Chapter 7
TRANSLATIONAL SEMANTICS

The previous chapter provided a definition of the semantics of a programming language in terms of the programming language itself. The primary example was based on a Lisp interpreter programmed in Lisp. Although this is an interesting academic exercise, it has little practical importance. However, programming language compilers are an integral part of the study of computer science. Compilers perform a translation of a high-level language into a low-level language. Executing this target program on a computer captures the semantics of the program in a high-level language.

In Chapter 3 we investigated the use of attribute grammars for context checking. These same techniques can be used to translate Wren into a machine-oriented language. Translational semantics is based on two notions:

1. The semantics of a programming language can be preserved when the language is translated into another form, called the target language.

2. The target language can be defined by a small number of primitive constructs that are closely related to an actual or a hypothetical machine architecture.

We first introduce the target language and then build an attribute grammar that is capable of translating a Wren program into this language. Finally, we implement this attribute grammar in the laboratory.

7.1 CONCEPTS AND EXAMPLES

In order to focus on issues relating to the translation process, we assume that the Wren program being translated obeys the context-sensitive conditions for the language as well as the context-free grammar. We parse the declaration section to ensure that the BNF is correct, but no attributes are associated with the declaration section. Context checking can be combined with code generation in a single attribute grammar, but we leave this task unfinished at this time.

The machine code is based on a primitive architecture with a single accumulator (Acc) and a memory addressable with symbolic labels and capable of holding integer values. In this translation, Boolean values are simulated by integers. We use names to indicate symbolic locations. The hypothetical machine has a load/store architecture:

- The LOAD instruction copies a value from a named location, whose value is not changed, to the accumulator, whose old value is overwritten, or transfers an integer constant into the accumulator.
- The STO instruction copies the value of the accumulator, whose value is not changed, to a named location, whose previous value is overwritten.

The target language has two input/output commands:

- GET transfers an integer value from the input device to the named location.
- PUT transfers the value stored at the named location to the output device.

There are four arithmetic operations—ADD, SUB, MULT and DIV—and three logical operations—AND, OR, and NOT. For the binary operations, the first operand is the current accumulator value and the second operand is specified in the instruction itself. The second operand can be either the contents of a named location or an integer constant. For Boolean values, the integer 1 is used to represent true and the integer 0 to represent false. The result of an operation is placed in the accumulator. The NOT operation has no argument; it simply inverts the 0 or 1 in the accumulator.

The target language contains one unconditional jump J and one conditional jump JF where the conditional jump is executed if the value in the accumulator is false (equal to zero). The argument of a jump instruction is a label instruction. For example, J L3 means to jump unconditionally to label L3, which appears in an instruction of the form L3 LABEL. The label instruction has no operand.

There are six test instructions; they test the value of the accumulator relative to zero. For example, TSTEQ tests whether the accumulator is equal to zero. The test instructions are destructive in the sense that the value in the accumulator is replaced by a 1 if the test is true and a 0 if the test is false. We will find this approach to be convenient when processing Boolean expressions. The five other test instructions are: TSTLT (less than zero), TSTLE (less than or equal zero), TSTNE (not equal zero), TSTGE (greater than or equal zero), and TSTGT (greater than zero). The NO-OP instruction performs no operation. Finally, the target language includes a HALT instruction. The complete instruction set is shown in Figure 7.1.

LOAD	<name> or <const>	Load accumulator from named location or load constant value
STO	<name>	Store accumulator to named location
GET	<name>	Input value to named location
PUT	<name>	Output value from named location
ADD	<name> or <const>	Acc ← Acc + <operand>
SUB	<name> or <const>	Acc ← Acc – <operand>
MULT	<name> or <const>	Acc ← Acc • <operand>
DIV	<name> or <const>	Acc ← Acc / <operand>
AND	<name> or 0 or 1	Acc ← Acc and <operand>
OR	<name> or 0 or 1	Acc ← Acc or <operand>
NOT		Acc ← not Acc
J	<label>	Jump unconditionally
JF	<label>	Jump on false (Acc = 0)
LABEL		Label instruction
TSTLT		Test if Acc **L**ess **T**han zero
TSTLE		Test if Acc **L**ess than or **E**qual zero
TSTNE		Test if Acc **N**ot **E**qual zero
TSTEQ		Test if Acc **EQ**ual zero
TSTGE		Test if Acc **G**reater than or **E**qual zero
TSTGT		Test if Acc **G**reater **T**han zero
NO-OP		No operation
HALT		Halt execution

Figure 7.1: Machine-oriented Target Language

A Program Translation

Consider a greatest common divisor (gcd) program:

```
program gcd is
    var m,n : integer;
begin
    read m; read n;
    while m < > n do
            if m < n  then n := n – m
                      else m := m – n
            end if
    end while;
    write m
end
```

This program translates into the following object code:

```
    GET    M
    GET    N
L1  LABEL
    LOAD   M
```

```
     SUB    N
     TSTNE
     JF     L2
     LOAD   M
     SUB    N
     TSTLT
     JF     L3
     LOAD   N
     SUB    M
     STO    N
     J      L4
L3   LABEL
     LOAD   M
     SUB    N
     STO    M
L4   LABEL
     J      L1
L2   LABEL
     LOAD   M
     STO    T1
     PUT    T1
     HALT
```

In Chapter 3 we saw that the semantics of a binary numeral can be expressed as the final value of a synthesized attribute at the root of the parse tree. We use the same approach here: The synthesized attribute *Code* integrates the pieces of object code in the target language from lower levels in the tree, and the final value at the root of the tree expresses the semantics of the Wren program in the form of its translation into object code in the target language.

We begin by discussing the constraints imposed by labels and temporary locations in the target language. Labels throughout the object program must be unique. With nested control structures, the labels do not appear in order, as we see from the sample program above. The labels L1 and L2 are associated with the **while** loop, the outer control structure, and the labels L3 and L4 are associated with the nested **if** structure. We use both an inherited attribute *InhLabel* and a synthesized attribute *SynLabel* working together to thread the current label value throughout the derivation tree.

The intermediate language uses temporary named locations, labeled T1, T2, T3, and so forth, to evaluate Boolean and arithmetic expressions. For our purposes, it is immaterial if these named locations are thought of as registers or as an area of main memory. One location must not be used for two

different purposes at the same time within a subexpression, but it can be reused once the final value for a subexpression has been processed. Since temporary locations need not be unique throughout the program, there is no need to maintain a synthesized attribute returning the last location used. However, we do need an inherited attribute, called *Temp*, that transfers the starting temporary location value to subexpressions.

Exercises

1. Generate object code that is semantically equivalent to the following Wren program that multiplies two numbers.

```
program multiply is
  var m, n, product : integer;
begin
  read m;  read n;
  product := 0;
  while n > 0 do
        if 2 * (n / 2) < > n then  (* if n is odd *)
                product := product + m
        end if;
        m := 2 * m;
        n := n / 2
  end while;
  write product
end
```

2. Generate object code that evaluates the following expression:

```
2 * (x – 1) * ( y / 4 ) – ( 12 * z + y )
```

7.2 ATTRIBUTE GRAMMAR CODE GENERATION

Figure 7.2 gives all of the attributes and the associated value types necessary to develop this attribute grammar. The nonterminals <variable>, <identifier>, <letter>, <numeral>, and <digit> all have an associated *Name* attribute that synthesizes an internal representation of identifier names and of base-ten numerals, as was done in the attribute grammar for Wren in Chapter 3. Since the source language uses lowercase letters as variable names, and the target language uses uppercase, we make a conversion between the two in the production for <letter>.

Attribute	Value
Name	Sequences of letters and/or digits
Temp	Natural numbers (integers ≥ 0)
SynLabel	Natural numbers
InhLabel	Natural numbers
OpCode	ADD, SUB, MULT, DIV
TestCode	TSTLT, TSTLE, TSTNE, TSTEQ, TSTGE, TSTGT
Code	Sequence of instructions of the following forms:
	(Load/Store, Name) as in (LOAD, X)
	(Input/Output, Name) as in (GET, X)
	(OpCode, Name) as in (ADD, 5)
	(BooleanOp, Name) as in (AND, T2)
	(Jump, Name) as in (J, L2)
	(Name, LABEL) as in (L3, LABEL)
	TestCode as in TSTNE
	NOT, NO-OP, or HALT

Figure 7.2: Attributes and Values

<variable> ::= <identifier>

Name(<variable>) ← *Name*(<identifier>)

<identifier> ::=

<letter>

Name(<identifier>) ← *Name*(<letter>)

| <identifier>$_2$ <letter>

Name(<identifier>) ← concat(*Name*(<identifier>$_2$),*Name*(<letter>))

| <identifier>$_2$ <digit>

Name(<identifier>) ← concat(*Name*(<identifier>$_2$),*Name*(<digit>))

<letter> ::=

a

Name(<letter>) ← 'A'

: : :

| **z**

Name(<letter>) ← 'Z'

<numeral> ::= <digit>

Name(<numeral>) ← *Name*(<digit>)

| <numeral>$_2$ <digit>

Name(<numeral>) ← concat(*Name*(<numeral>$_2$),*Name*(<digit>))

<digit> ::=
    0
        Name(<digit>) ← '0'
            :   :   :
    | **9**
        Name(<digit>) ← '9'

Expressions

We now turn our attention to the code generation for binary arithmetic expressions. Consider the general form

<left operand> <operator> <right operand>

where the left and right operands may be simple, as in the case of a variable name or numeral, or compound, as in the case of another operation or a parenthesized expression. The general case can be handled by the following code sequence, assuming that n is the value of the inherited attribute *Temp*:

```
code for <left operand>
STO       T<n+1>          (for example, if n = 0, this is T1)
code for <right operand>
STO       T<n+2>          (for example, if n = 0, this is T2)
LOAD      T<n+1>
OpCode    T<n+2>
```

In this situation, OpCode is determined by the <operator>. The inherited value for *Temp* is passed to the left operand while that value, incremented by one, is passed to the right operand, since the location T<n+1> is not available for use when generating code for the right operand. In general, the value of *Temp* represents the highest value used so far in the current subexpression.

As an example of translating expressions, consider the compound expression

x / (y – 5) * (z + 2 * y).

The expression expands to <term> that then expands to <term> <strong op> <element>. So, assuming *Temp* initially equals zero, the code expansion becomes

```
code for x/(y – 5)
STO    T1
code for (z + 2 * y)
STO    T2
LOAD   T1
MULT   T2
```

We show object code in bold when it first appears. *Temp* is passed unchanged to the code for the left operand and is incremented by 1 and passed to the right operand. When we expand the code for x/(y – 5) we have

```
code for x
STO   T1
code for (y – 5)
STO   T2
LOAD  T1
DIV   T2
STO   T1
code for (z + 2 * y)
STO   T2
LOAD  T1
MULT  T2
```

The code for x is LOAD X, so we proceed with the expansion of (y – 5) with *Temp* equal to 1, obtaining

```
LOAD  X
STO   T1
code for y
STO   T2
code for 5
STO   T3
LOAD  T2
SUB   T3
STO   T2
LOAD  T1
DIV   T2
STO   T1
code for (z + 2 * y)
STO   T2
LOAD  T1
MULT  T2
```

The code for y and for 5 is LOAD Y and LOAD 5, respectively. We now need to expand the code for (z + 2 * y) with *Temp* equal to 1.

```
LOAD  X
STO   T1
LOAD  Y
STO   T2
LOAD  5
```

```
STO   T3
LOAD  T2
SUB   T3
STO   T2
LOAD  T1
DIV   T2
STO   T1
code for z
STO   T2
code for 2*y
STO   T3
LOAD  T2
ADD   T3
STO   T2
LOAD  T1
MULT  T2
```

The code for z is LOAD Z. When we expand the code for 2*y, we use a *Temp* value of 2, the inherited value incremented by 1. We complete the code by using LOAD 2 and LOAD Y as the code for 2 and code for y, respectively. The complete code expansion is shown below.

```
LOAD  X
STO   T1
LOAD  Y
STO   T2
LOAD  5
STO   T3
LOAD  T2
SUB   T3
STO   T2
LOAD  T1
DIV   T2
STO   T1
LOAD  Z
STO   T2
LOAD  2
STO   T3
LOAD  Y
STO   T4
LOAD  T3
MULT  T4
```

```
STO     T3
LOAD    T2
ADD     T3
STO     T2
LOAD    T1
MULT    T2
```

If the result of this expression, which is currently in the accumulator, is to be saved, one more store instruction will be needed. The code generated in this way is correct but very lengthy. A human hand-generating code for the same expression can do much better. Working “from the inside out” and taking advantage of known arithmetic properties, such as the commutativity of addition, a human might produce the following code sequence:

```
LOAD    Y
SUB     5
STO     T1          -- T1 contains y – 5
LOAD    X
DIV     T1
STO     T1          -- T1 contains x/(y – 5)
LOAD    2
MULT    Y
ADD     Z
MULT    T1          -- accumulator contains x / (y – 5) *  (z + 2 * y)
```

Only ten instructions and one temporary location are needed, as compared with 26 instructions and four temporary locations for the code developed previously. We do not attempt to match hand-compiled code generated by a human for efficiency; however, there is one small optimization we can make that improves the code generation. Consider the special case

<left operand>   <operator>   <variable or numeral>

If we follow the previous scheme, the generated code is

```
code for <left operand>
STO       T<n+1>
code for <variable or numeral>
STO       T<n+2>
LOAD      T<n+1>
OpCode    T<n+2>
```

When the second operand is a variable or numeral, this code can be optimized to

```
code for <left operand>
OpCode    <variable or numeral>
```

This saves four instructions and two temporary locations. This code pattern occurs twice in the expression we were evaluating. The code for `y-5`

```
LOAD  Y      becomes     LOAD  Y
STO   T2                 SUB   5
LOAD  5
STO   T3
LOAD  T2
SUB   T3
```

The code for `2*y`

```
LOAD  2      becomes     LOAD  2
STO   T3                 MULT  Y
LOAD  Y
STO   T4
LOAD  T3
MULT  T4
```

When this one optimization technique is used, the code for the expression is reduced to 18 instructions and three temporary locations, as shown below.

```
LOAD  X
STO   T1
LOAD  Y
SUB   5
STO   T2
LOAD  T1
DIV   T2
STO   T1
LOAD  Z
STO   T2
LOAD  2
MULT  Y
STO   T3
LOAD  T2
ADD   T3
STO   T2
LOAD  T1
MULT  T2
```

Code optimization, a major topic in compiler theory, is very complex and occurs at many levels. A detailed discussion is beyond the scope of this text, and we will do no further optimization beyond this one technique.

We now turn our attention to the attribute grammar itself. Four of the attributes listed in Figure 7.2 are utilized in generating the code for arithmetic expressions: *Name*, *OpCode*, *Code*, and *Temp*.

First consider the attribute grammar for integer expression.

```
<integer expr> ::=
      <term>
         Code(<integer expr>) ← Code(<term>)
         Temp(<term>) ← Temp(<integer expr>)
    | <integer expr>2 <weak op> <term>
         Code(<integer expr>) ←
                  concat(Code(<integer expr>2),
                           optimize(Code(<term>),Temp(<integer expr>),
                                                          OpCode(<weak op>)))
         Temp(<integer expr>2) ← Temp(<integer expr>)
         Temp(<term>) ← Temp(<integer expr>)+1

<weak op> ::=
      +
         OpCode(<weak op>) ← ADD
    | –
         OpCode(<weak op>) ← SUB
```

Temp is inherited, as expected, *OpCode* synthesizes the appropriate object code operation, and *Code* is synthesized, as previously described. However, we need to say something about the utility procedure “optimize”.

```
optimize(code, temp, opcode) =
    if length(code) = 1 then                       -- a variable or numeral
                             [(opcode, secondField(first(code)))]
                         else
                             concat([(STO, temporary(temp+1))],
                                      code,
                                      [(STO, temporary(temp+2))],
                                      [(LOAD, temporary(temp+1))],
                                      [(opcode, temporary(temp+2))])
```

If the code for the second operand is a single item, indicating either a variable or a numeral, we generate a single instruction, the appropriate operation with that operand. Otherwise, we generate the more lengthy set of instructions and use two temporary locations. The utility procedure “temporary” accepts an integer argument and produces the corresponding temporary sym-

bol as a string. This requires another utility function "string" to convert an integer into the corresponding base-ten numeral.

```
temporary(integer) = concat('T',string(integer))
string(n) =  if n = 0 then '0'
                  :          :
             else if n = 9 then '9'
             else concat(string(n/10), string(n mod 10))
```

The code for <term> and <element> is very similar and will be given later in Figure 7.8. The sharp-eyed reader may notice that we have ignored negation as one of the alternatives for <element>. Developing this portion of the attribute grammar is left as an exercise. The complete, decorated parse tree for x / (y – 5) * (z + 2 * y) is shown in Figure 7.3 where we have used an infix operator @ to represent concatenation.

The code for <boolean expr> is similar to integer expression, except that the single operator is **or**. Since the Boolean operations are commutative, there is no need for a second temporary location. A Boolean term is similar to an integer term. In <boolean element> the constants **false** and **true** result in loading 0 and 1, respectively. Variables and parenthesized Boolean expressions are handled as expected. A <comparison> is another alternative for a Boolean element; we will discuss comparisons in a moment. The **not** operation results in the NOT instruction being appended after the code for the Boolean expression being negated. The complete code for Boolean expressions is given in Figure 7.8.

A comparison has the general form

<integer expr>$_1$ <relation> <integer expr>$_2$

Since the target language has test instructions based on comparisons with zero, we rewrite the comparison as

<integer expr>$_1$ – <integer expr>$_2$ <relation> 0

The generated code will be the code defined for the left side expression minus the right side expression followed by the test instruction. This code will be optimized if the right side expression is a constant or a variable. Here is an example with code optimization: x < y translates into

```
LOAD  X
SUB   Y
TSTLT
```

If the right side expression is complex, the code will not be optimized, as seen by x >= 2 * y, which translates to

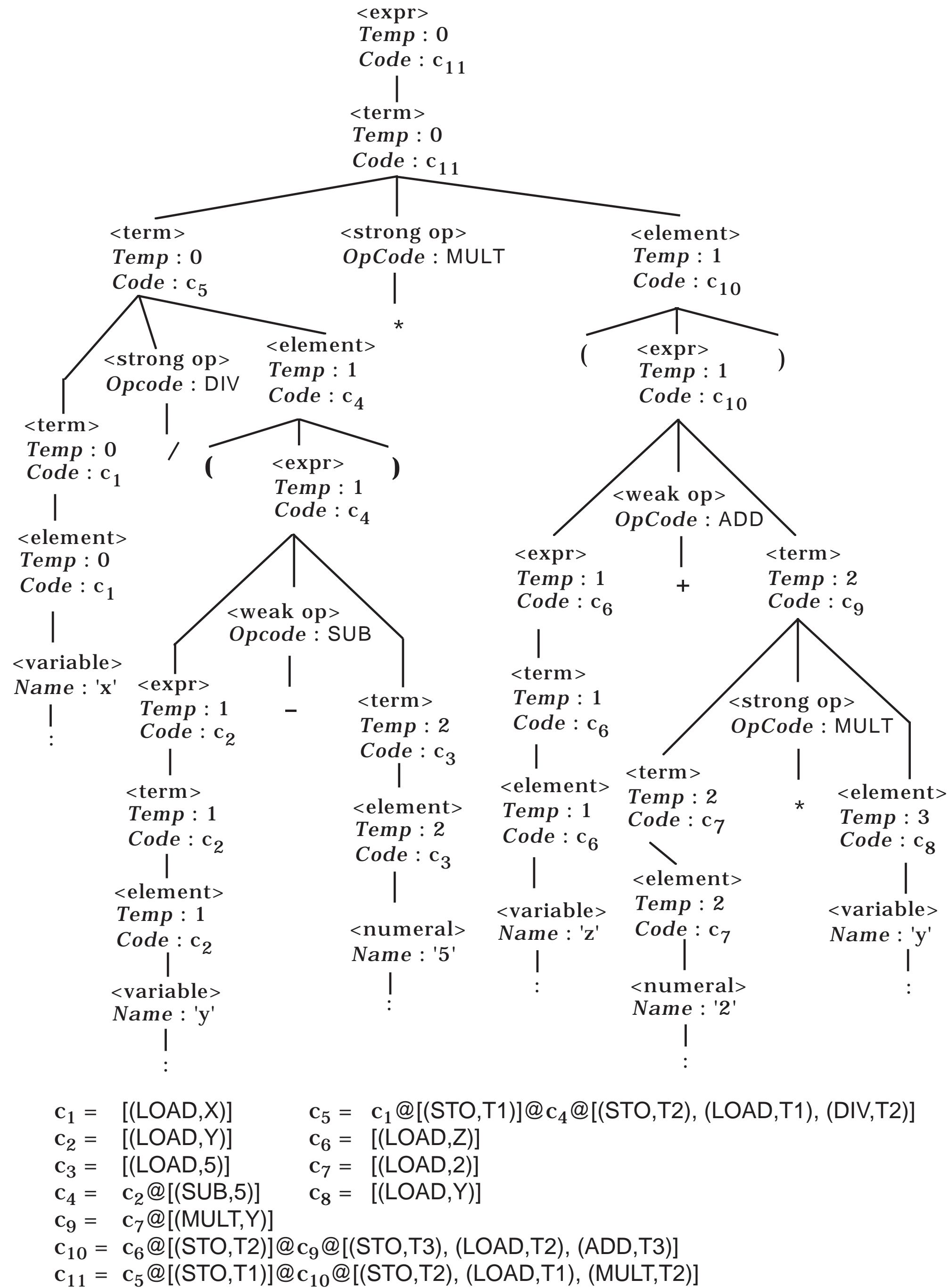

Figure 7.3: Expression Parse Tree for x/(y – 5) * (z + 2 * y)

```
LOAD  X
STO   T1
LOAD  2
MULT  Y
STO   T2
LOAD  T1
SUB   T2
TSTGE
```

There is a direct correspondence between the comparison operators in Wren and the test instructions in the target language:

<	becomes	TSTLT
<=	becomes	TSTLE
=	becomes	TSTEQ
>	becomes	TSTGT
>=	becomes	TSTGE
<>	becomes	TSTNE.

The following attribute grammar rules follow directly from this discussion.

<comparison> ::= <integer expr>$_1$ <relation> <integer expr>$_2$

Code(<comparison>) ← concat(*Code*(<integer expr>$_1$),
optimize(*Code*(<integer expr>$_2$),*Temp*(<comparison>),SUB),
[*TestCode*(<relation>)])

Temp(<integer expr>$_1$) ← *Temp*(<comparison>)

Temp(<integer expr>$_2$) ← *Temp*(<comparison>)+1

<relation> ::=
>
TestCode(<relation>) ← TSTGT
| >=
TestCode(<relation>) ← TSTGE
| <>
TestCode(<relation>) ← TSTNE
| =
TestCode(<relation>) ← TSTEQ
| <=
TestCode(<relation>) ← TSTLE
| <
TestCode(<relation>) ← TSTLT

Commands

The next major task is the generation of code for the commands in Wren. As mentioned earlier, all labeled locations must be unique throughout the ob-

ject code. We use two attributes, *InhLabel* and *SynLabel*, to thread the current label value throughout the tree. The <program> node concatenates HALT to the code generated by <block>. The program identifier provides source code documentation but does not contribute to the code generation.

<program> ::= **program** <identifier> **is** <block>

Code(<program>) ← concat(*Code*(<block>), [HALT])

The code for <block> is synthesized directly from the code for <command sequence>. The inherited attributes *Temp* and *InhLabel* are initialized to zero at this time. Parsing a declaration sequence does not involve any attributes for code generation.

<block> ::= <declaration sequence> **begin** <command sequence> **end**

Code(<block>) ← *Code*(<command sequence>)

Temp(<command sequence>) ← 0

InhLabel(<command sequence>) ← 0

The BNF for <declaration sequence> will be given in Figure 7.8, but no attributes are calculated for this nonterminal. The nonterminal <command sequence> allows two alternatives, a single command or a command followed by a command sequence. The first case, which describes a single command, passes the inherited attributes *Temp* and *InhLabel* to the child and synthesizes the attributes *Code* and *SynLabel* from the child. When the command sequence is a command followed by a second command sequence, the *Code* attributes from each of the children are concatenated and passed up to the parent. The inherited attribute *Temp* passes down to both children. The inherited attribute *InhLabel* is passed down to the first child <command>, synthesized out as *SynLabel*, passed over and inherited into <command sequence>$_2$. Finally *SynLabel* for <command sequence>$_2$ is passed back to <command sequence>. The attribute grammar for a command sequence appears below.

<command sequence> ::=

<command>

Code(<command sequence>) ← *Code*(<command>)

Temp(<command>) ← *Temp*(<command sequence>)

InhLabel(<command>) ← *InhLabel*(<command sequence>)

SynLabel(<command sequence>) ← *SynLabel*(<command>)

| <command> ; <command sequence>$_2$

Code(<command sequence>) ←

concat(*Code*(<command>),*Code*(<command sequence>$_2$))

Temp(<command>) ← *Temp*(<command sequence>)

Temp(<command sequence>$_2$) ← *Temp*(<command sequence>)

InhLabel(<command>) ← *InhLabel*(<command sequence>)

InhLabel(<command sequence>$_2$) ← *SynLabel*(<command>)
SynLabel(<command sequence>) ← *SynLabel*(<command sequence>$_2$)

This threading of label values in and out of commands is important and is illustrated in Figure 7.4.

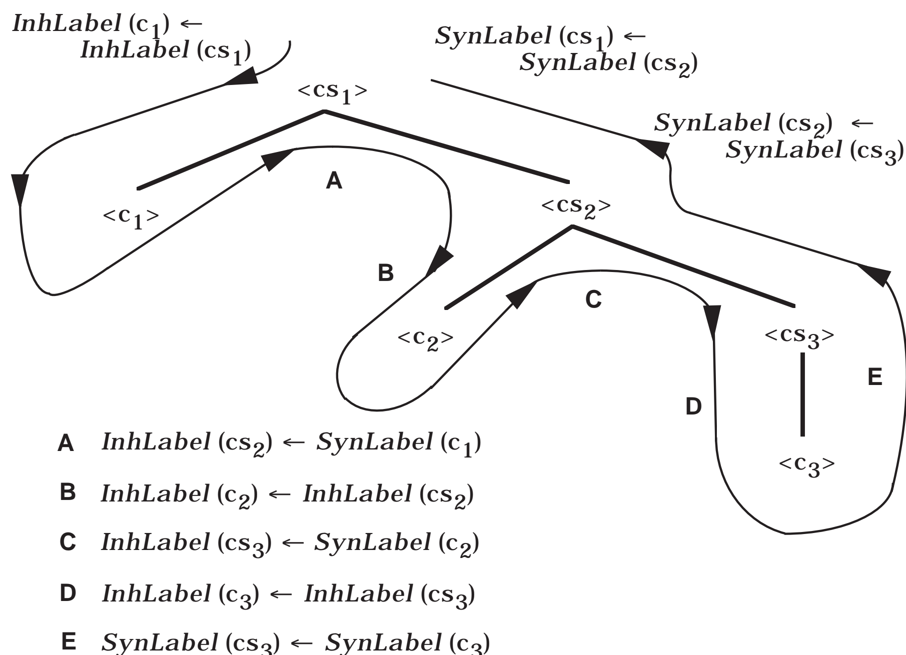

Figure 7.4: Threading of Label Attributes

Some commands, such as assignment, do not affect label values while others, such as **while** or **if**, require one or two label values. The threaded label value is incremented appropriately as it passes through the tree and, in this way, it ensures that all labeled locations are unique throughout the target code.

A command takes one of seven alternatives: input, output, **skip**, assignment, a single alternative **if**, a double alternative **if**, or **while**. Four of the commands—input, output, **skip**, and assignment—do not generate code with labels, so the inherited label value *InhLabel* is "turned around" unmodified and synthesized back as *SynLabel.* The **if** and **while** commands receive the inherited attribute *InhLabel* and synthesize back a different value for *SynLabel.*

The input, output, and **skip** commands are very simple. Input of a variable generates the code of GET followed by the variable name. Output generates code for the expression, stores the result in a temporary location, and then does a PUT of that temporary location. Finally, **skip** generates a NO-OP (no operation) instruction.

<command> ::= **read** <variable>

Code(<command>) ← [(GET, *Name*(<variable>))]

SynLabel(<command>) ← *InhLabel*(<command>)

<command> ::= **write** <integer expr>

Code(<command>) ← concat(*Code*(<integer expr>),
[(STO, temporary(*Temp*(<command>)+1))])
[(PUT, temporary(*Temp*(<command>)+1))])

Temp(<integer expr>) ← *Temp*(<command>)

SynLabel(<command>) ← *InhLabel*(<command>)

<command> ::= **skip**

Code(<command>) ← [NO-OP]

SynLabel(<command>) ← *InhLabel*(<command>)

We have already discussed in detail the code generation for both integer and Boolean expressions. The result of an expression is left in the accumulator, so the assignment command concatenates code to store that result in the target variable name for the assignment. Since an expression may need to use temporary locations, the inherited attribute *Temp* is passed down to <expr>.

<command> ::= <variable> **:=** <expr>

Code(<command>) ←
concat(*Code*(<expr>),[(STO, *Name*(<variable>))])

Temp(<expr>) ← *Temp*(<command>)

SynLabel(<command>) ← *InhLabel*(<command>)

The **while** command has the form

while <boolean expr> **do** <command sequence> **end while**

where the code generated by the Boolean expression is followed by a conditional jump on false. A flow diagram of the **while** command and the corresponding code appears in Figure 7.5. We assume that the incoming value of the *InhLabel* attribute is n. The attribute grammar definition for the **while** command follows directly.

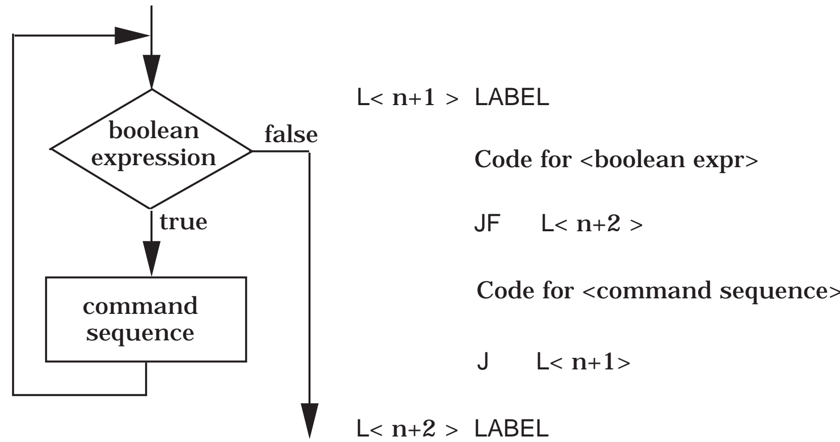

Figure 7.5: Flow Diagram and Code Generation for a **while** Command

<command> ::=

  while <boolean expr> **do** <command sequence> **end while**

    Code(<command>) ← concat(
      [(label(*InhLabel*(<command>)+1),LABEL)],
      Code(<boolean expr>),
      [(JF,label(*InhLabel*(<command>)+2))],
      Code(<command sequence>),
      [(J,label(*InhLabel*(<command>)+1))],
      [(label(*InhLabel*(<command>)+2),LABEL)])

    Temp(<boolean expr>) ← *Temp*(<command>)

    Temp(<command sequence>) ← *Temp*(<command>)

    InhLabel(<command sequence>) ← *InhLabel*(<command>)+2

    SynLabel(<command>) ← *SynLabel*(<command sequence>)

Since the **while** command itself needs two labels, *InhLabel* is incremented by two before being passed down to *InhLabel* for the command sequence, which may or may not generate new labels of its own. The *SynLabel* coming out of the command sequence is passed out of the **while** command. The inherited attribute *Temp* is passed down to both the Boolean expression and the command sequence. The utility function "label" converts an integer value into an appropriate label name.

  label(integer) = concat('L', string(integer))

The **if** command has two forms; we concentrate on the one with two alternatives

if <boolean expr> **then** <command sequence>
else <command sequence> **end if**

where the code generated by the Boolean expression is followed by a conditional jump on false. A flow diagram of the **if** command and the corresponding code appears in Figure 7.6. Again we assume the incoming value of the *InhLabel* attribute is n. The attribute grammar definition for this **if** command follows directly.

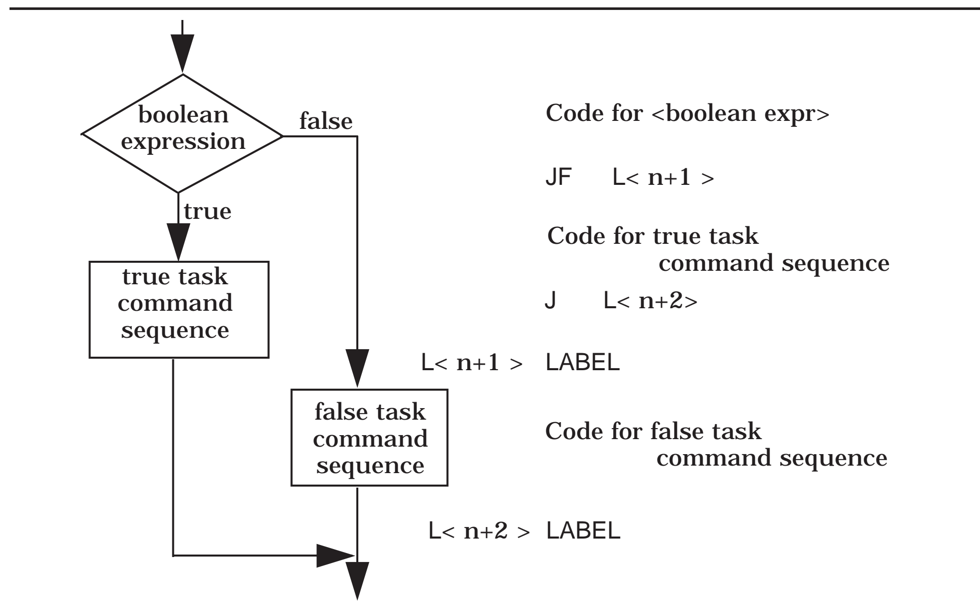

Figure 7.6: Flow Diagram and Code Generation for an **if** Command

<command> ::= **if** <boolean expr> **then** <command sequence>$_1$
else <command sequence>$_2$ **end if**

Code(<command>) ← concat(*Code*(<boolean expr>),
[(JF,label(*InhLabel*(<command>)+1))],
Code(<command sequence>$_1$),
[(J,label(*InhLabel*(<command>)+2))],
[(label(*InhLabel*(<command>)+1),LABEL)],
Code(<command sequence>$_2$),
[(label(*InhLabel*(<command>)+2),LABEL)])

Temp(<boolean expr>) ← *Temp*(<command>)
Temp(<command sequence>$_1$) ← *Temp*(<command>)
Temp(<command sequence>$_2$) ← *Temp*(<command>)

InhLabel(<command sequence>$_1$) ← *InhLabel*(<command>)+2
InhLabel(<command sequence>$_2$) ← *SynLabel*(<command sequence>$_1$)
SynLabel(<command>) ← *SynLabel*(<command sequence>$_2$)

Since the **if** command with two alternatives needs two labels, *InhLabel* is incremented by two before being passed down to the first command sequence. The *SynLabel* coming out of the first command sequence is threaded over as the input to *InhLabel* for the second command sequence. The *SynLabel* from the second command sequence is passed out of the **if** command. The inherited attribute *Temp* is passed down to the Boolean expression and to both command sequences.

The single alternative **if** command is simpler since it needs to generate only one label instruction. This attribute grammar clause will be presented in Figure 7.8. The attribute grammar for code generation for Wren is now complete. A summary of the synthesized and inherited attributes associated with each nonterminal is presented in Figure 7.7. The entire attribute grammar is given without interruption in Figure 7.8.

Nonterminal	Inherited Attributes	Synthesized Attributes
<program>	—	*Code*
<block>	—	*Code*
<declaration sequence>	—	—
<declaration>	—	—
<variable list>	—	—
<type>	—	—
<command sequence>	*Temp, InhLabel*	*Code, SynLabel*
<command>	*Temp, InhLabel*	*Code, SynLabel*
<expr>	*Temp*	*Code*
<integer expr>	*Temp*	*Code*
<term>	*Temp*	*Code*
<element>	*Temp*	*Code*
<weak op>	—	*OpCode*
<strong op>	—	*OpCode*
<boolean expr>	*Temp*	*Code*
<boolean term>	*Temp*	*Code*
<boolean element>	*Temp*	*Code*
<comparison>	*Temp*	*Code*
<relation>	—	*TestCode*
<variable>	—	*Name*
<identifier>	—	*Name*
<numeral>	—	*Name*
<letter>	—	*Name*
<digit>	—	*Name*

Figure 7.7: Attributes Associated with Nonterminal Symbols

<program> ::= **program** <identifier> **is** <block>
Code(<program>) ← concat(*Code*(<block>), [HALT])

<block> ::= <declaration sequence> **begin** <command sequence> **end**
Code(<block>) ← *Code*(<command sequence>)
Temp(<command sequence>) ← 0
InhLabel(<command sequence>) ← 0

<declaration sequence> ::= ε | <declaration> <declaration sequence>$_2$

<declaration> ::= **var** <variable list> **:** <type>**;**

<variable list> ::= <variable> | <variable> **,** <variable list>$_2$

<type> ::= **integer** | **boolean**

<command sequence> ::= <command>
Code(<command sequence>) ← *Code*(<command>)
Temp(<command>) ← *Temp*(<command sequence>)
InhLabel(<command>) ← *InhLabel*(<command sequence>)
SynLabel(<command sequence>) ← *SynLabel*(<command>)
| <command> ; <command sequence>$_2$
Code(<command sequence>) ←
concat(*Code*(<command>),*Code*(<command sequence>$_2$))
Temp(<command>) ← *Temp*(<command sequence>
Temp(<command sequence>$_2$) ← *Temp*(<command sequence>)
InhLabel(<command>) ← *InhLabel*(<command sequence>)
InhLabel(<command sequence>$_2$) ← *SynLabel*(<command>)
SynLabel(<command sequence>) ← *SynLabel*(<command sequence>$_2$)

<command> ::= <variable> **:=** <expr>
Code(<command>) ← concat(*Code*(<expr>),[(STO, *Name*(<variable>))])
Temp(<expr>) ← *Temp*(<command>)
SynLabel(<command>) ← *InhLabel*(<command>)

<command> ::= **read** <variable>
Code(<command>) ← [(GET, *Name*(<variable>))]
SynLabel(<command>) ← *InhLabel*(<command>)

<command> ::= **write** <integer expr>
Code(<command>) ←
concat(*Code*(<integer expr>),
[(STO, temporary(*Temp*(<command>)+1))],
[(PUT, temporary(*Temp*(<command>)+1))])
Temp(<integer expr>) ← *Temp*(<command>)
SynLabel(<command>) ← *InhLabel*(<command>)

Figure 7.8: Complete Attribute Grammar for Wren (Part 1)

<command> ::= **skip**
Code(<command>) ← [NO-OP]
SynLabel(<command>) ← *InhLabel*(<command>)

<command> ::= **while** <boolean expr> **do** <command sequence> **end while**
Code(<command>) ← concat([(label(*InhLabel*(<command>)+1),LABEL)],
Code(<boolean expr>),
[(JF,label(*InhLabel*(<command>)+2))],
Code(<command sequence>),
[(J,label(*InhLabel*(<command>)+1))],
[(label(*InhLabel*(<command>)+2),LABEL)]),
Temp(<boolean expr>) ← *Temp*(<command>)
Temp(<command sequence>) ← *Temp*(<command>)
InhLabel(<command sequence>) ← *InhLabel*(<command>)+2
SynLabel(<command>) ← *SynLabel*(<command sequence>)

<command> ::= **if** <boolean expr> **then** <command sequence> **end if**
Code(<command>) ← concat(*Code*(<boolean expr>),
[(JF,label(*InhLabel*(<command>)+1))],
Code(<command sequence>),
[(label(*InhLabel*(<command>)+1),LABEL)])
Temp(<boolean expr>) ← *Temp*(<command>)
Temp(<command sequence>) ← *Temp*(<command>)
InhLabel(<command sequence>) ← *InhLabel*(<command>)+1
SynLabel(<command>) ← *SynLabel*(<command sequence>)

<command> ::= **if** <boolean expr> **then** <command sequence>$_1$
else <command sequence>$_2$ **end if**
Code(<command>) ← concat(*Code*(<boolean expr>),
[(JF,label(*InhLabel*(<command>)+1))],
Code(<command sequence>$_1$),
[(J,label(*InhLabel*(<command>)+2))],
[(label(*InhLabel*(<command>)+1),LABEL)],
Code(<command sequence>$_2$),
[(label(*InhLabel*(<command>)+2),LABEL)])
Temp(<boolean expr>) ← *Temp*(<command>)
Temp(<command sequence>$_1$) ← *Temp*(<command>)
Temp(<command sequence>$_2$) ← *Temp*(<command>)
InhLabel(<command sequence>$_1$) ← *InhLabel*(<command>)+2
InhLabel(<command sequence>$_2$) ← *SynLabel*(<command sequence>$_1$)
SynLabel(<command>) ← *SynLabel*(<command sequence>$_2$)

Figure 7.8: Complete Attribute Grammar for Wren (Part 2)

```
<expr> ::=
        <integer expr>
          Code(<expr>) ← Code(<integer expr>)
          Temp(<integer expr>) ← Temp(<expr>)
      | <boolean expr>
          Code(<expr>) ← Code(<boolean expr>)
          Temp(<boolean expr>) ← Temp(<expr>)

<integer expr> ::=
        <term>
          Code(<integer expr>) ← Code(<term>)
          Temp(<term>) ← Temp(<integer expr>)
      | <integer expr>2 <weak op> <term>
          Code(<integer expr>) ← concat(Code(<integer expr>2),
                    optimize(Code(<term>),Temp(<integer expr>),OpCode(<weak op>)))
          Temp(<integer expr>2) ← Temp(<integer expr>)
          Temp(<term>) ← Temp(<integer expr>)+1

<weak op> ::=
        +
          OpCode(<weak op>) ← ADD
      | –
          OpCode(<weak op>) ← SUB

<term> ::=
        <element>
          Code(<term>) ← Code(<element>)
          Temp(<element>) ← Temp(<term>)
      | <term>2 <strong op> <element>
          Code(<term>) ← concat(Code(<term>2),
                    optimize(Code(<element>),Temp(<term>),OpCode(<strong op>)))
          Temp(<term>2) ← Temp(<term>)
          Temp(<element>) ← Temp(<term>)+1

<strong op> ::=
        *
          OpCode(<strong op>) ← MULT
      | /
          OpCode(<strong op>) ← DIV

<element> ::=
        <numeral>
          Code(<element>) ← [(LOAD, Name(<numeral>))]
      | <variable>
          Code(<element>) ← [(LOAD, Name(<variable>))]
      | ( <integer expr> )
          Code(<element>) ← Code(<integer expr>)
          Temp(<integer expr>) ← Temp(<element>)
```

Figure 7.8: Complete Attribute Grammar for Wren (Part 3)

```
<boolean expr> ::=
        <boolean term>
            Code(<boolean expr>) ← Code(<boolean term>)
            Temp(<boolean term>) ← Temp(<boolean expr>)
      | <boolean expr>_2  or  <boolean term>
            Code(<boolean expr>) ← concat(Code(<boolean expr>_2),
                                          [(STO, temporary(Temp(<boolean expr>)+1))],
                                          Code(<boolean term>,
                                          [(OR, temporary(Temp(<boolean expr>)+1))])
            Temp(<boolean expr>_2) ← Temp(<boolean expr>)
            Temp(<boolean term>) ← Temp(<boolean expr>)+1
<boolean term> ::=
        <boolean element>
            Code(<boolean term>) ← Code(<boolean element>)
            Temp(<boolean element>) ← