not need to compute that variable, and can use `d` to receive the value represented by the node labeled $-$. in Fig. 8.12. The block then becomes

```
a = b + c
d = a - d
c = d + c
```

https://hemanthrajhemu.github.io

However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.[1]

Example 8.11: When we look for common subexpressions, we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed. Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence

```
a = b + c;
b = b - d
c = c + d
e = b + c
```

is the same, namely $b_0 + c_0$. That is, even though b and c both change between the first and last statements, their sum remains the same, because $b + c = (b - d) + (c + d)$. The DAG for this sequence is shown in Fig. 8.13, but does not exhibit any common subexpressions. However, algebraic identities applied to the DAG, as discussed in Section 8.5.4, may expose the equivalence. □

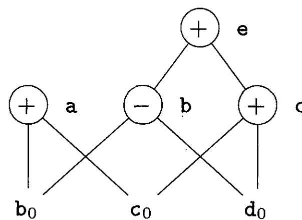


Figure 8.13: DAG for basic block in Example 8.11

8.5.3 Dead Code Elimination

The operation on DAG's that corresponds to dead-code elimination can be implemented as follows. We delete from a DAG any root (node with no ancestors) that has no live variables attached. Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Example 8.12: If, in Fig. 8.13, a and b are live but c and e are not, we can immediately remove the root labeled e. Then, the node labeled c becomes a root and can be removed. The roots labeled a and b remain, since they each have live variables attached. □

[1]In general, we must be careful, when reconstructing code from DAG's, how we choose the names of variables. If a variable x is defined twice, or if it is assigned once and the initial value x_0 is also used, then we must make sure that we do not change the value of x until we have made all uses of the node whose value x previously held.

https://hemanthrajhemu.github.io

8.5.4 The Use of Algebraic Identities

Algebraic identities represent another important class of optimizations on basic blocks. For example, we may apply arithmetic identities, such as

$$x + 0 = 0 + x = x \qquad x - 0 = x$$
$$x \times 1 = 1 \times x = x \qquad x/1 = x$$

to eliminate computations from a basic block.

Another class of algebraic optimizations includes local *reduction in strength*, that is, replacing a more expensive operator by a cheaper one as in:

EXPENSIVE		CHEAPER
x^2	=	$x \times x$
$2 \times x$	=	$x + x$
$x/2$	=	$x \times 0.5$

A third class of related optimizations is *constant folding*. Here we evaluate constant expressions at compile time and replace the constant expressions by their values.[2] Thus the expression $2 * 3.14$ would be replaced by 6.28. Many constant expressions arise in practice because of the frequent use of symbolic constants in programs.

The DAG-construction process can help us apply these and other more general algebraic transformations such as commutativity and associativity. For example, suppose the language reference manual specifies that $*$ is commutative; that is, $x*y = y*x$. Before we create a new node labeled $*$ with left child M and right child N, we always check whether such a node already exists. However, because $*$ is commutative, we should then check for a node having operator $*$, left child N, and right child M.

The relational operators such as $<$ and $=$ sometimes generate unexpected common subexpressions. For example, the condition $x > y$ can also be tested by subtracting the arguments and performing a test on the condition code set by the subtraction.[3] Thus, only one node of the DAG may need to be generated for $x - y$ and $x > y$.

Associative laws might also be applicable to expose common subexpressions. For example, if the source code has the assignments

```
a = b + c;
e = c + d + b;
```

the following intermediate code might be generated:

[2] Arithmetic expressions should be evaluated the same way at compile time as they are at run time. K. Thompson has suggested an elegant solution to constant folding: compile the constant expression, execute the target code on the spot, and replace the expression with the result. Thus, the compiler does not need to contain an interpreter.

[3] The subtraction can, however, introduce overflows and underflows while a compare instruction would not.

https://hemanthrajhemu.github.io

```
a = b + c
t = c + d
e = t + b
```

If `t` is not needed outside this block, we can change this sequence to

```
a = b + c
e = a + d
```

using both the associativity and commutativity of $+$.

The compiler writer should examine the language reference manual carefully to determine what rearrangements of computations are permitted, since (because of possible overflows or underflows) computer arithmetic does not always obey the algebraic identities of mathematics. For example, the Fortran standard states that a compiler may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated. Thus, a compiler may evaluate $x * y - x * z$ as $x * (y - z)$, but it may not evaluate $a + (b - c)$ as $(a + b) - c$. A Fortran compiler must therefore keep track of where parentheses were present in the source language expressions if it is to optimize programs in accordance with the language definition.

8.5.5 Representation of Array References

At first glance, it might appear that the array-indexing instructions can be treated like any other operator. Consider for instance the sequence of three-address statements:

```
x = a[i]
a[j] = y
z = a[i]
```

If we think of `a[i]` as an operation involving `a` and `i`, similar to $a + i$, then it might appear as if the two uses of `a[i]` were a common subexpression. In that case, we might be tempted to "optimize" by replacing the third instruction `z = a[i]` by the simpler `z = x`. However, since `j` could equal `i`, the middle statement may in fact change the value of `a[i]`; thus, it is not legal to make this change.

The proper way to represent array accesses in a DAG is as follows.

1. An assignment from an array, like `x = a[i]`, is represented by creating a node with operator =[] and two children representing the initial value of the array, a_0 in this case, and the index `i`. Variable `x` becomes a label of this new node.

2. An assignment to an array, like `a[j] = y`, is represented by a new node with operator []= and three children representing a_0, `j` and `y`. There is no variable labeling this node. What is different is that the creation of

https://hemanthrajhemu.github.io

this node *kills* all currently constructed nodes whose value depends on a_0. A node that has been killed cannot receive any more labels; that is, it cannot become a common subexpression.

Example 8.13: The DAG for the basic block

```
x = a[i]
a[j] = y
z = a[i]
```

is shown in Fig. 8.14. The node N for x is created first, but when the node labeled []= is created, N is killed. Thus, when the node for z is created, it cannot be identified with N, and a new node with the same operands a_0 and i_0 must be created instead. □

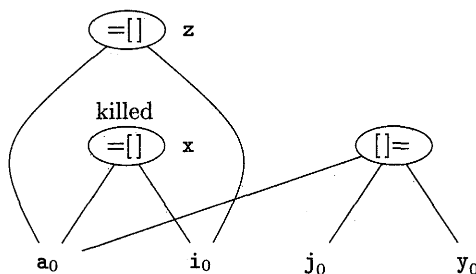


Figure 8.14: The DAG for a sequence of array assignments

Example 8.14: Sometimes, a node must be killed even though none of its children have an array like a_0 in Example 8.13 as attached variable. Likewise, a node can kill if it has a descendant that is an array, even though none of its children are array nodes. For instance, consider the three-address code

```
b = 12 + a
x = b[i]
b[j] = y
```

What is happening here is that, for efficiency reasons, b has been defined to be a position in an array a. For example, if the elements of a are four bytes long, then b represents the fourth element of a. If j and i represent the same value, then b[i] and b[j] represent the same location. Therefore it is important to have the third instruction, b[j] = y, kill the node with x as its attached variable. However, as we see in Fig. 8.15, both the killed node and the node that does the killing have a_0 as a grandchild, not as a child. □

https://hemanthrajhemu.github.io

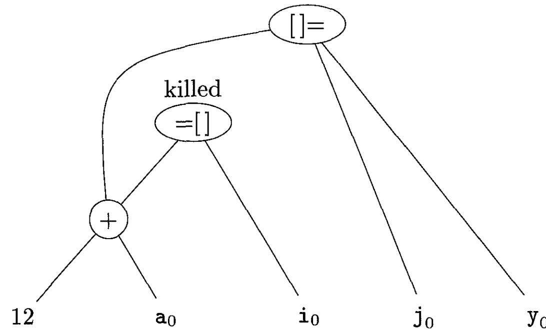


Figure 8.15: A node that kills a use of an array need not have that array as a child

8.5.6 Pointer Assignments and Procedure Calls

When we assign indirectly through a pointer, as in the assignments

```
x = *p
*q = y
```

we do not know what `p` or `q` point to. In effect, `x = *p` is a use of every variable whatsoever, and `*q = y` is a possible assignment to every variable. As a consequence, the operator =* must take all nodes that are currently associated with identifiers as arguments, which is relevant for dead-code elimination. More importantly, the *= operator kills all other nodes so far constructed in the DAG.

There are global pointer analyses one could perform that might limit the set of variables a pointer could reference at a given place in the code. Even local analysis could restrict the scope of a pointer. For instance, in the sequence

```
p = &x
*p = y
```

we know that `x`, and no other variable, is given the value of `y`, so we don't need to kill any node but the node to which `x` was attached.

Procedure calls behave much like assignments through pointers. In the absence of global data-flow information, we must assume that a procedure uses and changes any data to which it has access. Thus, if variable `x` is in the scope of a procedure P, a call to P both uses the node with attached variable `x` and kills that node.

8.5.7 Reassembling Basic Blocks From DAG's

After we perform whatever optimizations are possible while constructing the DAG or by manipulating the DAG once constructed, we may reconstitute the three-address code for the basic block from which we built the DAG. For each

https://hemanthrajhemu.github.io

node that has one or more attached variables, we construct a three-address statement that computes the value of one of those variables. We prefer to compute the result into a variable that is live on exit from the block. However, if we do not have global live-variable information to work from, we need to assume that every variable of the program (but not temporaries that are generated by the compiler to process expressions) is live on exit from the block.

If the node has more than one live variable attached, then we have to introduce copy statements to give the correct value to each of those variables. Sometimes, global optimization can eliminate those copies, if we can arrange to use one of two variables in place of the other.

Example 8.15: Recall the DAG of Fig. 8.12. In the discussion following Example 8.10, we decided that if `b` is not live on exit from the block, then the three statements

```
a = b + c
d = a - d
c = d + c
```

suffice to reconstruct the basic block. The third instruction, `c = d + c`, must use `d` as an operand rather than `b`, because the optimized block never computes `b`.

If both `b` and `d` are live on exit, or if we are not sure whether or not they are live on exit, then we need to compute `b` as well as `d`. We can do so with the sequence

```
a = b + c
d = a - d
b = d
c = d + c
```

This basic block is still more efficient than the original. Although the number of instructions is the same, we have replaced a subtraction by a copy, which tends to be less expensive on most machines. Further, it may be that by doing a global analysis, we can eliminate the use of this computation of `b` outside the block by replacing it by uses of `d`. In that case, we can come back to this basic block and eliminate `b = d` later. Intuitively, we can eliminate this copy if wherever this value of `b` is used, `d` is still holding the same value. That situation may or may not be true, depending on how the program recomputes `d`. □

When reconstructing the basic block from a DAG, we not only need to worry about what variables are used to hold the values of the DAG's nodes, but we also need to worry about the order in which we list the instructions computing the values of the various nodes. The rules to remember are

1. The order of instructions must respect the order of nodes in the DAG. That is, we cannot compute a node's value until we have computed a value for each of its children.

https://hemanthrajhemu.github.io

2. Assignments to an array must follow all previous assignments to, or evaluations from, the same array, according to the order of these instructions in the original basic block.

3. Evaluations of array elements must follow any previous (according to the original block) assignments to the same array. The only permutation allowed is that two evaluations from the same array may be done in either order, as long as neither crosses over an assignment to that array.

4. Any use of a variable must follow all previous (according to the original block) procedure calls or indirect assignments through a pointer.

5. Any procedure call or indirect assignment through a pointer must follow all previous (according to the original block) evaluations of any variable.

That is, when reordering code, no statement may cross a procedure call or assignment through a pointer, and uses of the same array may cross each other only if both are array accesses, but not assignments to elements of the array.

8.5.8 Exercises for Section 8.5

Exercise 8.5.1: Construct the DAG for the basic block

```
d = b * c
e = a + b
b = b * c
a = e - d
```

Exercise 8.5.2: Simplify the three-address code of Exercise 8.5.1, assuming

a) Only a is live on exit from the block.

b) a, b, and c are live on exit from the block.

Exercise 8.5.3: Construct the basic block for the code in block B_6 of Fig. 8.9. Do not forget to include the comparison $i \leq 10$.

Exercise 8.5.4: Construct the basic block for the code in block B_3 of Fig. 8.9.

Exercise 8.5.5: Extend Algorithm 8.7 to process three-statements of the form

a) `a[i] = b`

b) `a = b[i]`

c) `a = *b`

c) `*a = b`

Exercise 8.5.6: Construct the DAG for the basic block

https://hemanthrajhemu.github.io

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

on the assumption that

a) `p` can point anywhere.

b) `p` can point only to `b` or `d`.

! **Exercise 8.5.7:** If a pointer or array expression, such as `a[i]` or `*p` is assigned and then used, without the possibility of being changed in the interim, we can take advantage of the situation to simplify the DAG. For example, in the code of Exercise 8.5.6, since `p` is not assigned between the second and fourth statements, the statement `e = *p` can be replaced by `e = c`, regardless of what `p` points to. Revise the DAG-construction algorithm to take advantage of such situations, and apply your algorithm to the code of Example 8.5.6.

Exercise 8.5.8: Suppose a basic block is formed from the C assignment statements

```
x = a + b + c + d + e + f;
y = a + c + e;
```

a) Give the three-address statements (only one addition per statement) for this block.

b) Use the associative and commutative laws to modify the block to use the fewest possible number of instructions, assuming both `x` and `y` are live on exit from the block.

8.6 A Simple Code Generator

In this section, we shall consider an algorithm that generates code for a single basic block. It considers each three-address instruction in turn, and keeps track of what values are in what registers so it can avoid generating unnecessary loads and stores.

One of the primary issues during code generation is deciding how to use registers to best advantage. There are four principal uses of registers:

- In most machine architectures, some or all of the operands of an operation must be in registers in order to perform the operation.
- Registers make good temporaries — places to hold the result of a subexpression while a larger expression is being evaluated, or more generally, a place to hold a variable that is used only within a single basic block.

https://hemanthrajhemu.github.io

- Registers are used to hold (*global*) values that are computed in one basic block and used in other blocks, for example, a loop index that is incremented going around the loop and is used several times within the loop.

- Registers are often used to help with run-time storage management, for example, to manage the run-time stack, including the maintenance of stack pointers and possibly the top elements of the stack itself.

These are competing needs, since the number of registers available is limited.

The algorithm in this section assumes that some set of registers is available to hold the values that are used within the block. Typically, this set of registers does not include all the registers of the machine, since some registers are reserved for global variables and managing the stack. We assume that the basic block has already been transformed into a preferred sequence of three-address instructions, by transformations such as combining common subexpressions. We further assume that for each operator, there is exactly one machine instruction that takes the necessary operands in registers and performs that operation, leaving the result in a register. The machine instructions are of the form

- `LD` *reg, mem*

- `ST` *mem, reg*

- *OP reg, reg, reg*

8.6.1 Register and Address Descriptors

Our code-generation algorithm considers each three-address instruction in turn and decides what loads are necessary to get the needed operands into registers. After generating the loads, it generates the operation itself. Then, if there is a need to store the result into a memory location, it also generates that store.

In order to make the needed decisions, we require a data structure that tells us what program variables currently have their value in a register, and which register or registers, if so. We also need to know whether the memory location for a given variable currently has the proper value for that variable, since a new value for the variable may have been computed in a register and not yet stored. The desired data structure has the following descriptors:

1. For each available register, a *register descriptor* keeps track of the variable names whose current value is in that register. Since we shall use only those registers that are available for local use within a basic block, we assume that initially, all register descriptors are empty. As the code generation progresses, each register will hold the value of zero or more names.

2. For each program variable, an *address descriptor* keeps track of the location or locations where the current value of that variable can be found. The location might be a register, a memory address, a stack location, or some set of more than one of these. The information can be stored in the symbol-table entry for that variable name.

https://hemanthrajhemu.github.io

8.6.2 The Code-Generation Algorithm

An essential part of the algorithm is a function *getReg*(I), which selects registers for each memory location associated with the three-address instruction I. Function *getReg* has access to the register and address descriptors for all the variables of the basic block, and may also have access to certain useful data-flow information such as the variables that are live on exit from the block. We shall discuss *getReg* after presenting the basic algorithm. While we do not know the total number of registers available for local data belonging to a basic block, we assume that there are enough registers so that, after freeing all available registers by storing their values in memory, there are enough registers to accomplish any three-address operation.

In a three-address instruction such as `x = y + z`, we shall treat `+` as a generic operator and `ADD` as the equivalent machine instruction. We do not, therefore, take advantage of commutativity of `+`. Thus, when we implement the operation, the value of `y` must be in the second register mentioned in the `ADD` instruction, never the third. A possible improvement to the algorithm is to generate code for both `x = y + z` and `x = z + y` whenever `+` is a commutative operator, and pick the better code sequence.

Machine Instructions for Operations

For a three-address instruction such as $x = y + z$, do the following:

1. Use *getReg*($x = y + z$) to select registers for x, y, and z. Call these R_x, R_y, and R_z.

2. If y is not in R_y (according to the register descriptor for R_y), then issue an instruction `LD` R_y, y', where y' is one of the memory locations for y (according to the address descriptor for y).

3. Similarly, if z is not in R_z, issue and instruction `LD` R_z, z', where z' is a location for z.

4. Issue the instruction `ADD` R_x, R_y, R_z.

Machine Instructions for Copy Statements

There is an important special case: a three-address copy statement of the form $x = y$. We assume that *getReg* will always choose the same register for both x and y. If y is not already in that register R_y, then generate the machine instruction `LD` R_y, `y`. If y was already in R_y, we do nothing. It is only necessary that we adjust the register description for R_y so that it includes x as one of the values found there.

https://hemanthrajhemu.github.io

Ending the Basic Block

As we have described the algorithm, variables used by the block may wind up with their only location being a register. If the variable is a temporary used only within the block, that is fine; when the block ends, we can forget about the value of the temporary and assume its register is empty. However, if the variable is live on exit from the block, or if we don't know which variables are live on exit, then we need to assume that the value of the variable is needed later. In that case, for each variable x whose location descriptor does not say that its value is located in the memory location for x, we must generate the instruction `ST` x, R, where R is a register in which x's value exists at the end of the block.

Managing Register and Address Descriptors

As the code-generation algorithm issues load, store, and other machine instructions, it needs to update the register and address descriptors. The rules are as follows:

1. For the instruction `LD` R, x
   (a) Change the register descriptor for register R so it holds only x.
   (b) Change the address descriptor for x by adding register R as an additional location.
2. For the instruction `ST` x, R, change the address descriptor for x to include its own memory location.
3. For an operation such as `ADD` R_x, R_y, R_z implementing a three-address instruction $x = y + z$
   (a) Change the register descriptor for R_x so that it holds only x.
   (b) Change the address descriptor for x so that its only location is R_x. Note that the memory location for x is *not* now in the address descriptor for x.
   (c) Remove R_x from the address descriptor of any variable other than x.
4. When we process a copy statement $x = y$, after generating the load for y into register R_y, if needed, and after managing descriptors as for all load statements (per rule 1):
   (a) Add x to the register descriptor for R_y.
   (b) Change the address descriptor for x so that its only location is R_y.

Example 8.16 : Let us translate the basic block consisting of the three-address statements

https://hemanthrajhemu.github.io

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

Here we assume that t, u, and v are temporaries, local to the block, while a, b, c, and d are variables that are live on exit from the block. Since we have not yet discussed how the function *getReg* might work, we shall simply assume that there are as many registers as we need, but that when a register's value is no longer needed (for example, it holds only a temporary, all of whose uses have been passed), then we reuse its register.

A summary of all the machine-code instructions generated is in Fig. 8.16. The figure also shows the register and address descriptors before and after the translation of each three-address instruction.

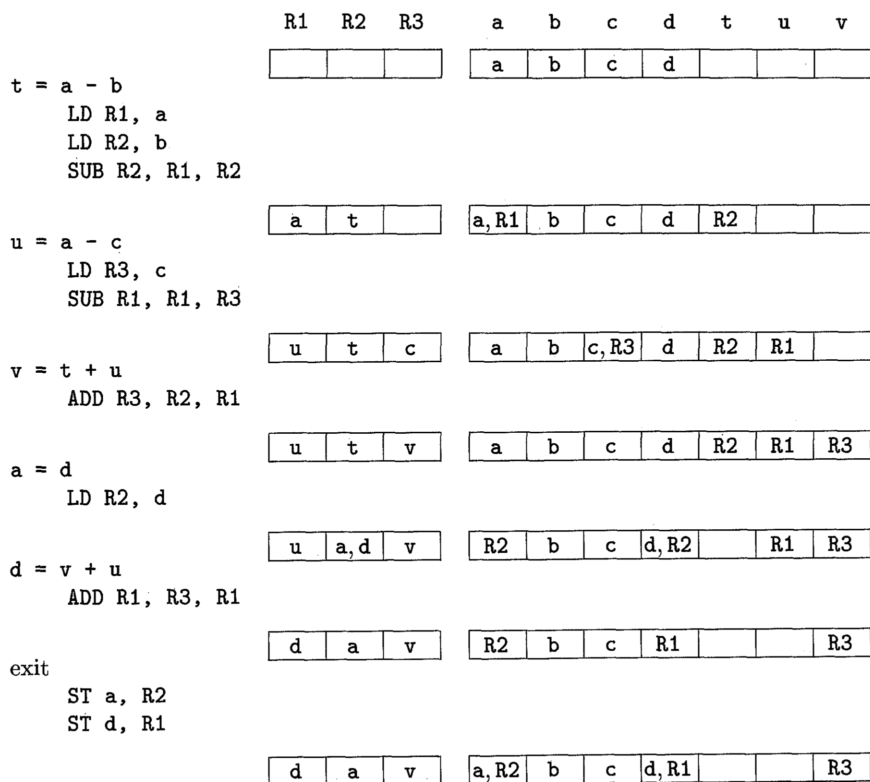


Figure 8.16: Instructions generated and the changes in the register and address descriptors

For the first three-address instruction, t = a - b we need to issue three instructions, since nothing is in a register initially. Thus, we see a and b loaded

https://hemanthrajhemu.github.io

into registers R1 and R2, and the value t produced in register R2. Notice that we can use R2 for t because the value b previously in R2 is not needed within the block. Since b is presumably live on exit from the block, had it not been in its own memory location (as indicated by its address descriptor), we would have had to store R2 into b first. The decision to do so, had we needed R2, would be taken by *getReg*.

The second instruction, u = a - c, does not require a load of a, since it is already in register R1. Further, we can reuse R1 for the result, u, since the value of a, previously in that register, is no longer needed within the block, and its value is in its own memory location if a is needed outside the block. Note that we change the address descriptor for a to indicate that it is no longer in R1, but is in the memory location called a.

The third instruction, v = t + u, requires only the addition. Further, we can use R3 for the result, v, since the value of c in that register is no longer needed within the block, and c has its value in its own memory location.

The copy instruction, a = d, requires a load of d, since it is not in memory. We show register R2's descriptor holding both a and d. The addition of a to the register descriptor is the result of our processing the copy statement, and is not the result of any machine instruction.

The fifth instruction, d = v + u, uses two values that are in registers. Since u is a temporary whose value is no longer needed, we have chosen to reuse its register R1 for the new value of d. Notice that d is now in only R1, and is not in its own memory location. The same holds for a, which is in R2 and not in the memory location called a. As a result, we need a "coda" to the machine code for the basic block that stores the live-on-exit variables a and d into their memory locations. We show these as the last two instructions. □

8.6.3 Design of the Function *getReg*

Lastly, let us consider how to implement *getReg*(I), for a three-address instruction I. There are many options, although there are also some absolute prohibitions against choices that lead to incorrect code due to the loss of the value of one or more live variables. We begin our examination with the case of an operation step, for which we again use $x = y + z$ as the generic example. First, we must pick a register for y and a register for z. The issues are the same, so we shall concentrate on picking register R_y for y. The rules are as follows:

1. If y is currently in a register, pick a register already containing y as R_y. Do not issue a machine instruction to load this register, as none is needed.

2. If y is not in a register, but there is a register that is currently empty, pick one such register as R_y.

3. The difficult case occurs when y is not in a register, and there is no register that is currently empty. We need to pick one of the allowable registers anyway, and we need to make it safe to reuse. Let R be a candidate

https://hemanthrajhemu.github.io

register, and suppose v is one of the variables that the register descriptor for R says is in R. We need to make sure that v's value either is not really needed, or that there is somewhere else we can go to get the value of R. The possibilities are:

(a) If the address descriptor for v says that v is somewhere besides R, then we are OK.

(b) If v is x, the value being computed by instruction I, and x is not also one of the other operands of instruction I (z in this example), then we are OK. The reason is that in this case, we know this value of x is never again going to be used, so we are free to ignore it.

(c) Otherwise, if v is not used later (that is, after the instruction I, there are no further uses of v, and if v is live on exit from the block, then v is recomputed within the block), then we are OK.

(d) If we are not OK by one of the first two cases, then we need to generate the store instruction `ST` v, R to place a copy of v in its own memory location. This operation is called a *spill.*

Since R may hold several variables at the moment, we repeat the above steps for each such variable v. At the end, R's "score" is the number of store instructions we needed to generate. Pick one of the registers with the lowest score.

Now, consider the selection of the register R_x. The issues and options are almost as for y, so we shall only mention the differences.

1. Since a new value of x is being computed, a register that holds only x is always an acceptable choice for R_x. This statement holds even if x is one of y and z, since our machine instructions allows two registers to be the same in one instruction.

2. If y is not used after instruction I, in the sense described for variable v in item (3c), and R_y holds only y after being loaded, if necessary, then R_y can also be used as R_x. A similar option holds regarding z and R_z.

The last matter to consider specially is the case when I is a copy instruction $x = y$. We pick the register R_y as above. Then, we always choose $R_x = R_y$.

8.6.4 Exercises for Section 8.6

Exercise 8.6.1: For each of the following C assignment statements

a) `x = a + b*c;`

b) `x = a/(b+c) - d*(e+f);`

c) `x = a[i] + 1;`

https://hemanthrajhemu.github.io

d) `a[i] = b[c[i]];`

e) `a[i][j] = b[i][k] + c[k][j];`

f) `*p++ = *q++;`

generate three-address code, assuming that all array elements are integers taking four bytes each. In parts (d) and (e), assume that `a`, `b`, and `c` are constants giving the location of the first (0th) elements of the arrays with those names, as in all previous examples of array accesses in this chapter.

! **Exercise 8.6.2:** Repeat Exercise 8.6.1 parts (d) and (e), assuming that the arrays `a`, `b`, and `c` are located via pointers, `pa`, `pb`, and `pc`, respectively, pointing to the locations of their respective first elements.

Exercise 8.6.3: Convert your three-address code from Exercise 8.6.1 into machine code for the machine model of this section. You may use as many registers as you need.

Exercise 8.6.4: Convert your three-address code from Exercise 8.6.1 into machine code, using the simple code-generation algorithm of this section, assuming three registers are available. Show the register and address descriptors after each step.

Exercise 8.6.5: Repeat Exercise 8.6.4, but assuming only two registers are available.

8.7 Peephole Optimization

While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure. Nevertheless, many simple transformations can significantly improve the running time or space requirement of the target program.

A simple but effective technique for locally improving the target code is *peephole optimization*, which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

The peephole is a small, sliding window on a program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may

https://hemanthrajhemu.github.io

spawn opportunities for additional improvements. In general, repeated passes over the target code are necessary to get the maximum benefit. In this section, we shall give the following examples of program transformations that are characteristic of peephole optimizations:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

8.7.1 Eliminating Redundant Loads and Stores

If we see the instruction sequence

```
LD a, R0
ST R0, a
```

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register R0. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe.

Redundant loads and stores of this nature would not be generated by the simple code generation algorithm of the previous section. However, a naive code generation algorithm like the one in Section 8.1.3 would generate redundant sequences such as these.

8.7.2 Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1. In the intermediate representation, this code may look like

```
    if debug == 1 goto L1
    goto L2
L1: print debugging information
L2:
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus, no matter what the value of debug, the code sequence above can be replaced by

https://hemanthrajhemu.github.io

```
     if debug != 1 goto L2
     print debugging information
L2:
```

If debug is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```
     if 0 != 1 goto L2
     print debugging information
L2:
```

Now the argument of the first statement always evaluates to *true*, so the statement can be replaced by goto L2. Then all statements that print debugging information are unreachable and can be eliminated one at a time.

8.7.3 Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
     goto L1
     ...
L1:  goto L2
```

by the sequence

```
     goto L2
     ...
L1:  goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement L1: goto L2 provided it is preceded by an unconditional jump.

Similarly, the sequence

```
     if a < b goto L1
     ...
L1:  goto L2
```

can be replaced by the sequence

```
     if a < b goto L2
     ...
L1:  goto L2
```

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

https://hemanthrajhemu.github.io

```
     goto L1
     . . .
L1:  if a < b goto L2
L3:
```

may be replaced by the sequence

```
     if a < b goto L2
     goto L3
     . . .
L3:
```

While the number of instructions in the two sequences is the same, we sometimes skip the unconditional jump in the second sequence, but never in the first. Thus, the second sequence is superior to the first in execution time.

8.7.4 Algebraic Simplification and Reduction in Strength

In Section 8.5 we discussed algebraic identities that could be used to simplify DAG's. These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

```
x = x + 0
```

or

```
x = x * 1
```

in the peephole.

Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be approximated as multiplication by a constant, which may be cheaper.

8.7.5 Use of Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like `x = x + 1`.

https://hemanthrajhemu.github.io

8.7.6 Exercises for Section 8.7

Exercise 8.7.1: Construct an algorithm that will perform redundant-instruction elimination in a sliding peephole on target machine code.

Exercise 8.7.2: Construct an algorithm that will do flow-of-control optimizations in a sliding peephole on target machine code.

Exercise 8.7.3: Construct an algorithm that will do simple algebraic simplifications and reductions in strength in a sliding peephole on target machine code.

8.8 Register Allocation and Assignment

Instructions involving only register operands are faster than those involving memory operands. On modern machines, processor speeds are often an order of magnitude or more faster than memory speeds. Therefore, efficient utilization of registers is vitally important in generating good code. This section presents various strategies for deciding at each point in a program what values should reside in registers (register allocation) and in which register each value should reside (register assignment).

One approach to register allocation and assignment is to assign specific values in the target program to certain registers. For example, we could decide to assign base addresses to one group of registers, arithmetic computations to another, the top of the stack to a fixed register, and so on.

This approach has the advantage that it simplifies the design of a code generator. Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated into the other registers. Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers, and the like, and to allow the remaining registers to be used by the code generator as it sees fit.

8.8.1 Global Register Allocation

The code generation algorithm in Section 8.6 used registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block. To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (*globally*). Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop. For the time being, assume that we know the loop structure of a flow graph, and that we know what values computed in a basic block are used outside that block. The next chapter covers techniques for computing this information.

https://hemanthrajhemu.github.io

One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop. The selected values may be different in different loops. Registers not already allocated may be used to hold values local to one block as in Section 8.6. This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation. Yet the method is simple to implement and was used in Fortran H, the optimizing Fortran compiler developed by IBM for the 360-series machines in the late 1960s.

With early C compilers, a programmer could do some register allocation explicitly by using register declarations to keep certain values in registers for the duration of a procedure. Judicious use of register declarations did speed up many programs, but programmers were encouraged to first profile their programs to determine the program's hotspots before doing their own register allocation.

8.8.2 Usage Counts

In this section we shall assume that the savings to be realized by keeping a variable x in a register for the duration of a loop L is one unit of cost for each reference to x if x is already in a register. However, if we use the approach in Section 8.6 to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block. Thus we count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block. We also save two units if we can avoid a store of x at the end of a block. Thus, if x is allocated a register, we count a savings of two for each block in loop L for which x is live on exit and in which x is assigned a value.

