7

Syntax-Directed Translation

The parsers discussed in Chapters Chapter:global:five and Chapter:global:six can recognize syntactically valid inputs. However, compilers are typically required to perform some translation of the input source into a target representation, as discussed in Chapter Chapter:global:two. Some compilers are *single-pass*, translating programs as they parse without taking any intermediate steps. Most compilers accomplish translation using multiple passes. Instead of repeatedly scanning the input program, compilers typically create an intermediate structure called the **abstract syntax tree** (AST) as a by-product of the parse. The AST then serves as a mechanism for conveying information between compiler passes. In this chapter we study how to formulate grammars and production-specific code sequences for the purpose of direct translation or the creation of ASTs. Sections 7.3 and 7.4 consider bottom-up and top-down translation, respectively; Section 7.5 considers the design and synthesis of ASTs.

7.1 Overview

The work performed by a compiler while parsing is generally termed *syntax-directed translation*. The grammar on which the parser is based causes a specific sequence of derivation steps to be taken for a given input program. In constructing the derivation, a parser performs a sequence of *syntactic actions* as described in Chapters Chapter:global:five and Chapter:global:six; such actions (*e.g.*, *shift* and *reduce* for LR parsing) pertain only with the grammar's terminal and nonterminal symbols. To achieve syntax-directed translation, most parser generators allow a segment of

computer code to be associated with each grammar production; such code is executed whenever the production participates in the derivation. Typically, the code deals with the *meaning* of the grammar symbols. For example, the syntactic token identifier has a specific value when a production involving identifier is applied. The *semantic actions* performed by the parser through execution of the supplied code segment can reference the actual string comprising the terminal, perhaps for the purpose of generating code to load or store the value associated with the identifier.

In automatically generated parsers, the parser driver (Figure Figure:six:driver for LR parsing) is usually responsible for executing the semantic actions. To simplify calling conventions, the driver and the grammar's semantic actions are written in the same programming language. Parser generators also provide a mechanism for the supplied semantic actions to reference semantic values associated with the associated production's grammar symbols. Semantic actions are also easily inserted into *ad hoc* parsers by specifying code sequences that execute in concert with the parser.

Formulating an appropriate set of semantic actions requires a firm understanding of how derivations are traced in a parse, be it bottom-up or top-down. When one encouters difficulties in writing clear and elegant semantic actions, grammar reformulation can often simplify the task at hand by computing semantic values at more appropriate points. Even after much work is expended to obtain an LALR(1) grammar for a programming language, it is not unusual to revise the grammar during this phase of compiler construction.

7.2 Synthesized and inherited attributes

In a parse (derivation) tree, each node corresponds to a grammar symbol utilized in some derivation step. The information associated with a grammar symbol by semantic actions become *attributes* of the parse tree. In Section 7.3 we examine how to specify semantic actions for bottom-up (LALR(1)) parsers, which essentially create parse trees in a *postorder* fashion. For syntax-directed translation, this style nicely accommodates situations in which attributes primarily flow from the leaves of a derivation tree toward its root. If we imagine that each node of a parse tree can consume and produce information, then nodes consume information from their children and produce information for their parent in a bottom-up parse. An example using such *synthetic attributes* flow is shown in Figure 7.1: the attribute associated with each node is an expression value, and the value of the entire computation becomes available at the root of the parse tree.

By contrast, consider the problem of propagating symbol table information through a parse tree, so that inner scopes have access to information present in outer scopes. As shown in Figure 7.1, such *inherited attributes* flow from the root of the parse tree toward its leaves. As discussed in Section 7.4, top-down parsing nice accommodates such information flow because its parse trees are created in a preorder fashion.

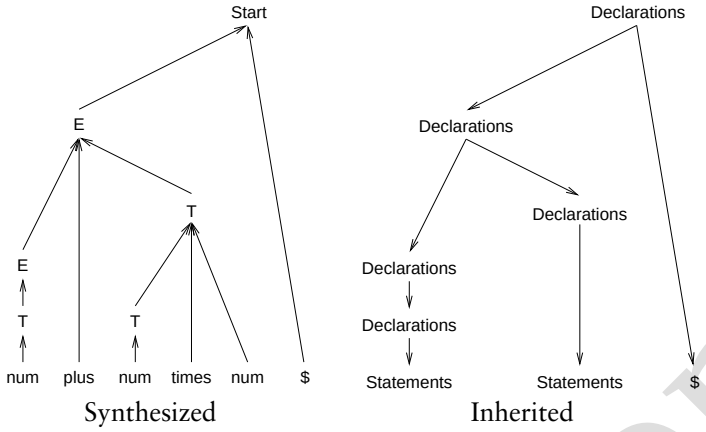

Figure 7.1: Synthesized and inherited attributes.

Most programming languages require information flow in both directions. While some tools allow both styles of propagation to coexist peacefully, such tools are not in widespread use; moreover, choreographing attribute flow can be difficult to specify and maintain. In the ensuing sections we focus only on syntax-directed parsers that allow attributes to be synthesized or inherited, but not both. In practice, this is not a serious limitation. attributes flowing in the opposite direction are typically accommodated by global structures. For example, symbol information can be stored in a global *symbol table* in a parser based on synthesized attributes.

7.3 Bottom-up translation

In this section we consider how to incorporate semantic actions into bottom-up parsers. Such parsers are almost always generated automatically by tools such as Cup, Yacc, or Bison, which allow specification of code sequences that execute as reductions are performed.

Consider an LR parser that is about to perform a reduction using the rule $A \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n$. As discussed in Chapter Chapter:global:six, the symbols $\mathcal{X}_1 \ldots \mathcal{X}_n$ are on top-of-stack prior to the reduction, with $\mathcal{X}_n$ topmost. The n symbols are popped from the stack and the symbol A is pushed on the stack. In such a bottom-up parse, it is possible that previous reductions have associated semantic information with the symbols $\mathcal{X}_1 \ldots \mathcal{X}_n$. The semantic action associated with $A \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n$ can take such information into consideration in synthesizing the semantic information to be associated with A. In other words, the bottom-up parser operates two stacks: the *syntactic stack* manipulates terminal and nonterminal symbols as described in Chapter Chapter:global:six, while the *semantic stack* manipulates values associated with the symbols. The code for maintaining both stacks is generated

1 Start $\rightarrow$ Digs_{ans} \$
    call PRINT(*ans*)

2 Digs_{up} $\rightarrow$ Digs_{below} d_{next}
    $up \leftarrow below \times 10 + next$

3 | d_{first}
    $up \leftarrow first$

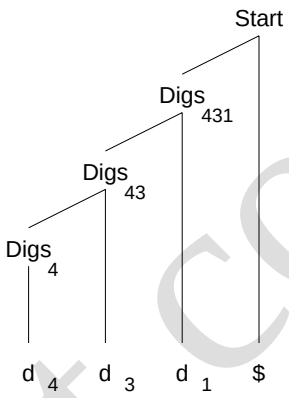

Figure 7.2: Possible grammar and parse tree for decimal 4 3 1 $.

automatically by the parser generator, based on the grammar and semantic-action code provided by the compiler writer.

7.3.1 Left recursive rules

To illustrate a bottom-up style of translation, we begin with the grammar shown in Figure 7.2. The translation task consists of computing the value of a string of digits, base 10. Compilers normally process such strings during lexical analysis, as described in Chapter Chapter:global:three. Our example serves pedagogically to illustrate the problems that can arise with syntax-directed translation during a bottom-up parse.

Syntactically, the terminal d represents a single digit; the nonterminal Digs generates one or more such symbols. Some of the grammar symbols are annotated with tags, as in Digs_{up}. The symbol Digs is a grammar terminal or nonterminal symbol; the symbol *up* represents the semantic value associated with the symbol. Such values are provided by the scanner for terminal symbols; for nonterminals, the grammar's semantic actions establish the values. For example, Rule 3 contains d_{first}. The syntactic element d represents an integer, as discovered by the scanner; the semantic tag *first* represents the integer's value, also provided by the scanner. Parser generators offer methods for declaring the *type* of the semantic symbols; for

the sake of language-neutrality and clarity we omit type declarations in our examples. In the grammar of Figure 7.2, all semantic tags would be declared to have integer value.

We now analyze how the semantic actions in Figure 7.2 compute the base-10 value of the digit string. To appreciate the interaction of Rules 2 and 3, we examine the order in which a bottom-up parse applies these rules. Figure 7.2 shows a parse tree for the input string `4 3 1 $`. In Chapter Chapter:global:six we learned that a bottom-up parse traces a rightmost derivation in reverse. The rules for `Digs` are applied as follows:

`Digs` → `d` This rule must be applied first; `d` corresponds to the input digit `4`. The semantic action causes the value of the first digit (4) to be transferred as the semantic value for `Digs`. Semantic actions such as these are often called *copy rules* because they serve only to propagate values up the parse tree.

`Digs` → `Digs d` Each subsequent `d` is processed by this rule. The semantic action

$$up \leftarrow below \times 10 + next$$

multiplies the value computed thus far (*below*) by 10 and then adds in the semantic value of the current `d` (*next*).

As seen in the above example, left-recursive rules are amenable to semantic processing of input text from left to right. Exercise 1 considers the difficulty of computing a digit string's value when the grammar rule is right-recursive.

7.3.2 Rule cloning

We extend our language slightly, so that a string of digits can be prefaced by an `x`, in which case the digit string should be interpreted base-8 rather than base-10; any input digit out of range for a base-8 number is ignored. For this task we propose the grammar shown in Figure 7.3. Rule 3 generates strings of digits as before; Rule 2 generates such strings prefaced with an `x`. Prior to inserting semantic actions, we examine the parse tree shown in Figure 7.3. As was the case with our previous example, the first `d` symbol is processed first by Rule 4 and the reamining `d` symbols are processed by Rule 5. If the string is to be interpreted base-8, then the semantic action for Rule 5 should multiply the number passed up the parse tree by 8; otherwise, 10 should be used at the multiplier.

The problem with the parse tree shown in Figure 7.3 is that the `x` is shifted on the stack with no possible action to perform: actions are possible only on reductions. Thus, when the semantic actions for Rule 5 execute, it is unknown whether we are processing a base-10 or base-8 number.

There are several approaches to remedy this problem, but the one we consider first involves *cloning* productions in the grammar. We construct two kinds of digit strings, one derived from `OctDigs` and the other derived from `DecDigs`, to obtain the grammar and semantic actions shown in Figure 7.4. Rules 4 and 5 interpret strings of digits base-10; Rules 6 and 7 generate the same syntactic strings but interpret

1	Start	→	Num $
2	Num	→	x Digs
3		|	Digs
4	Digs	→	Digs d
5		|	d

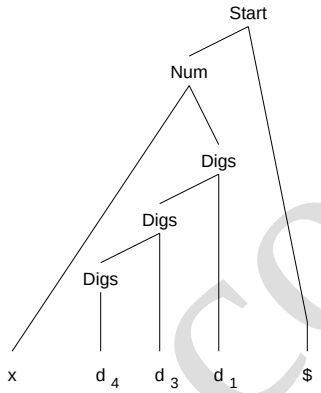

Figure 7.3: Possible grammar and parse tree for octal x 4 3 1 $.

1	Start	→	Num_{ans} $
			call PRINT(ans)
2	Num_{ans}	→	x OctDigs_{octans}
			$ans \leftarrow octans$
3		|	DecDigs_{decans}
			$ans \leftarrow decans$
4	DecDigs_{up}	→	DecDigs_{below} d_{next}
			$up \leftarrow below \times 10 + next$
5		|	d_{first}
			$up \leftarrow first$
6	OctDigs_{up}	→	OctDigs_{below} d_{next}
			$up \leftarrow below \times 8 + next$
7		|	d_{first}
			$up \leftarrow first$

Figure 7.4: Grammar with cloned productions.

```
1  Start          →  Num_ans $
                         call PRINT(ans)
2  Num_ans        →  SignalOctal Digs_octans
                         ans ← octans
3                 |  SignalDecimal Digs_decans
                         ans ← decans
4  SignalOctal    →  x
                         base ← 8
5  SignalDecimal  →  λ
                         base ← 10
6  Digs_up        →  Digs_below d_next
                         up ← below × base + next
7                 |  d_first
                         up ← first
```

Figure 7.5: Use of λ-rules to take semantic action.

their meaning base-8. The semantic check for octal digits is left as Exercise 2. With this example, we see that grammars are modified for reasons other than the parsing issues raised in Chapters Chapter:global:four and Chapter:global:five. Often, a translation task can become significantly easier if the grammar can be modified to accommodate efficient flow of semantic information.

If the grammars in Figures 7.3 and 7.4 were examined without their semantic actions, the grammar in Figure 7.3 would be favored because of its simplicity. Insertion of semantic actions often involves inflating a grammar with productions that are not necessary from a syntactic point of view. These extra productions allow needed flexibility in accomplishing semantic actions at appropriate points in the parse. However, grammar modifications can affect the parsability of a grammar. Such modifications should be made in small increments, so that the compiler writer can respond quickly and accurately to any parsability problems raised by the parser generator.

7.3.3 Global variables and λ-rules

We now examine an alternative for the grammar shown in Figure 7.3. Recall that the x was shifted without semantic notice: actions are performed only at reductions. We can modify the grammar from Figure 7.3 so that a reduction takes place after the x has been shifted but prior to processing any of the digits. The resulting grammar is shown in Figure 7.5. This grammar avoids copying the productions that generate a string of digits; however, the grammar assigns and

references a global variable (*base*) as a *side-effect* of processing an x (Rule 4) or nothing (Rule 5).

In crafting a grammar with appropriate semantic actions, care must be taken to avoid changing the actual language. Although octal strings are preceded by an x, decimal strings are unheralded in this language; the λ-rule present in Figure 7.5 serves only to cause a reduction when an x has *not* been seen. This semantic action is required to initialize *base* to its default value of 10. If the λ-rule were absent and *base* were initialized prior to operating the parser, then the parser would not correctly parse a base-8 number followed by a base-10 number, because *base* would have the residual value of 8.

While they can be quite useful, global variables are typically considered undesirable because they can render a grammar difficult to write and maintain. Synthesized attributes, such as the digits that are pushed up a parse tree, can be localized with regard to a few productions; on the other hand, global variables can be referenced in *any* semantic action. However, programming language constructs that lie outside the context-free languages must be accommodated using global variables; symbol tables (Chapter Chapter:global:eight) are the most common structure deserving of global scope.

7.3.4 Aggressive grammar restructuring

We expand our digits example further, so that a single-digit base is supplied along with the introductory x. Some examples of legal inputs and their interpretation are as follows:

Input	Meaning	Value (base-10)
`4 3 1 $`	431_{10}	431
`x 8 4 3 1 $`	431_{8}	281
`x 5 4 3 1 $`	431_{5}	116

A grammar for the new language is shown in Figure 7.6; semantic actions are included that use the global variable *base* to enforce the correct interpretation of digit strings.

While the solution in Figure 7.6 suffices, there is a solution that avoids use of global variables. The grammar as given does not place information in the parse tree where it is useful for semantic actions. This is a common problem, and the following steps are suggested as a mechanism for obtaining a grammar more appropriate for bottom-up, syntax-directed translation:

1. Sketch the parse tree that would allow bottom-up synthesis and translation without use of global variables.

2. Revise the grammar to achieve the desired parse tree.

3. Verify that the revised grammar is still suitable for parser construction. For example, if Cup or Yacc must process the grammar, then the grammar should remain LALR(1) after revision.

```
1  Start        →  Num_ans $
                       call PRINT(ans)
2  Num_ans      →  x SetBase Digs_baseans
                       ans ← baseans
3               |  SetBaseTen Digs_decans
                       ans ← decans
4  SetBase      →  d_val
                       base ← val
5  SetBaseTen   →  λ
                       base ← 10
6  Digs_up      →  Digs_below d_next
                       up ← below × base + next
7               |  d_first
                       up ← first
```

Figure 7.6: Strings with an optionally specified base.

4. Verify that the grammar still generates the same language, using proof techniques or rigorous testing.

The last step is the trickiest, because it is easy to miss subtle variations among grammars and languages.

For the problem at hand, we can avoid a global *base* variable if we can process the base early and have it propagate up the parse tree along with the value of the processed input string. Figure 7.7 sketches the parse tree we desire for the input `x 5 4 3 1 $`. The x and the first d, which specifies the base are processed in the triangle; the base can then propagate up the tree and participate in the semantic actions. From this tree we rewrite the rules for Digs to obtain the grammar shown in Figure 7.8.

The grammar in Figure 7.8 is fairly concise but it involves one novelty: the semantic value for Digs consists of two components

val which contains the interpreted value of the processed input string

base which conveys the base for interpretation

The semantic actions for Rule 3 serve to copy the base from its inception at Rule 2.

Experienced compiler writers make frequent use of rich semantic types and grammar restructuring to localize the effects of semantic actions. The resulting parsers are (mostly) free of global variables and easier to understand and modify.

Unfortunately, our grammar modification is deficient because it involves a subtle change in the language; this point is explored in great detail in Exercise 3.

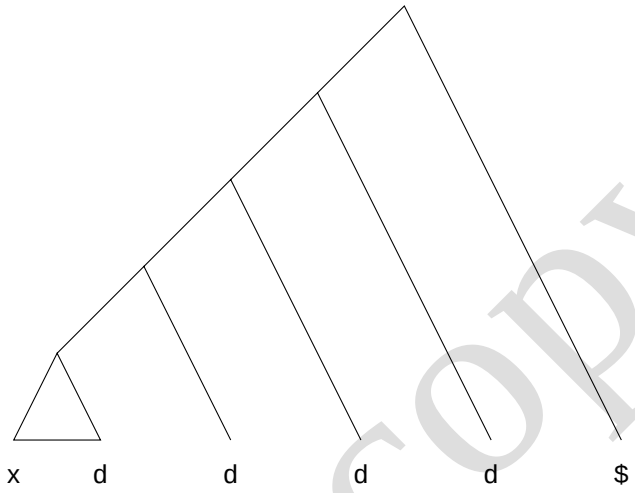

Figure 7.7: Desired parse tree.

1	Start	$\rightarrow$	Digs_{ans} \$
			call PRINT($ans.val$)
2	Digs_{up}	$\rightarrow$	$\text{SetBase}_{basespec}$
			$up.base \leftarrow basespec$
			$up.val \leftarrow 0$
3		|	Digs_{below} d_{next}
			$up.val \leftarrow below.val \times below.base + next$
			$up.base \leftarrow below.base$
4	Setbase_{n}	$\rightarrow$	λ
			$n \leftarrow 10$
5		|	x d_{num}
			$n \leftarrow num$

Figure 7.8: Grammar that avoids global variables.

1	Start	→	Value $
2	Value	→	num
3		|	lp Expr rp
4	Expr	→	plus Value Value
5		|	prod Values
6	Values	→	Value Values
7		|	λ

Figure 7.9: Grammar for Lisp-like expressions.

7.4 Top-down translation

In this section we discuss how semantic actions can be performed in top-down parsers. Such parsers are typically generated by hand in the *recursive-descent* style discussed in Chapters Chapter:global:two and Chapter:global:five. The result of such parser construction is a program; semantic actions are accomplished by code sequences inserted directly into the parser.

To illustrate this style of translation, we consider the processing of Lisp-like expressions, defined by the grammar shown in Figure 7.9. Rule 1 generates an outermost expression whose value should be printed by the semantic actions. A Value is defined by Rules 2 and 3, which generate an atomic value via a num or a parenthesized expression, respectively. Rules 4 and 5 generate a the sum of two values and a product of zero or more values, respectively. Follwowing are some observations about this grammar.

- This disparate treatment of addition (which takes two operands) and multiplication (which takes arbitrary operands) is for pedagogical purposes; it would otherwise be considered poor language design.
- A more complete expression grammar is considered in Exercises 5 and 6.
- The recursive rule for Values is right-recursive to accommodate top-down parsing.

A recursive-descent parser for the grammar in Figure 7.9 is shown in Figure 7.10. As discussed in Chapter Chapter:global:five, the parser contains a procedure for each nonterminal in the grammar. To conserve space, the error checks normally present at the end of each **switch** statement are absent in Figure 7.10. If top-down parsers applied semantic actions only when a derivation step is taken, then information could only be passed from the root of a parse tree toward its leaves. To evaluate programs written in the language of Figure 7.9, the tree must be evaluated bottom-up. It is common in recursive descent parsing to allow semantic actions both before and after a rule's right-hand side has been discovered. Of course, in an *ad hoc* parser, code can be inserted anywhere.

```
procedure START(ts)
    switch ()
        case ts.PEEK( ) ∈ { num, lp }
            ans ← VALUE( )
            call MATCH(ts, $)
            call PRINT(ans)
end
function VALUE(ts) : int
    switch ()
        case ts.PEEK( ) ∈ { num }
            call MATCH(ts, num)
            ans ← num.VALUEOF( )
            return (ans)
        case ts.PEEK( ) ∈ { lp }
            call MATCH(ts, lp)
            ans ← EXPR( )
            call MATCH(ts, rp)
            return (ans)
end
function EXPR(ts) : int
    switch ()
        case ts.PEEK( ) ∈ { plus }
            call MATCH(ts, plus)
            op1 ← VALUE( )                          1
            op2 ← VALUE( )                          2
            return (op1 + op2)                      3
        case ts.PEEK( ) ∈ { prod }
            call MATCH(ts, prod)
            ans ← VALUES( )                         4
            return (ans)
end
function VALUES(ts) : int
    switch ()
        case ts.PEEK( ) ∈ { num, lp }
            factor ← VALUE( )
            product ← VALUES( )
            return (factor × product)               5
        case ts.PEEK( ) ∈ { rp }
            return (1)                              6
end
```

Figure 7.10: Recursive-descent parser with semantic actions.

In bottom-up parsing, semantic values are associated with grammar symbols. This effect can be accomplished in a recursive-descent parser by instrumenting the parsing procedures to return semantic values. The code associated with semantically active nonterminal symbols is modified to return an appropriate semantic value as follows.

- In Figure 7.10, each method except START is responsible for computing a value associated with the string of symbols it derives; these methods are therefore instrumented to return an integer.
- The code for Rule 4 captures the values of its two operands at Steps **1** and **2**, returning their sum at Step **3**.
- The code at Step **4** anticipates that VALUES will return the product of its expressions.
- Because Values participates only in Rule 5, the semantic actions in VALUES can be customized to form such a product. Exercise 5 considers accommodation of functionally separate distinct uses of the nonterminal Values.
- The most subtle aspect of the semantic actions concerns the formation of a product of values in VALUES. With reference to the parse tree shown in Figure 7.11, a leftmost derivation generates the Value symbols from left to right. However, the semantic actions compute values *after* a rule has been applied, in the style of an LR bottom-up parse. Thus, the invocation of VALUES that finishes first is the one corresponding to Values$\rightarrow\lambda$; this rule seeds the product at Step **6**. The semantic actions then progress up the tree applying the code at Step **5**, incorporating Value expressions from right to left.

7.5 Abstract Syntax Trees

While most of a compiler's tasks can be performed in a single pass via syntax-directed translation, modern software practice frowns upon delegating so much functionality to a single component such as the parser. Tasks such as semantic analysis, symbol table construction, program optimization, and code generation are each deserving of separate treatment in a compiler; squeezing them into a single compiler pass is an admirable feat of engineering, but the resulting compiler is difficult to extend or maintain.

In designing multipass compilers, considerable attention should be devoted to designing the compiler's **intermediate representation** (IR): the form that serves to communicate information between compiler passes. In some systems, the source language itself serves as the IR; such *source-to-source* compilers are useful in research and educational settings, where the effects of a compiler's restructuring transformations (Chapter Chapter:global:sixteen) can be easily seen. Such systems

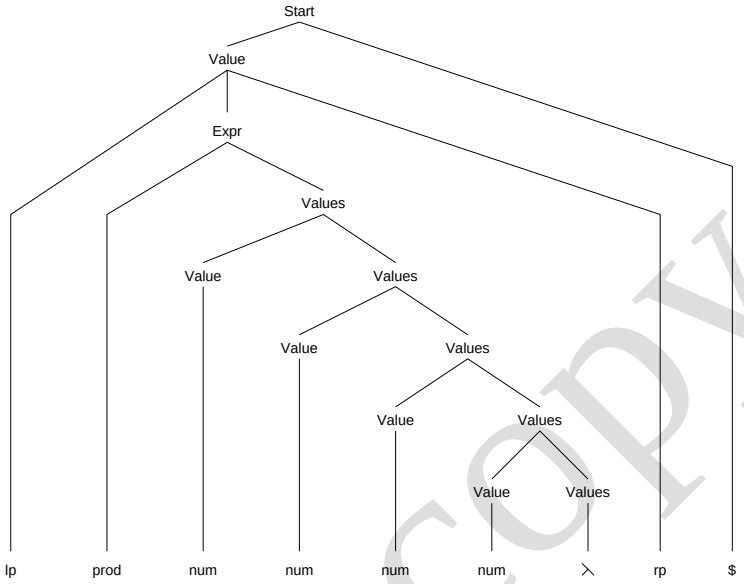

Figure 7.11: Parse tree involving Values.

can also serve as preprocessors. For example, a source Java program could be compiled into an optimized Java program, suitable as input for any Java compiler.

However, tasks such as semantic analysis and code generation create artifacts of a source program's representation that are difficult to express in the source language itself. For example, it seems difficult and pointless to encode a symbol table in the source language so that it could be passed along with the source program to subsequent compiler passes. We therefore seek an IR that can represent a source program faithfully, but in a form that is easily annotated and digested by the compiler's passes.

Parse trees such those shown in Figures 7.2 and 7.3 are *concrete*, in the sense that they result directly from application of a grammar. Consider the nonterminal Digs and its parse subtree shown in Figure 7.2. The tree leans to the left because the rules for Digs are left-recursive. If these rules were right-recursive, then the parse subtree for Digs changes accordingly. Abstractly, the symbol *Digs* represents a list of d symbols. The order in which these symbols occur as input tokens is important; whether these tokens are derived by left- or right-recursive rules does not affect the *meaning* of the list of symbols.[1]

In a practice that has become quite common, the parser can create an artifact of syntax analysis called the **abstract syntax tree** (AST). The symbols derived from Digs can be represented abstractly by the AST shown in Figure 7.12. There is

[1]However, the rules can simplify or hinder syntax-directed translation; see Exercise 1.

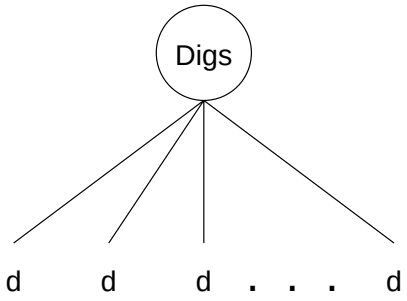

Figure 7.12: Abstract syntax tree for Digs.

no grammar that can generate such a concrete syntax tree directly: each production has a fixed number of symbols on its right-hand side. However, the tree in Figure 7.12 captures the order in which the d symbols appear, which suffices to translate the sequence of symbols into a number.

Because the AST is centrally involved in most of a compiler's activities, its design can and should evolve with the functionality of the compiler. A properly designed AST simplies both the work required in a single compiler pass as well as the manner in which the compiler passes communicate. Given a source programming language L, the development of a grammar and an AST typically proceeds as follows:

1. An unambiguous grammar for L is devised. The grammar may contain productions that serve to disambiguate the grammar. The grammar is proved unambiguous through successful construction of an LR or LL parser.

2. An AST for L is devised. The AST design typically elides grammar details concerned with disambiguation. Semantically useless symbols such as "," and ";" are also omitted.

3. Semantic actions are placed in the grammar to construct the AST. The design of the AST may require grammar modifications to simplify or localize construction. The semantic actions often rely upon utility methods for creating and manipulating AST components.

4. Passes of the compiler are designed. Each pass may place new requirements on the AST in terms of its structure and contents; grammar and AST redesign may then be desirable.

To illustrate the process of AST design, we consider the language Cicada, which offers *c*onstants, *i*dentifiers, *c*onditionals, *a*ssignment, *d*ereferences, and *a*rithmetic expressions. While this language is relatively simple, its components are found in most programming languages; moreover, issues that arise in real AST design are present in Cicada.

```
 1  Start  →  Stmt $
 2  Stmt   →  L assign E
 3         |  E
 4         |  if ( E ) Stmt fi
 5         |  if ( E ) Stmt else Stmt fi
 6  L      →  id
 7         |  deref R
 8  R      →  L
 9         |  num
10         |  addr L
11  E      →  E plus T
12         |  T
13  T      →  T times F
14         |  F
15  F      →  ( E )
16         |  R
```

Figure 7.13: Infix assignment and expression grammar.

7.5.1 Devising an unambiguous grammar.

The defining grammar for Cicada is shown in Figure 7.13. This grammar is already unambiguous and is suitable for LALR(1); however, it is not suitable for LL(1) parsing because of its left-recursion. Strings derived from the nonterminal E are in the language of standard expressions, as discussed in Chapter Chapter:global:six. We add assignment expressions and conditionals through Rules 2, 4, and 5. Rule 3 allows a statement to resolve to an expression, whose value in this language is ignored.

Left and right values

It is a common convention in programming languages for an assignment statement $x \leftarrow y$ to have the effect of storing the *value* of variable y at *location* x: the meaning of an identifier differs across the assignment operator. On the left-hand side, x refers to the address at which x is stored; on the right-hand side, y refers to the current value stored in y. Programming languages are designed with this disparity in semantics because the prevalent case is indeed $x \leftarrow y$; consistent semantics would require special annotation on the left- or right-hand side, which would render programs difficult to read. Due to the change in meaning on the left- and right-hand sides of the assignment operator, the location and value of a variable are called the variable's **left value** (Lvalue) and **right value** (Rvalue), respectively. In Figure 7.13, the rules involving nonterminals L and R derive syntactically valid Lvalues and Rvalues:

- An assignment is generated by Rule 2, which expects an Lvalue prior to the assignment operator and an expression afterwards.
- An Lvalue is by definition any construct that can be the target of an assignment. Thus, Rule 6 defines the prevalent case of an identifier as the target of assignment.

  Some languages allow addresses to be formed from values, introduced by a deferencing operator ($\star$ in C). Rule 7 can therefore also serve as an Lvalue, with the semantics of assignment to the address formed from the value R.
- The nonterminal R appears in Figure 7.13 wherever an Rvalue is expected. Semantically, an Lvalue can serve as an Rvalue by implicit dereference, as is the case for y in $x \leftarrow y$. Programming languages are usually designed to obviate the need for explicit syntax to dereference y. Transformation of an Lvalue into an Rvalue is accommodated by Rule 8. Additionally, constructs such as constants and addresses—which cannot be Lvalues—are explicitly included as Rvalues by Rules 9 and 10.

  Although the dereference operator appears syntactically in Rule 7, code generation will call for dereferencing only when Rule 8 is applied. This point is emphasized in Exercise 9.
- Expressions are generated by Rule 3, with multiplication having precedence over addition. An expression (E) is a sum of one or more terms (T); a term is the product of one or more factors (F). Finally, a factor can derive either a parenthesized expression or an Rvalue. Rule 15 provides syntax to alter the default evaluation order for expressions. Rule 16 allows F to generate any construct that has an Rvalue.

7.5.2 Designing the AST

As a first attempt at an AST design, it is helpful to expose unnecessary detail by examining concrete syntax trees such as the one shown in Figure 7.14. We examine each syntactic construct and determine which portions are relevant to the meaning of a Cicada program.

Consistency

It is important that each node type in the AST have consistent meaning. Returning to our Lvalue and Rvalue discussion, an AST node representing an identifier should have consistent meaning, regardless of its occurrence in the AST. It is easy to dereference an address to obtain the current value at that address; it is not always possible to go the other way. We therefore adopt the convention that an identifier node always means the Lvalue of the identifier; if a value is desired, then the AST will require a deference operator.

Consider the assignment $x \leftarrow y$. The AST we desire for this fragment is shown in Figure 7.15. The assignment AST node is therefore defined as a binary node

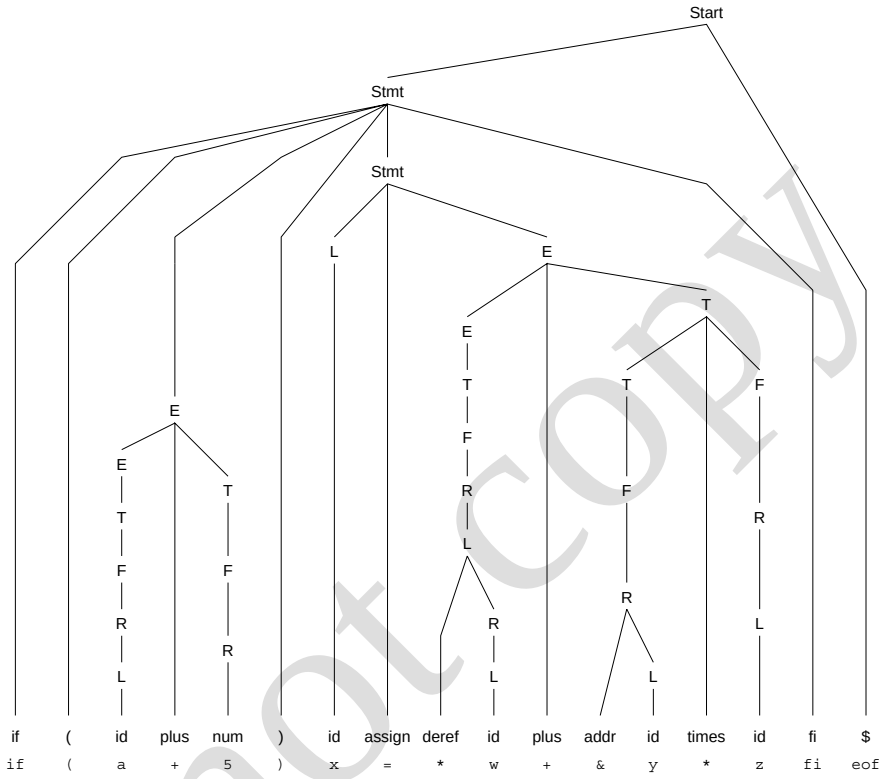

Figure 7.14: Parse tree for a Cicada program.

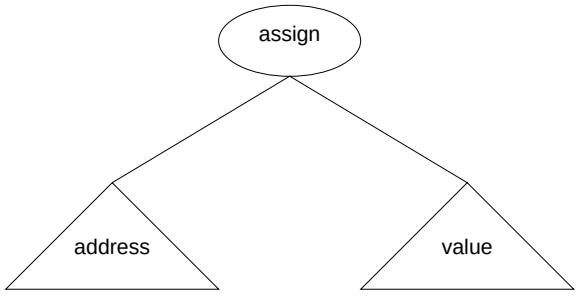

Figure 7.15: AST for a simple assignment.

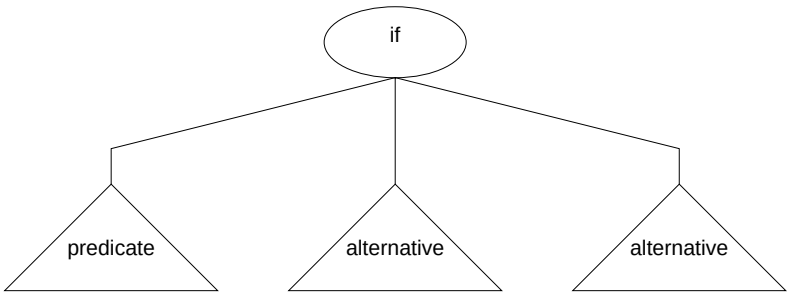

Figure 7.16: AST for a conditional statement.

(two children): its left child evaluates to the address at which the assignment should store the value obtained by evaluating the right child. In Chapter Chapter:global:codegen, code generation for such subtrees will rely upon this AST design; consistency of design will allow the same code sequence to be generated each time an identifier node is encountered.

Cicada has two conditional statements, defined by Rules 4 and 5. Syntactically it is expedient to offer both forms, so that programmers need not supply an empty alternative when one is not needed. However, for consistency it may be desirable to summarize both forms of conditional statement by a single AST construct, whose alternative under a false predicate is empty.

Elimination of detail

Programming languages often contain syntactic components that serve aesthetic or technical purposes without adding any semantic value. For example, parentheses surround predicates in `if` and `while` statements in C and Java. These serve to punctuate the statement, and the closing parenthesis is needed to separate the conditional from the rest of the `if` statement; in Pascal, the keyword `then` serves this same purpose. As shown in Figure 7.16, The AST design for conditionals in Cicada retains only the predicate and the subtree for the branch executed when the predicate holds; as discussed above, a single AST construct represents the semantics of Rules 4 and 5.

Given our convention that identifiers are always interpreted as Lvalues in the AST, the address operator's appearance in an AST through Rule 10 would be redundant. Our AST design therefore does not use the address operator. Similarly, the parentheses surrounding an expression for Rule 15 are unnecessary for grouping, since the AST we create implies an evaluation order for expressions.

Unnecessary detail can occur in concrete syntax trees for the purpose of structure and disambiguation in the grammar. The rules for E, T, and F enforce the precedence of dereference over multiplication over addition; the rules also structure the parse so that arithmetic operators are left-associating. As a result, the

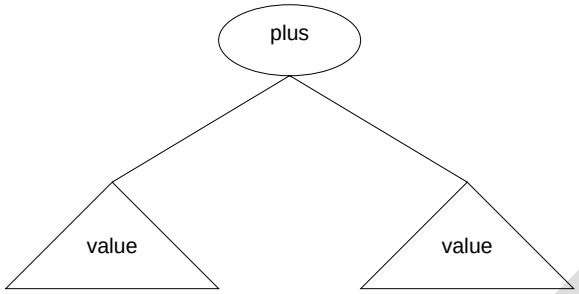

Figure 7.17: AST for E→E plus T.

grammar contains a number of *unit productions*: rules of the form A→B that supply no semantic content. Rules such as E→T escape the recursion on E, but the value under the E is identical to the value under the T. Moreover, the concrete subtree for E→E plus T contains detail in excess of the sum of two values. As shown in Figure 7.17, the AST for this construct retains only the operator and the two values to be summed. The other arithmetic operators for Cicada are similarly accommodated.

7.5.3 Constructing the AST

After designing the AST, semantic actions need to be installed into the parser to construct the vertices and edges that comprise the tree. It is often useful at this point to reconsider the grammar with an eye toward simplifying AST construction. In Cicada we have decided to represent Rules 4 and 5 with the single AST construct shown in Figure 7.16. If we insert semantic actions into the grammar of Figure 7.13 we would find that Rules 4 and 5 have much of their semantic code in common. It is therefore convenient to rewrite Stmt as shown in Figure 7.18: a new nonterminal CondStmt generates the two statements previously contained directly under Stmt. No change to the language results from this modification; however, the semantic actions needed to construct a conditional AST node can be confined to Stmt→CondStmt if we arrange for each rule for CondStmt to provide a predicate and two alternatives for conditional execution.

An efficient AST data structure

While any tree data structure can in theory represent an AST, a better design can result by observing the following:

- The AST is typically constructed bottom-up. A list of siblings is generated, and the list is later adopted by a parent. The AST data structure should therefore support siblings of temporarily unknown descent.

```
1   Start              →  Stmt_ast $
                          return (ast)
2   Stmt_result        →  L_target assign E_source
                          result ← MakeFamily(assign, source, target)
3                     |   E_expr
                          result ← expr
4                     |   CondStmt_triple
                          result ← MakeFamily(if, triple.pred, triple.t, triple.f)
5   CondStmt_triple    →  if ( E_expr ) Stmt_stmt fi
                          triple.pred ← expr
                          triple.t ← stmt
                          triple.f ← MakeConstNode(0)                          7
6                     |   if ( E_expr ) Stmt_s1 else Stmt_s2 fi
                          triple.pred ← expr
                          triple.t ← s1
                          triple.f ← s2
7   L_result           →  id_name
                          result ← MakeIdNode(name)
8                     |   deref R_val
                          result ← val                                         8
9   R_result           →  L_val
                          result ← MakeFamily(deref, val)                      9
10                    |   num_val
                          result ← MakeConstNode(val)
11                    |   addr L_val
                          result ← val                                        10
12  E_result           →  E_v1 plus T_v2
                          result ← MakeFamily(plus, v1, v2)
13                    |   T_v
                          result ← v
14  T_result           →  T_v1 times F_v2
                          result ← MakeFamily(times, v1, v2)
15                    |   F_v
                          result ← v
16  F_result           →  ( E_v )
                          result ← v
17                    |   R_v
                          result ← v
```

Figure 7.18: Semantic actions for AST creation in Cicada.

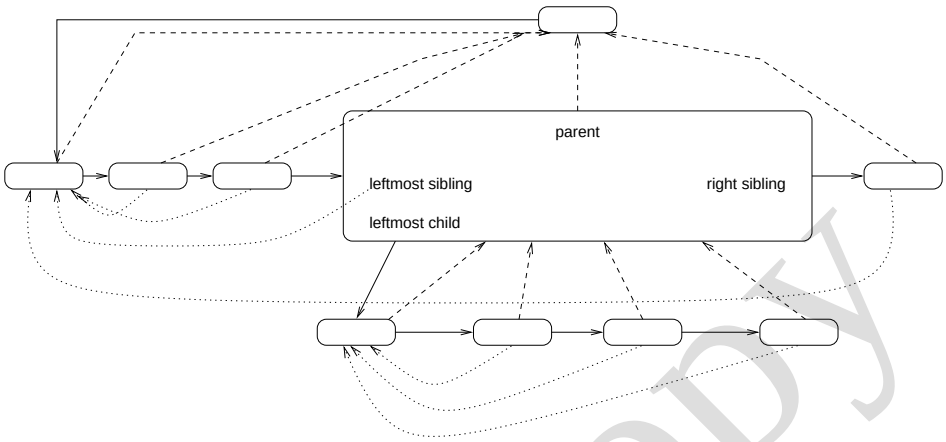

Figure 7.19: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

- Lists of siblings are typically generated by recursive rules. The AST data structure should simplify adoption of siblings at either end of a list.
- Some AST nodes have a fixed number of children. For example, binary addition and multiplication require two children; the conditional AST node developed for Cicada requires three children. However, some programming language constructs may require an arbitrarily large number of children. Such constructs include compound statements, which accommodate zero or more statements, and method parameter and argument lists. The data structure should efficiently support tree nodes with arbitrary degree.

Figure 7.19 depicts an organization for an AST data structure that accommodates the above with a constant number of fields per node. Each node contains a reference to its parent, its next (right) sibling, its leftmost child, and its leftmost sibling.

Infrastructure for creating ASTs

To facilitate construction of an AST we rely upon the following set of constructors and methods to create and link AST nodes. create and link AST nodes. methods for creating and linking AST nodes:

MAKECONSTNODE(*val*) creates a node that represents the integer constant *val*.

MAKEIDNODE(*id*) creates a node for the identifier *id*. During symbol-table construction (Chapter Chapter:global:8), this node is often replaced by a node

that references the appropriate symbol table entry. This node refers to the Lvalue of *id*.

MAKEOPERATORNODE(*oper*) creates an AST node that represents the operation *oper*. It is expedient to specify *oper* as an actual terminal symbol in the grammar. For Cicada, *oper* could be plus, times, or deref.

x.MAKESIBLINGS(*y*) causes *y* to become *x*'s right sibling. To facilitate chaining of nodes through recursive grammar rules, this method returns a reference to the rightmost sibling resulting from the invocation.

x.HEAD( ) returns the leftmost sibling of *x*.

x.ADOPTCHILDREN(*y*) adopts *y* and its siblings under the parent *x*.

MAKEFAMILY($op, kid_1, kid_2, \ldots, kid_n$) generates a family with exactly *n* children under the operation *op*. The code for the most common case ($n = 2$) is:

```
function MAKEFAMILY(op, kid1, kid2) : node
    kids ← kid1.MAKESIBLINGS(kid2)
    parent ← MAKEOPERATORNODE(op)
    call parent.ADOPTCHILDREN(kids)
    return (parent)
end
```

The semantic actions for constructing a Cicada AST are shown in Figure 7.18. The prevalent action is to glue siblings and a parent together via MAKEFAMILY; conditionals and dereferences are processed as follows:

- The conditional statements at Rules 5 and 6 generate three values that are passed to Rule 4. These values are consistently supplied as the predicate and the two alternatives of the if statement.
- Step **7** (arbitrarily) synthesizes the constant 0 for the missing else alternative of Rule 5. The AST design should be influenced by the source language's definition, which should specify the particular value—if any—that is associated with a missing else clause.
- Step **10** executes when an object's address is taken. However, the actual work necessary to perform this task is less than one might think. Instead of expending effort to take an object's address, the addr operator actually *prevents* the dereference that would have been performed had the id been processed by the rule R→L.
- Step **8** represents a syntactic dereference. However, the AST does not call for a dereference, and the subtree corresponding to the dereferenced Rvalue is simply copied as the result. While this may seem a mistake, the actual dereference occurs further down the tree, when an Lvalue becomes an Rvalue, as described next.

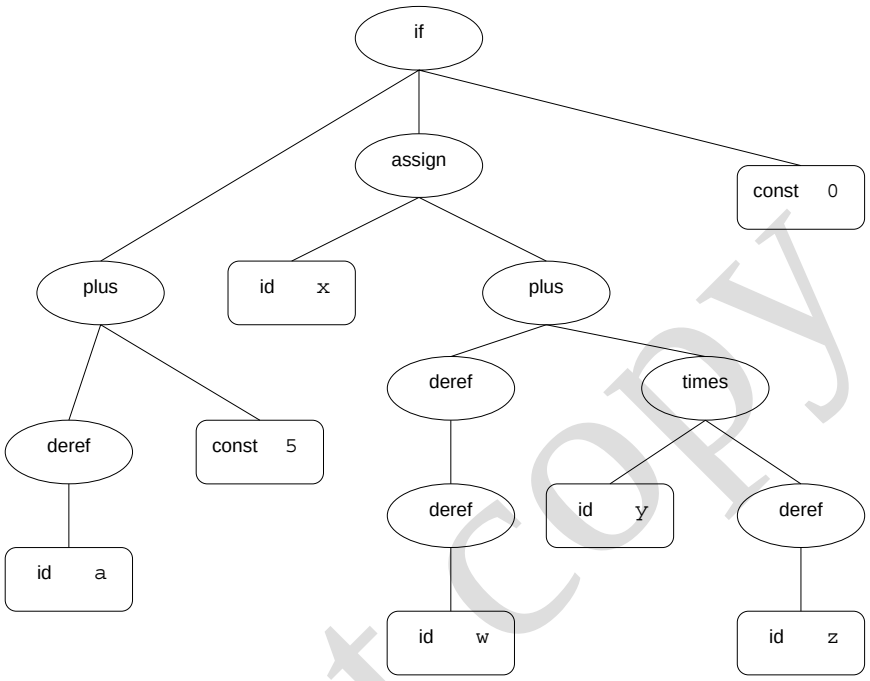

Figure 7.20: AST for the Cicada program in Figure 7.14.

- Step **9** generates an AST node that represents an actual dereference, in response to the transformation of an Lvalue into an Rvalue.
- The AST is complete at Rule 1; the root is contained in the semantic value *ast*.

Figure 7.20 shows the AST that results from the parse of Figure 7.14. The double dereference of `w` occurs in response to the input string's "$\star$ `w`", used as an Rvalue. Indeed, the tree shown in Figure 7.14 contains two R$\rightarrow$L productions above the terminal `w`. The use of `y` without dereference in Figure 7.20 occurs because of the `addr` operator in Figure 7.14.

List processing using left-recursive rules

A common idiom in grammars and semantic actions is the processing of a list of grammar symbols via a recursive production. In bottom-up parsing, such rules are usually left-recursive, which minimizes stack usage and creates the prevalent left-association. In top-down parsing, such rules are necessarily right-recursive.

To illustrate how such left- or right-recursive lists can be distilled into an AST we recall the grammar from Section 7.3 that defined base-10 numbers. In Figure 7.2, the semantic actions compute the value of a number. Figure 7.21 reprises

```
1  Start_ast  →  Digs_kids $
                    ast ← MAKEOPERATORNODE( Digs )
                    call ast.ADOPTCHILDREN( kids )
2  Digs_up    →  Digs_below d_next
                    sib ← MAKECONSTNODE( next )                    11
                    up ← below.MAKESIBLINGS( sib )                 12
3             |  d_first
                    up ← MAKECONSTNODE( first )
```

Figure 7.21: AST construction for a left-recursive rule.

the grammar, but with the goal of generating an AST to represent the list of digits. The utility method MAKESIBLINGS was carefully defined to return the rightmost of a list of siblings. Thus, Step **12** returns the most recently processed digit, made into an AST node by Step **11**.

Summary

Syntax-directed translation can accomplish translation directly via semantic actions that execute in concert with top-down or bottom-up parses. More commonly, an AST is constructed as a by-product of the parse; the AST serves as a record of the parse as well as a repository for information created and consumed by a compiler's passes. The design of an AST is routinely revised to simplify or facilitate compilation.

Exercises

1. Consider a right-recursive formulation for Digs of Figure 7.2, resulting in the following grammar.

```
1  Start      →  Digs_ans $
                    call PRINT(ans)
2  Digs_up    →  d_next Digs_below
                    up ← below × 10 + next
3             |  d_first
                    up ← first
```

   Are the semantic actions still correct? If so, explain why they are still valid; if not, provide a set of semantic actions that does the job properly.

2. The semantic actions in Figure 7.4 fail to ensure that digits interpreted base-8 are in the range $0 \ldots 7$. Show the semantic actions that need to be inserted into Figure 7.4 so that digits that lie outside the octal range are effectively ignored.

3. The grammar in Figure 7.8 is almost faithful to the language originally defined in Figure 7.6. To appreciate the difference, consider how each grammar treats the input string `x 5 $`.

   (a) In what respects do the grammars generate different languages?

   (b) Modify the grammar in Figure 7.8 to maintain its style of semantic processing but to respect the language definition in Figure 7.6.

4. The language generated by the grammar in Figure 7.8 uses the terminal x to introduce the base. A more common convention is to separate the base from the string of digits by some terminal symbol. Instead of `x 8 4 3 1` to represent 431_8, a language following the common convention would use `8 x 4 3 1`.

   (a) Design an LALR(1) grammar for such a language and specify the semantic actions that compute the string's numeric value. In your solution, allow the absence of a base specification to default to base 10, as in Figure 7.8.

   (b) Discuss the tradeoffs of this language and your grammar as compared with the language and grammar of Figure 7.8.

5. Consider the addition of the the rule

   Expr → sum Values

   to the grammar in Figure 7.9.

(a) Does this change make the grammar ambiguous?

(b) Is the grammar still LL(1) parsable?

(c) Show how the semantic actions in Figure 7.10 must be changed to accommodate this new language construct; modify the grammar if necessary but avoid use of global variables.

6. Consider the addition of the rule

$$\text{Expr} \rightarrow \text{mean Values}$$

to the grammar in Figure 7.9. This rule defines an expression that computes the average of its values, defined by:

$$\texttt{(mean } v_1 v_2 \ldots v_n) = \left\lfloor \frac{\sum_{i=1}^{n} v_i}{n} \right\rfloor$$

(a) Does this render the grammar ambiguous?

(b) Is the grammar still LL(1) parsable?

(c) Describe your approach for syntax-directed translation of this new construct.

(d) Modify the grammar of Figure 7.10 accordingly and insert the appropriate semantic actions into Figure 7.10.

7. Although arithmetic expressions are typically evaluated from left to right, the semantic actions in VALUES cause a product computed from right to left. Modify the grammar and semantic actions of Figures 7.9 and 7.10 so that products are computed from left to right.

8. Verify that the grammar in Figure 7.13 is unambiguous using an LALR(1) parser generator.

9. Suppose that the terminals assign, deref, and addr correspond to the input symbols =, $\star$, and &, respectively. Using the grammar in Figure 7.13

- show parse trees for the following strings;
- show where indirections actually occur by circling the parse tree nodes that correspond to the rule R → L.

(a) `x = y`

(b) $\star$ `x = y`

(c) $\star$ `x = y`

(d) $\star$ `x =` $\star$ `y`

(e) $\star$ $\star$ `x = & y`

(f) $\star$ `16 = 256`

10. Construct an LL(1) grammar for the langauge in Figure 7.13.

11. Suppose Cicada were extended to include binary subtraction and unary negation, so that the expression `minus y minus x times 3` has the effect of negating `y` prior to subtraction of the product of `x` and `3`.

    (a) Following the steps outlined in Section 7.5, modify the grammar to include these new operators. Your grammar should grant negation strongest precedence, at a level equivalent to deref. Binary subtraction can appear at the same level as binary addition.

    (b) Modify the chapter's AST design to include subtraction and negation by including a minus operator node.

    (c) Install the appropriate semantic actions into the parser.

12. Modify the grammar and semantic actions of Figure 7.21 so that the rule for Digs is right-recursive. Your semantic actions should create the identical AST.

13. Using a standard LALR(1) grammar for C or Java, find syntactic punctuation that can be eliminated without introducing LALR(1) conflicts. Explain why the (apparently unnecessary) punctuation is included in the language. As a hint, consider the parentheses that surround the predicate of an `if` statement.