On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units. Similarly, for each exit block B of loop L at which x is live on entry to some successor of B outside of L, we must store x at a cost of two. However, on the assumption that the loop is iterated many times, we may neglect these debits since they occur only once each time we enter the loop. Thus, an approximate formula for the benefit to be realized from allocating a register x within loop L is

$$\sum_{\text{blocks } B \text{ in } L} use(x, B) + 2 * live(x, B) \tag{8.1}$$

where $use(x, B)$ is the number of times x is used in B prior to any definition of x; $live(x, B)$ is 1 if x is live on exit from B and is assigned a value in B, and $live(x, B)$ is 0 otherwise. Note that (8.1) is approximate, because not all blocks in a loop are executed with equal frequency and also because (8.1) is based on the assumption that a loop is iterated many times. On specific machines a formula analogous to (8.1), but possibly quite different from it, would have to be developed.

https://hemanthrajhemu.github.io

Example 8.17: Consider the the basic blocks in the inner loop depicted in Fig. 8.17, where jump and conditional jump statements have been omitted. Assume registers `R0`, `R1`, and `R2` are allocated to hold values throughout the loop. Variables live on entry into and on exit from each block are shown in Fig. 8.17 for convenience, immediately above and below each block, respectively. There are some subtle points about live variables that we address in the next chapter. For example, notice that both `e` and `f` are live at the end of B_1, but of these, only `e` is live on entry to B_2 and only `f` on entry to B_3. In general, the variables live at the end of a block are the union of those live at the beginning of each of its successor blocks.

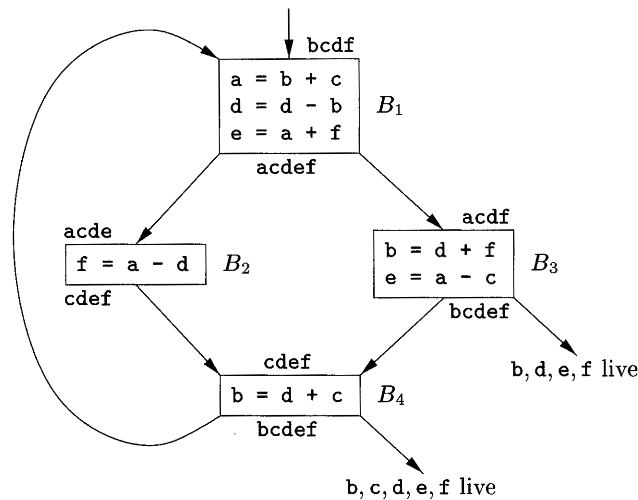


Figure 8.17: Flow graph of an inner loop

To evaluate (8.1) for $x = $ `a`, we observe that `a` is live on exit from B_1 and is assigned a value there, but is not live on exit from B_2, B_3, or B_4. Thus, $\sum_{B \text{ in } L} use(\texttt{a}, B) = 2$. Hence the value of (8.1) for $x = $ `a` is 4. That is, four units of cost can be saved by selecting `a` for one of the global registers. The values of (8.1) for `b`, `c`, `d`, `e`, and `f` are 5, 3, 6, 4, and 4, respectively. Thus, we may select `a`, `b`, and `d` for registers `R0`, `R1`, and `R2`, respectively. Using `R0` for `e` or `f` instead of `a` would be another choice with the same apparent benefit. Figure 8.18 shows the assembly code generated from Fig. 8.17, assuming that the strategy of Section 8.6 is used to generate code for each block. We do not show the generated code for the omitted conditional or unconditional jumps that end each block in Fig. 8.17, and we therefore do not show the generated code as a single stream as it would appear in practice. □

https://hemanthrajhemu.github.io

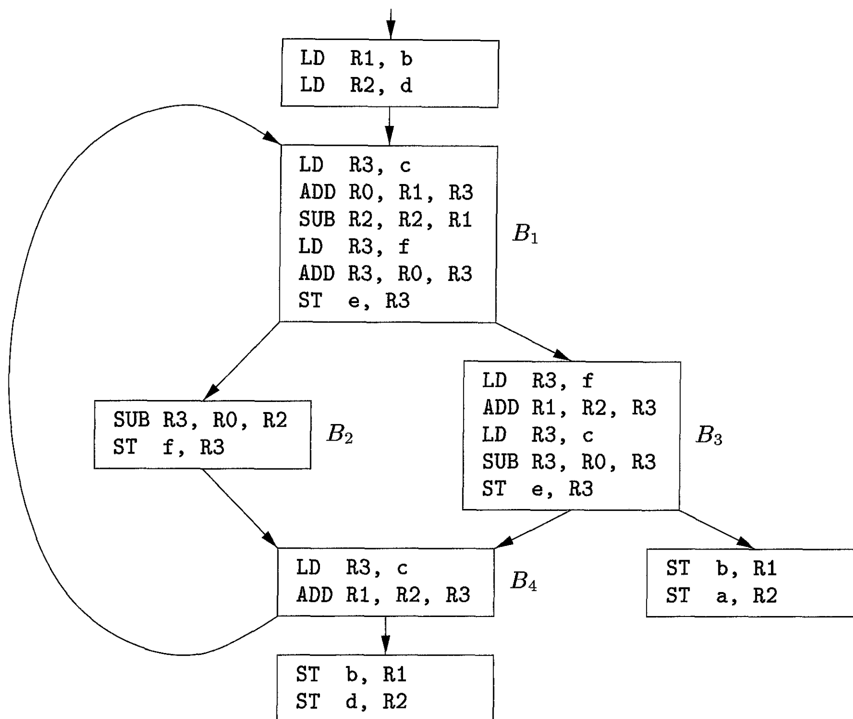


Figure 8.18: Code sequence using global register assignment

8.8.3 Register Assignment for Outer Loops

Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger enclosing loops. If an outer loop L_1 contains an inner loop L_2, the names allocated registers in L_2 need not be allocated registers in $L_1 - L_2$. Similarly, if we choose to allocate x a register in L_2 but not L_1, we must load x on entrance to L_2 and store x on exit from L_2. We leave as an exercise the derivation of a criterion for selecting names to be allocated registers in an outer loop L, given that choices have already been made for all loops nested within L.

8.8.4 Register Allocation by Graph Coloring

When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (*spilled*) into a memory location in order to free up a register. Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

In the method, two passes are used. In the first, target-machine instructions are selected as though there are an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and

https://hemanthrajhemu.github.io

the three-address instructions become machine-language instructions. If access to variables requires instructions that use stack pointers, display pointers, base registers, or other quantities that assist access, then we assume that these quantities are held in registers reserved for each purpose. Normally, their use is directly translatable into an access mode for an address mentioned in a machine instruction. If access is more complex, the access must be broken into several machine instructions, and a temporary symbolic register (or several) may need to be created.

Once the instructions have been selected, a second pass assigns physical registers to symbolic ones. The goal is to find an assignment that minimizes the cost of spills.

In the second pass, for each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined. For example, a register-interference graph for Fig. 8.17 would have nodes for names `a` and `d`. In block B_1, `a` is live at the second statement, which defines `d`; therefore, in the graph there would be an edge between the nodes for `a` and `d`.

An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers. A graph is said to be *colored* if each node has been assigned a color in such a way that no two adjacent nodes have the same color. A color represents a register, and the color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.

Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice. Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G'. A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.

By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes. In the latter case a k-coloring is no longer possible. At this point a node is spilled by introducing code to store and reload the register. Chaitin has devised several heuristics for choosing the node to spill. A general rule is to avoid introducing spill code into inner loops.

8.8.5 Exercises for Section 8.8

Exercise 8.8.1: Construct the register-interference graph for the program in Fig. 8.17.

Exercise 8.8.2: Devise a register-allocation strategy on the assumption that we automatically store all registers on the stack before each procedure call and restore them after the return.

https://hemanthrajhemu.github.io

8.9 Instruction Selection by Tree Rewriting

Instruction selection can be a large combinatorial task, especially on machines that are rich in addressing modes, such as CISC machines, or on machines with special-purpose instructions, say, for signal processing. Even if we assume that the order of evaluation is given and that registers are allocated by a separate mechanism, instruction selection — the problem of selecting target-language instructions to implement the operators in the intermediate representation — remains a large combinatorial task.

In this section, we treat instruction selection as a tree-rewriting problem. Tree representations of target instructions have been used effectively in code-generator generators, which automatically construct the instruction-selection phase of a code generator from a high-level specification of the target machine. Better code might be obtained for some machines by using DAG's rather than trees, but DAG matching is more complex than tree matching.

8.9.1 Tree-Translation Schemes

Throughout this section, the input to the code-generation process will be a sequence of trees at the semantic level of the target machine. The trees are what we might get after inserting run-time addresses into the intermediate representation, as described in Section 8.3. In addition, the leaves of the trees contain information about the storage types of their labels.

Example 8.18: Figure 8.19 contains a tree for the assignment statement `a[i] = b + 1`, where the array `a` is stored on the run-time stack and the variable `b` is a global in memory location M_b. The run-time addresses of locals `a` and `i` are given as constant offsets C_a and C_i from `SP`, the register containing the pointer to the beginning of the current activation record.

The assignment to `a[i]` is an indirect assignment in which the r-value of the location for `a[i]` is set to the r-value of the expression `b + 1`. The addresses of array `a` and variable `i` are given by adding the values of the constant C_a and C_i, respectively, to the contents of register `SP`. We simplify array-address calculations by assuming that all values are one-byte characters. (Some instruction sets make special provisions for multiplications by constants, such as 2, 4, and 8, during address calculations.)

In the tree, the **ind** operator treats its argument as a memory address. As the left child of an assignment operator, the **ind** node gives the location into which the r-value on the right side of the assignment operator is to be stored. If an argument of a + or **ind** operator is a memory location or a register, then the contents of that memory location or register are taken as the value. The leaves in the tree are labeled with attributes; a subscript indicates the value of the attribute. □

The target code is generated by applying a sequence of tree-rewriting rules to reduce the input tree to a single node. Each tree-rewriting rule has the form

https://hemanthrajhemu.github.io

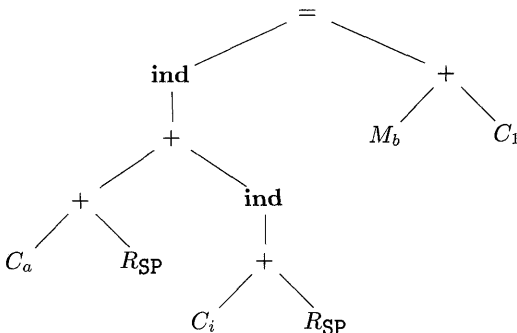


Figure 8.19: Intermediate-code tree for `a[i] = b + 1`

$$\textit{replacement} \leftarrow \textit{template} \ \{ \ \textit{action} \ \}$$

where *replacement* is a single node, *template* is a tree, and *action* is a code fragment, as in a syntax-directed translation scheme.

A set of tree-rewriting rules is called a *tree-translation scheme.*

Each tree-rewriting rule represents the translation of a portion of the tree given by the template. The translation consists of a possibly empty sequence of machine instructions that is emitted by the action associated with the template. The leaves of the template are attributes with subscripts, as in the input tree. Sometimes, certain restrictions apply to the values of the subscripts in the templates; these restrictions are specified as semantic predicates that must be satisfied before the template is said to match. For example, a predicate might specify that the value of a constant fall in a certain range.

A tree-translation scheme is a convenient way to represent the instruction-selection phase of a code generator. As an example of a tree-rewriting rule, consider the rule for the register-to-register add instruction:

$$R_i \leftarrow \underset{R_i \quad R_j}{+} \qquad \{ \ \texttt{ADD R}i\texttt{, R}i\texttt{, R}j \ \}$$

This rule is used as follows. If the input tree contains a subtree that matches this tree template, that is, a subtree whose root is labeled by the operator + and whose left and right children are quantities in registers i and j, then we can replace that subtree by a single node labeled R_i and emit the instruction `ADD Ri, Ri, Rj` as output. We call this replacement a *tiling* of the subtree. More than one template may match a subtree at a given time; we shall describe shortly some mechanisms for deciding which rule to apply in cases of conflict.

Example 8.19: Figure 8.20 contains tree-rewriting rules for a few instructions of our target machine. These rules will be used in a running example throughout this section. The first two rules correspond to load instructions, the next two

https://hemanthrajhemu.github.io

to store instructions, and the remainder to indexed loads and additions. Note that rule (8) requires the value of the constant to be 1. This condition would be specified by a semantic predicate. □

8.9.2 Code Generation by Tiling an Input Tree

A tree-translation scheme works as follows. Given an input tree, the templates in the tree-rewriting rules are applied to tile its subtrees. If a template matches, the matching subtree in the input tree is replaced with the replacement node of the rule and the action associated with the rule is done. If the action contains a sequence of machine instructions, the instructions are emitted. This process is repeated until the tree is reduced to a single node, or until no more templates match. The sequence of machine instructions generated as the input tree is reduced to a single node constitutes the output of the tree-translation scheme on the given input tree.

The process of specifying a code generator becomes similar to that of using a syntax-directed translation scheme to specify a translator. We write a tree-translation scheme to describe the instruction set of a target machine. In practice, we would like to find a scheme that causes a minimal-cost instruction sequence to be generated for each input tree. Several tools are available to help build a code generator automatically from a tree-translation scheme.

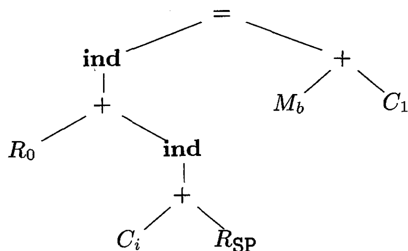
Example 8.20: Let us use the tree-translation scheme in Fig. 8.20 to generate code for the input tree in Fig. 8.19. Suppose that the first rule is applied to load the constant C_a into register `R0`:

1) $R_0 \leftarrow C_a$ { `LD R0, #a` }

The label of the leftmost leaf then changes from C_a to R_0 and the instruction `LD R0,#a` is generated. The seventh rule now matches the leftmost subtree with root labeled +:

7) $R_0 \leftarrow$ + (with children R_0 and R_{SP}) { `ADD R0, R0, SP` }

Using this rule, we rewrite this subtree as a single node labeled R_0 and generate the instruction `ADD R0,R0,SP`. Now the tree looks like



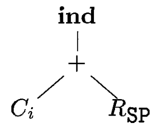
https://hemanthrajhemu.github.io

1)	$R_i \leftarrow C_a$	{ LD R*i*, #*a* }
2)	$R_i \leftarrow M_x$	{ LD R*i*, *x* }
3)	$M \leftarrow$ = M_x R_i	{ ST *x*, R*i* }
4)	$M \leftarrow$ = **ind** R_j R_i	{ ST *R*i*, R*j* }
5)	$R_i \leftarrow$ **ind** + C_a R_j	{ LD R*i*, *a*(R*j*) }
6)	$R_i \leftarrow$ + R_i **ind** + C_a R_j	{ ADD R*i*, R*i*, *a*(R*j*) }
7)	$R_i \leftarrow$ + R_i R_j	{ ADD R*i*, R*i*, R*j* }
8)	$R_i \leftarrow$ + R_i C_1	{ INC R*i* }

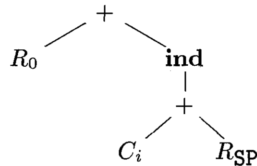
Figure 8.20: Tree-rewriting rules for some target-machine instructions

https://hemanthrajhemu.github.io

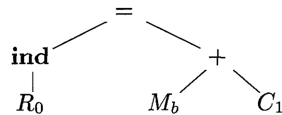
At this point, we could apply rule (5) to reduce the subtree



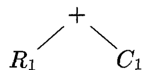
to a single node labeled, say, R_1. We could also use rule (6) to reduce the larger subtree



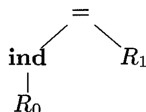
to a single node labeled R_0 and generate the instruction `ADD R0, R0, i(SP)`. Assuming that it is more efficient to use a single instruction to compute the larger subtree rather than the smaller one, we choose rule (6) to get



In the right subtree, rule (2) applies to the leaf M_b. It generates an instruction to load `b` into register `R1`, say. Now, using rule (8) we can match the subtree



and generate the increment instruction `INC R1`. At this point, the input tree has been reduced to



This remaining tree is matched by rule (4), which reduces the tree to a single node and generates the instruction `ST *R0, R1`. We generate the following code sequence:

```
LD  R0, #a
ADD R0, R0, SP
ADD R0, R0, i(SP)
LD  R1, b
INC R1
ST  *R0, R1
```

https://hemanthrajhemu.github.io

in the process of reducing the tree to a single node. □

In order to implement the tree-reduction process in Example 8.18, we must address some issues related to tree-pattern matching:

- How is tree-pattern matching to be done? The efficiency of the code-generation process (at compile time) depends on the efficiency of the tree-matching algorithm.
- What do we do if more than one template matches at a given time? The efficiency of the generated code (at run time) may depend on the order in which templates are matched, since different match sequences will in general lead to different target-machine code sequences, some more efficient than others.

If no template matches, then the code-generation process blocks. At the other extreme, we need to guard against the possibility of a single node being rewritten indefinitely, generating an infinite sequence of register move instructions or an infinite sequence of loads and stores.

To prevent blocking, we assume that each operator in the intermediate code can be implemented by one or more target-machine instructions. We further assume that there are enough registers to compute each tree node by itself. Then, no matter how the tree matching proceeds, the remaining tree can always be translated into target-machine instructions.

8.9.3 Pattern Matching by Parsing

Before considering general tree matching, we consider a specialized approach that uses an LR parser to do the pattern matching. The input tree can be treated as a string by using its prefix representation. For example, the prefix representation for the tree in Fig. 8.19 is

$$= \mathbf{ind} + + C_a\ R_{\mathsf{SP}}\ \mathbf{ind} + C_i\ R_{\mathsf{SP}} + M_b\ C_1$$

The tree-translation scheme can be converted into a syntax-directed translation scheme by replacing the tree-rewriting rules with the productions of a context-free grammar in which the right sides are prefix representations of the instruction templates.

Example 8.21: The syntax-directed translation scheme in Fig. 8.21 is based on the tree-translation scheme in Fig. 8.20.

The nonterminals of the underlying grammar are R and M. The terminal **m** represents a specific memory location, such as the location for the global variable b in Example 8.18. The production $M \rightarrow \mathbf{m}$ in Rule (10) can be thought of as matching M with **m** prior to using one of the templates involving M. Similarly, we introduce a terminal **sp** for register SP and add the production $R \rightarrow \mathbf{SP}$. Finally, terminal **c** represents constants.

Using these terminals, the string for the input tree in Fig. 8.19 is

https://hemanthrajhemu.github.io

1)	$R_i \rightarrow \mathbf{c}_a$		`{ LD Ri, #a }`
2)	$R_i \rightarrow M_x$		`{ LD Ri, x }`
3)	$M \rightarrow\ = M_x\ R_i$		`{ ST x, Ri }`
4)	$M \rightarrow\ = \mathbf{ind}\ R_i\ R_j$		`{ ST *Ri, Rj }`
5)	$R_i \rightarrow \mathbf{ind}\ +\ \mathbf{c}_a\ R_j$		`{ LD Ri, a(Rj) }`
6)	$R_i \rightarrow +\ R_i\ \mathbf{ind}\ +\ \mathbf{c}_a\ R_j$		`{ ADD Ri, Ri, a(Rj) }`
7)	$R_i \rightarrow +\ R_i\ R_j$		`{ ADD Ri, Ri, Rj }`
8)	$R_i \rightarrow +\ R_i\ \mathbf{c}_1$		`{ INC Ri }`
9)	$R \rightarrow \mathbf{sp}$		
10)	$M \rightarrow \mathbf{m}$		

Figure 8.21: Syntax-directed translation scheme constructed from Fig. 8.20

$$= \mathbf{ind} + + \mathbf{c}_a\ \mathbf{sp}\ \mathbf{ind} + \mathbf{c}_i\ \mathbf{sp} + \mathbf{m}_b\ \mathbf{c}_1$$

□

From the productions of the translation scheme we build an LR parser using one of the LR-parser construction techniques of Chapter 4. The target code is generated by emitting the machine instruction corresponding to each reduction.

A code-generation grammar is usually highly ambiguous, and some care needs to be given to how the parsing-action conflicts are resolved when the parser is constructed. In the absence of cost information, a general rule is to favor larger reductions over smaller ones. This means that in a reduce-reduce conflict, the longer reduction is favored; in a shift-reduce conflict, the shift move is chosen. This "maximal munch" approach causes a larger number of operations to be performed with a single machine instruction.

There are some benefits to using LR parsing in code generation. First, the parsing method is efficient and well understood, so reliable and efficient code generators can be produced using the algorithms described in Chapter 4. Second, it is relatively easy to retarget the resulting code generator; a code selector for a new machine can be constructed by writing a grammar to describe the instructions of the new machine. Third, the quality of the code generated can be made efficient by adding special-case productions to take advantage of machine idioms.

However, there are some challenges as well. A left-to-right order of evaluation is fixed by the parsing method. Also, for some machines with large numbers of addressing modes, the machine-description grammar and resulting parser can become inordinately large. As a consequence, specialized techniques are necessary to encode and process the machine-description grammars. We must also be careful that the resulting parser does not block (has no next move) while parsing an expression tree, either because the grammar does not handle some operator patterns or because the parser has made the wrong resolution of some parsing-action conflict. We must also make sure the parser does not get into an

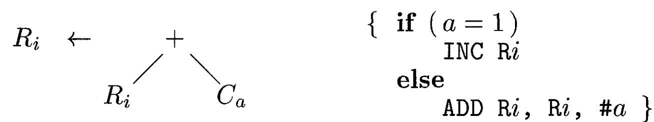
https://hemanthrajhemu.github.io

infinite loop of reductions of productions with single symbols on the right side. The looping problem can be solved using a state-splitting technique at the time the parser tables are generated.

8.9.4 Routines for Semantic Checking

In a code-generation translation scheme, the same attributes appear as in an input tree, but often with restrictions on what values the subscripts can have. For example, a machine instruction may require that an attribute value fall in a certain range or that the values of two attributes be related.

These restrictions on attribute values can be specified as predicates that are invoked before a reduction is made. In fact, the general use of semantic actions and predicates can provide greater flexibility and ease of description than a purely grammatical specification of a code generator. Generic templates can be used to represent classes of instructions and the semantic actions can then be used to pick instructions for specific cases. For example, two forms of the addition instruction can be represented with one template:



Parsing-action conflicts can be resolved by disambiguating predicates that can allow different selection strategies to be used in different contexts. A smaller description of a target machine is possible because certain aspects of the machine architecture, such as addressing modes, can be factored into the attributes. The complication in this approach is that it may become difficult to verify the accuracy of the translation scheme as a faithful description of the target machine, although this problem is shared to some degree by all code generators.

8.9.5 General Tree Matching

The LR-parsing approach to pattern matching based on prefix representations favors the left operand of a binary operator. In a prefix representation **op** $E_1\ E_2$, the limited-lookahead LR parsing decisions must be made on the basis of some prefix of E_1, since E_1 can be arbitrarily long. Thus, pattern matching can miss nuances of the target-instruction set that are due to right operands.

Instead prefix representation, we could use a postfix representation. But, then an LR-parsing approach to pattern matching would favor the right operand.

For a hand-written code generator, we can use tree templates, as in Fig. 8.20, as a guide and write an ad-hoc matcher. For example, if the root of the input tree is labeled **ind**, then the only pattern that could match is for rule (5); otherwise, if the root is labeled +, then the patterns that could match are for rules (6-8).

https://hemanthrajhemu.github.io

For a code-generator generator, we need a general tree-matching algorithm. An efficient top-down algorithm can be developed by extending the string-pattern-matching techniques of Chapter 3. The idea is to represent each template as a set of strings, where a string corresponds to a path from the root to a leaf in the template. We treat all operands equally by including the position number of a child, from left to right, in the strings.

Example 8.22: In building the set of strings for an instruction set, we shall drop the subscripts, since pattern matching is based on the attributes alone, not on their values.

The templates in Fig. 8.22 have the following set of strings from the root to a leaf:

$$C$$
$$+ \; 1 \; R$$
$$+ \; 2 \; \textbf{ind} \; 1 \; + \; 1 \; C$$
$$+ \; 2 \; \textbf{ind} \; 1 \; + \; 2 \; R$$
$$+ \; 2 \; R$$

The string C represents the template with C at the root. The string $+ 1 R$ represents the $+$ and its left operand R in the two templates that have $+$ at the root. □

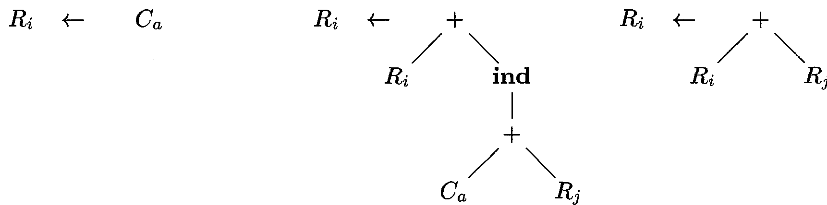


Figure 8.22: An instruction set for tree matching

Using sets of strings as in Example 8.22, a tree-pattern matcher can be constructed by using techniques for efficiently matching multiple strings in parallel.

In practice, the tree-rewriting process can be implemented by running the tree-pattern matcher during a depth-first traversal of the input tree and performing the reductions as the nodes are visited for the last time.

Instruction costs can be taken into account by associating with each tree-rewriting rule the cost of the sequence of machine instructions generated if that rule is applied. In Section 8.11, we discuss a dynamic programming algorithm that can be used in conjunction with tree-pattern matching.

By running the dynamic programming algorithm concurrently, we can select an optimal sequence of matches using the cost information associated with each rule. We may need to defer deciding upon a match until the cost of all alternatives is known. Using this approach, a small, efficient code generator can

https://hemanthrajhemu.github.io

be constructed quickly from a tree-rewriting scheme. Moreover, the dynamic programming algorithm frees the code-generator designer from having to resolve conflicting matches or decide upon an order for the evaluation.

8.9.6 Exercises for Section 8.9

Exercise 8.9.1: Construct syntax trees for each of the following statements assuming all nonconstant operands are in memory locations:

a) `x = a * b + c * d;`

b) `x[i] = y[j] * z[k];`

c) `x = x + 1;`

Use the tree-rewriting scheme in Fig. 8.20 to generate code for each statement.

Exercise 8.9.2: Repeat Exercise 8.9.1 above using the syntax-directed translation scheme in Fig. 8.21 in place of the tree-rewriting scheme.

! **Exercise 8.9.3:** Extend the tree-rewriting scheme in Fig. 8.20 to apply to while-statements.

! **Exercise 8.9.4:** How would you extend tree rewriting to apply to DAG's?

8.10 Optimal Code Generation for Expressions

We can choose registers optimally when a basic block consists of a single expression evaluation, or if we accept that it is sufficient to generate code for a block one expression at a time. In the following algorithm, we introduce a numbering scheme for the nodes of an expression tree (a syntax tree for an expression) that allows us to generate optimal code for an expression tree when there is a fixed number of registers with which to evaluate the expression.

8.10.1 Ershov Numbers

We begin by assigning to the nodes of an expression tree a number that tells how many registers are needed to evaluate that node without storing any temporaries. These numbers are sometimes called *Ershov numbers*, after A. Ershov, who used a similar scheme for machines with a single arithmetic register. For our machine model, the rules are:

1. Label any leaf 1.

2. The label of an interior node with one child is the label of its child.

3. The label of an interior node with two children is

https://hemanthrajhemu.github.io

(a) The larger of the labels of its children, if those labels are different.

(b) One plus the label of its children if the labels are the same.

Example 8.23 : In Fig. 8.23 we see an expression tree (with operators omitted) that might be the tree for expression $(a - b) + e \times (c + d)$ or the three-address code:

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

Each of the five leaves is labeled 1 by rule (1). Then, we can label the interior node for `t1 = a - b`, since both of its children are labeled. Rule (3b) applies, so it gets label one more than the labels of its children, that is, 2. The same holds for the interior node for `t2 = c + d`.

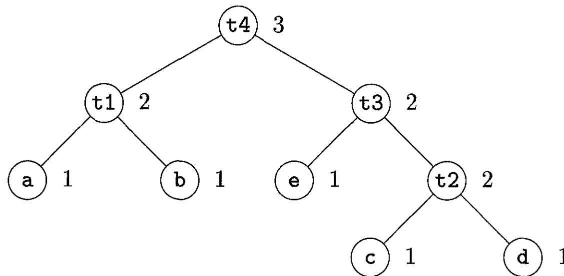


Figure 8.23: A tree labeled with Ershov numbers

Now, we can work on the node for `t3 = e * t2`. Its children have labels 1 and 2, so the label of the node for `t3` is the maximum, 2, by rule (3a). Finally, the root, the node for `t4 = t1 + t3`, has two children with label 2, and therefore it gets label 3. □

8.10.2 Generating Code From Labeled Expression Trees

It can be proved that, in our machine model, where all operands must be in registers, and registers can be used by both an operand and the result of an operation, the label of a node is the fewest registers with which the expression can be evaluated using no stores of temporary results. Since in this model, we are forced to load each operand, and we are forced to compute the result corresponding to each interior node, the only thing that can make the generated code inferior to the optimal code is if there are unnecessary stores of temporaries. The argument for this claim is embedded in the following algorithm for generating code with no stores of temporaries, using a number of registers equal to the label of the root.

Algorithm 8.24 : Generating code from a labeled expression tree.

https://hemanthrajhemu.github.io

INPUT: A labeled tree with each operand appearing once (that is, no common subexpressions).

OUTPUT: An optimal sequence of machine instructions to evaluate the root into a register.

METHOD: The following is a recursive algorithm to generate the machine code. The steps below are applied, starting at the root of the tree. If the algorithm is applied to a node with label k, then only k registers will be used. However, there is a "base" $b \geq 1$ for the registers used so that the actual registers used are $R_b, R_{b+1}, \ldots R_{b+k-1}$. The result always appears in R_{b+k-1}.

1. To generate machine code for an interior node with label k and two children with equal labels (which must be $k-1$) do the following:
   (a) Recursively generate code for the right child, using base $b+1$. The result of the right child appears in register R_{b+k}.
   (b) Recursively generate code for the left child, using base b; the result appears in R_{b+k-1}.
   (c) Generate the instruction `OP` $R_{b+k}, R_{b+k-1}, R_{b+k}$, where `OP` is the appropriate operation for the interior node in question.
2. Suppose we have an interior node with label k and children with unequal labels. Then one of the children, which we'll call the "big" child, has label k, and the other child, the "little" child, has some label $m < k$. Do the following to generate code for this interior node, using base b:
   (a) Recursively generate code for the big child, using base b; the result appears in register R_{b+k-1}.
   (b) Recursively generate code for the small child, using base b; the result appears in register R_{b+m-1}. Note that since $m < k$, neither R_{b+k-1} nor any higher-numbered register is used.
   (c) Generate the instruction `OP` $R_{b+k-1}, R_{b+m-1}, R_{b+k-1}$ or the instruction `OP` $R_{b+k-1}, R_{b+k-1}, R_{b+m-1}$, depending on whether the big child is the right or left child, respectively.
3. For a leaf representing operand x, if the base is b generate the instruction `LD` R_b, x.

□

Example 8.25: Let us apply Algorithm 8.24 to the tree of Fig. 8.23. Since the label of the root is 3, the result will appear in R_3, and only R_1, R_2, and R_3 will be used. The base for the root is $b = 1$. Since the root has children of equal labels, we generate code for the right child first, with base 2.

When we generate code for the right child of the root, labeled `t3`, we find the big child is the right child and the little child is the left child. We thus generate code for the right child first, with $b = 2$. Applying the rules for equal-labeled children and leaves, we generate the following code for the node labeled $t2$:

https://hemanthrajhemu.github.io

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
```

Next, we generate code for the left child of the right child of the root; this node is the leaf labeled e. Since $b = 2$, the proper instruction is

```
LD  R2, e
```

Now we can complete the code for the right child of the root by adding the instruction

```
MUL R3, R2, R3
```

The algorithm proceeds to generate code for the left child of the root, leaving the result in R_2, and with base 1. The complete sequence of instructions is shown in Fig. 8.24. □

```
LD  R3, d
LD  R2, c
ADD R3, R2, R3
LD  R2, e
MUL R3, R2, R3
LD  R2, b
LD  R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Figure 8.24: Optimal three-register code for the tree of Fig. 8.23

8.10.3 Evaluating Expressions with an Insufficient Supply of Registers

When there are fewer registers available than the label of the root of the tree, we cannot apply Algorithm 8.24 directly. We need to introduce some store instructions that spill values of subtrees into memory, and we then need to load those values back into registers as needed. Here is the modified algorithm that takes into account a limitation on the number of registers.

Algorithm 8.26: Generating code from a labeled expression tree.

INPUT: A labeled tree with each operand appearing once (i.e., no common subexpressions) and a number of registers $r \geq 2$.

OUTPUT: An optimal sequence of machine instructions to evaluate the root into a register, using no more than r registers, which we assume are $R_1, R_2, \ldots, R_r$.

https://hemanthrajhemu.github.io

METHOD: Apply the following recursive algorithm, starting at the root of the tree, with base $b = 1$. For a node N with label r or less, the algorithm is exactly the same as Algorithm 8.24, and we shall not repeat those steps here. However, for interior nodes with a label $k > r$, we need to work on each side of the tree separately and store the result of the larger subtree. That result is brought back into memory just before node N is evaluated, and the final step will take place in registers R_{r-1} and R_r. The modifications to the basic algorithm are as follows:

1. Node N has at least one child with label r or greater. Pick the larger child (or either if their labels are the same) to be the "big" child and let the other child be the "little" child.

2. Recursively generate code for the big child, using base $b = 1$. The result of this evaluation will appear in register R_r.

3. Generate the machine instruction ST t_k, R_r, where t_k is a temporary variable used for temporary results used to help evaluate nodes with label k.

4. Generate code for the little child as follows. If the little child has label r or greater, pick base $b = 1$. If the label of the little child is $j < r$, then pick $b = r - j$. Then recursively apply this algorithm to the little child; the result appears in R_r.

5. Generate the instruction LD R_{r-1}, t_k.

6. If the big child is the right child of N, then generate the instruction OP R_r, R_r, R_{r-1}. If the big child is the left child, generate OP R_r, R_{r-1}, R_r.

□

Example 8.27: Let us revisit the expression represented by Fig. 8.23, but now assume that $r = 2$; that is, only registers R1 and R2 are available to hold temporaries used in the evaluation of expressions. When we apply Algorithm 8.26 to Fig. 8.23, we see that the root, with label 3, has a label that is larger than $r = 2$. Thus, we need to identify one of the children as the "big" child. Since they have equal labels, either would do. Suppose we pick the right child as the big child.

Since the label of the big child of the root is 2, there are enough registers. We thus apply Algorithm 8.24 to this subtree, with $b = 1$ and two registers. The result looks very much like the code we generated in Fig. 8.24, but with registers R1 and R2 in place of R2 and R3. This code is

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
```

https://hemanthrajhemu.github.io

Now, since we need both registers for the left child of the root, we need to generate the instruction

```
ST  t3, R2
```

Next, the left child of the root is handled. Again, the number of registers is sufficient for this child, and the code is

```
LD  R2, b
LD  R1, a
SUB R2, R1, R2
```

Finally, we reload the temporary that holds the right child of the root with the instruction

```
LD  R1, t3
```

and execute the operation at the root of the tree with the instruction

```
ADD R2, R2, R1
```

The complete sequence of instructions is shown in Fig. 8.25. □

```
LD  R2, d
LD  R1, c
ADD R2, R1, R2
LD  R1, e
MUL R2, R1, R2
ST  t3, R2
LD  R2, b
LD  R1, a
SUB R2, R1, R2
LD  R1, t3
ADD R2, R2, R1
```

Figure 8.25: Optimal three-register code for the tree of Fig. 8.23, using only two registers

8.10.4 Exercises for Section 8.10

Exercise 8.10.1: Compute Ershov numbers for the following expressions:

a) $a/(b+c) - d*(e+f)$.

b) $a + b*\big(c*(d+e)\big)$.

https://hemanthrajhemu.github.io

c) $(-a + *p) * \big((b - *q)/(-c + *r)\big)$.

Exercise 8.10.2: Generate optimal code using two registers for each of the expressions of Exercise 8.10.1.

Exercise 8.10.3: Generate optimal code using three registers for each of the expressions of Exercise 8.10.1.

! **Exercise 8.10.4:** Generalize the computation of Ershov numbers to expression trees with interior nodes with three or more children.

! **Exercise 8.10.5:** An assignment to an array element, such as `a[i] = x`, appears to be an operator with three operands: a, i, and x. How would you modify the tree-labeling scheme to generate optimal code for this machine model?

! **Exercise 8.10.6:** The original Ershov numbers were used for a machine that allowed the right operand of an expression to be in memory, rather than a register. How would you modify the tree-labeling scheme to generate optimal code for this machine model?

! **Exercise 8.10.7:** Some machines require two registers for certain single-precision values. Suppose that the result of a multiplication of single-register quantities requires two consecutive registers, and when we divide a/b, the value of a must be held in two consecutive registers. How would you modify the tree-labeling scheme to generate optimal code for this machine model?

8.11 Dynamic Programming Code-Generation

Algorithm 8.26 in Section 8.10 produces optimal code from an expression tree using an amount of time that is a linear function of the size of the tree. This procedure works for machines in which all computation is done in registers and in which instructions consist of an operator applied to two registers or to a register and a memory location.

An algorithm based on the principle of dynamic programming can be used to extend the class of machines for which optimal code can be generated from expression trees in linear time. The dynamic programming algorithm applies to a broad class of register machines with complex instruction sets.

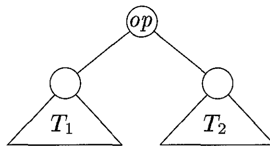
The dynamic programming algorithm can be used to generate code for any machine with r interchangeable registers `R0`, `R1`, ... , `R`$r-1$ and load, store, and add instructions. For simplicity, we assume every instruction costs one unit, although the dynamic programming algorithm can easily be modified to work even if each instruction has its own cost.

https://hemanthrajhemu.github.io

8.11.1 Contiguous Evaluation

The dynamic programming algorithm partitions the problem of generating optimal code for an expression into the subproblems of generating optimal code for the subexpressions of the given expression. As a simple example, consider an expression E of the form $E_1 + E_2$. An optimal program for E is formed by combining optimal programs for E_1 and E_2, in one or the other order, followed by code to evaluate the operator $+$. The subproblems of generating optimal code for E_1 and E_2 are solved similarly.

An optimal program produced by the dynamic programming algorithm has an important property. It evaluates an expression $E = E_1$ **op** E_2 "contiguously." We can appreciate what this means by looking at the syntax tree T for E:



Here T_1 and T_2 are trees for E_1 and E_2, respectively.

We say a program P evaluates a tree T *contiguously* if it first evaluates those subtrees of T that need to be computed into memory. Then, it evaluates the remainder of T either in the order T_1, T_2, and then the root, or in the order T_2, T_1, and then the root, in either case using the previously computed values from memory whenever necessary. As an example of noncontiguous evaluation, P might first evaluate part of T_1 leaving the value in a register (instead of memory), next evaluate T_2, and then return to evaluate the rest of T_1.

For the register machine in this section, we can prove that given any machine-language program P to evaluate an expression tree T, we can find an equivalent program P' such that

1. P' is of no higher cost than P,
2. P' uses no more registers than P, and
3. P' evaluates the tree contiguously.

This result implies that every expression tree can be evaluated optimally by a contiguous program.

By way of contrast, machines with even-odd register pairs do not always have optimal contiguous evaluations; the x86 architecture uses register pairs for multiplication and division. For such machines, we can give examples of expression trees in which an optimal machine language program must first evaluate into a register a portion of the left subtree of the root, then a portion of the right subtree, then another part of the left subtree, then another part of the right, and so on. This type of oscillation is unnecessary for an optimal evaluation of any expression tree using the machine in this section.

https://hemanthrajhemu.github.io

The contiguous evaluation property defined above ensures that for any expression tree T there always exists an optimal program that consists of optimal programs for subtrees of the root, followed by an instruction to evaluate the root. This property allows us to use a dynamic programming algorithm to generate an optimal program for T.

8.11.2 The Dynamic Programming Algorithm

The dynamic programming algorithm proceeds in three phases (suppose the target machine has r registers):

1. Compute bottom-up for each node n of the expression tree T an array C of costs, in which the ith component $C[i]$ is the optimal cost of computing the subtree S rooted at n into a register, assuming i registers are available for the computation, for $1 \leq i \leq r$.

2. Traverse T, using the cost vectors to determine which subtrees of T must be computed into memory.

3. Traverse each tree using the cost vectors and associated instructions to generate the final target code. The code for the subtrees computed into memory locations is generated first.

Each of these phases can be implemented to run in time linearly proportional to the size of the expression tree.

The cost of computing a node n includes whatever loads and stores are necessary to evaluate S in the given number of registers. It also includes the cost of computing the operator at the root of S. The zeroth component of the cost vector is the optimal cost of computing the subtree S into memory. The contiguous evaluation property ensures that an optimal program for S can be generated by considering combinations of optimal programs only for the subtrees of the root of S. This restriction reduces the number of cases that need to be considered.

In order to compute the costs $C[i]$ at node n, we view the instructions as tree-rewriting rules, as in Section 8.9. Consider each template E that matches the input tree at node n. By examining the cost vectors at the corresponding descendants of n, determine the costs of evaluating the operands at the leaves of E. For those operands of E that are registers, consider all possible orders in which the corresponding subtrees of T can be evaluated into registers. In each ordering, the first subtree corresponding to a register operand can be evaluated using i available registers, the second using $i-1$ registers, and so on. To account for node n, add in the cost of the instruction associated with the template E. The value $C[i]$ is then the minimum cost over all possible orders.

The cost vectors for the entire tree T can be computed bottom up in time linearly proportional to the number of nodes in T. It is convenient to store at each node the instruction used to achieve the best cost for $C[i]$ for each value

https://hemanthrajhemu.github.io

of i. The smallest cost in the vector for the root of T gives the minimum cost of evaluating T.

Example 8.28: Consider a machine having two registers R0 and R1, and the following instructions, each of unit cost:

```
LD  Ri, Mj        // Ri = Mj
op  Ri, Ri, Rj    // Ri = Ri op Rj
op  Ri, Ri, Mj    // Ri = Ri op Mj
LD  Ri, Rj        // Ri = Rj
ST  Mi, Rj        // Mi = Rj
```

In these instructions, Ri is either R0 or R1, and Mj is a memory location. The operator *op* corresponds to an arithmetic operators.

Let us apply the dynamic programming algorithm to generate optimal code for the syntax tree in Fig 8.26. In the first phase, we compute the cost vectors shown at each node. To illustrate this cost computation, consider the cost vector at the leaf a. $C[0]$, the cost of computing a into memory, is 0 since it is already there. $C[1]$, the cost of computing a into a register, is 1 since we can load it into a register with the instruction `LD R0, a`. $C[2]$, the cost of loading a into a register with two registers available, is the same as that with one register available. The cost vector at leaf a is therefore $(0, 1, 1)$.

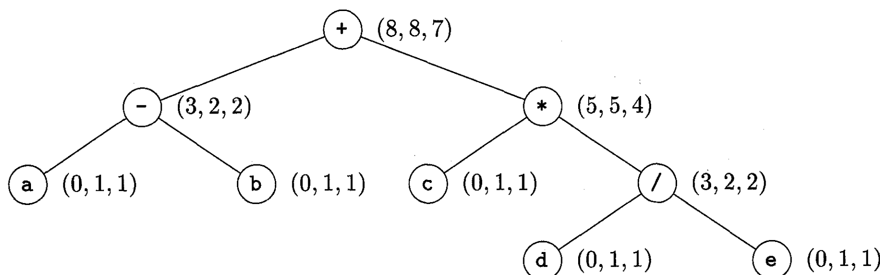


Figure 8.26: Syntax tree for (a-b)+c*(d/e) with cost vector at each node

Consider the cost vector at the root. We first determine the minimum cost of computing the root with one and two registers available. The machine instruction `ADD R0,R0,M` matches the root, because the root is labeled with the operator +. Using this instruction, the minimum cost of evaluating the root with one register available is the minimum cost of computing its right subtree into memory, plus the minimum cost of computing its left subtree into the register, plus 1 for the instruction. No other way exists. The cost vectors at the right and left children of the root show that the minimum cost of computing the root with one register available is $5 + 2 + 1 = 8$.

Now consider the minimum cost of evaluating the root with two registers available. Three cases arise depending on which instruction is used to compute the root and in what order the left and right subtrees of the root are evaluated.

https://hemanthrajhemu.github.io

1. Compute the left subtree with two registers available into register R0, compute the right subtree with one register available into register R1, and use the instruction ADD R0,R0,R1 to compute the root. This sequence has cost $2+5+1=8$.

2. Compute the right subtree with two registers available into R1, compute the left subtree with one register available into R0, and use the instruction ADD R0,R0,R1. This sequence has cost $4+2+1=7$.

3. Compute the right subtree into memory location M, compute the left subtree with two registers available into register R0, and use the instruction ADD R0,R0,M. This sequence has cost $5+2+1=8$.

The second choice gives the minimum cost 7.

The minimum cost of computing the root into memory is determined by adding one to the minimum cost of computing the root with all registers available; that is, we compute the root into a register and then store the result. The cost vector at the root is therefore (8,8,7).

From the cost vectors we can easily construct the code sequence by making a traversal of the tree. From the tree in Fig. 8.26, assuming two registers are available, an optimal code sequence is

```
LD  R0, c          // R0 = c
LD  R1, d          // R1 = d
DIV R1, R1, e      // R1 = R1 / e
MUL R0, R0, R1     // R0 = R0 * R1
LD  R1, a          // R1 = a
SUB R1, R1, b      // R1 = R1 - b
ADD R1, R1, R0     // R1 = R1 + R0
```

□

Dynamic programming techniques have been used in a number of compilers, including the second version of the portable C compiler, PCC2. The technique facilitates retargeting because of the applicability of the dynamic programming technique to a broad class of machines.

8.11.3 Exercises for Section 8.11

Exercise 8.11.1: Augment the tree-rewriting scheme in Fig. 8.20 with costs, and use dynamic programming and tree matching to generate code for the statements in Exercise 8.9.1.

!! **Exercise 8.11.2:** How would you extend dynamic programming to do optimal code generation on dags?

https://hemanthrajhemu.github.io

8.12 Summary of Chapter 8

- *Code generation* is the final phase of a compiler. The code generator maps the intermediate representation produced by the front end, or if there is a code optimization phase by the code optimizer, into the target program.

- *Instruction selection* is the process of choosing target-language instructions for each IR statement.

- *Register allocation* is the process of deciding which IR values to keep in registers. Graph coloring is an effective technique for doing register allocation in compilers.

- *Register assignment* is the process of deciding which register should hold a given IR value.

- A *retargetable compiler* is one that can generate code for multiple instruction sets.

- A *virtual machine* is an interpreter for a bytecode intermediate language produced by languages such as Java and C#.

- A *CISC machine* is typically a two-address machine with relatively few registers, several register classes, and variable-length instructions with complex addressing modes.

- A *RISC machine* is typically a three-address machine with many registers in which operations are done in registers.

- A *basic block* is a maximal sequence of consecutive three-address statements in which flow of control can only enter at the first statement of the block and leave at the last statement without halting or branching except possibly at the last statement in the basic block.

- A *flow graph* is a graphical representation of a program in which the nodes of the graph are basic blocks and the edges of the graph show how control can flow among the blocks.

- A *loop* in a flow graph is a strongly connected region with a single entry point called the loop header.

- A *DAG* representation of a basic block is a directed acyclic graph in which the nodes of the DAG represent the statements within the block and each child of a node corresponds to the statement that is the last definition of an operand used in the statement.

- *Peephole optimizations* are local code-improving transformations that can be applied to a program, usually through a sliding window.

https://hemanthrajhemu.github.io

- ✦ *Instruction selection* can be done by a tree-rewriting process in which tree patterns corresponding to machine instructions are used to tile a syntax tree. We can associate costs with the tree-rewriting rules and apply dynamic programming to obtain an optimal tiling for useful classes of machines and expressions.

- ✦ An *Ershov number* tells how many registers are needed to evaluate an expression without storing any temporaries.

- ✦ *Spill code* is an instruction sequence that stores a value in a register into memory in order to make room to hold another value in that register.

8.13 References for Chapter 8

Many of the techniques covered in this chapter have their origins in the earliest compilers. Ershov's labeling algorithm appeared in 1958 [7]. Sethi and Ullman [16] used this labeling in an algorithm that they prove generated optimal code for arithmetic expressions. Aho and Johnson [1] used dynamic programming to generate optimal code for expression trees on CISC machines. Hennessy and Patterson [12] has a good discussion on the evolution of CISC and RISC machine architectures and the tradeoffs involved in designing a good instruction set.

RISC architectures became popular after 1990, although their origins go back to computers like the CDC 6600, first delivered in 1964. Many of the computers designed before 1990 were CISC machines, but most of the general-purpose computers installed after 1990 are still CISC machines because they are based on the Intel 80x86 architecture and its descendants, such as the Pentium. The Burroughs B5000 delivered in 1963 was an early stack-based machine.

Many of the heuristics for code generation proposed in this chapter have been used in various compilers. Our strategy of allocating a fixed number of registers to hold variables for the duration of a loop was used in the implementation of Fortran H by Lowry and Medlock [13].

Efficient register allocation techniques have also been studied from the time of the earliest compilers. Graph coloring as a register-allocation technique was proposed by Cocke, Ershov [8], and Schwartz [15]. Many variants of graph-coloring algorithms have been proposed for register allocation. Our treatment of graph coloring follows Chaitin [3] [4]. Chow and Hennessy describe their priority-based coloring algorithm for register allocation in [5]. See [6] for a discussion of more recent graph-splitting and rewriting techniques for register allocation.

Lexical analyzer and parser generators spurred the development of pattern-directed instruction selection. Glanville and Graham [11] used LR-parser generation techniques for automated instruction selection. Table-driven code generators evolved into a variety of tree-pattern matching code-generation tools [14]. Aho, Ganapathi, and Tjiang [2] combined efficient tree-pattern matching

https://hemanthrajhemu.github.io

techniques with dynamic programming in the code generation tool *twig*. Fraser, Hanson, and Proebsting [10] further refined these ideas in their simple efficient code-generator generator.

1. Aho, A. V. and S. C. Johnson, "Optimal code generation for expression trees," *J. ACM* **23**:3, pp. 488–501.

2. Aho, A. V., M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Programming Languages and Systems* **11**:4 (1989), pp. 491–516.

3. Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages* **6**:1 (1981), pp. 47–57.

4. Chaitin, G. J., "Register allocation and spilling via graph coloring," *ACM SIGPLAN Notices* **17**:6 (1982), pp. 201–207.

5. Chow, F. and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Trans. Programming Languages and Systems* **12**:4 (1990), pp. 501–536.

6. Cooper, K. D. and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, San Francisco CA, 2004.

7. Ershov, A. P., "On programming of arithmetic operations," *Comm. ACM* **1**:8 (1958), pp. 3–6. Also, *Comm. ACM* **1**:9 (1958), p. 16.

8. Ershov, A. P., *The Alpha Automatic Programming System*, Academic Press, New York, 1971.

9. Fischer, C. N. and R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin-Cummings, Redwood City, CA, 1991.

10. Fraser, C. W., D. R. Hanson, and T. A. Proebsting, "Engineering a simple, efficient code generator generator," *ACM Letters on Programming Languages and Systems* **1**:3 (1992), pp. 213–226.

11. Glanville, R. S. and S. L. Graham, "A new method for compiler code generation," *Conf. Rec. Fifth ACM Symposium on Principles of Programming Languages* (1978), pp. 231–240.

12. Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufman, San Francisco, 2003.

13. Lowry, E. S. and C. W. Medlock, "Object code optimization," *Comm. ACM* **12**:1 (1969), pp. 13–22.

https://hemanthrajhemu.github.io

14. Pelegri-Llopart, E. and S. L. Graham, "Optimal code generation for expressions trees: an application of BURS theory," *Conf. Rec. Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), pp. 294–308.

15. Schwartz, J. T., *On Programming: An Interim Report on the SETL Project*, Technical Report, Courant Institute of Mathematical Sciences, New York, 1973.

16. Sethi, R. and J. D. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM* **17**:4 (1970), pp. 715–728.

https://hemanthrajhemu.github.io

https://hemanthrajhemu.github.io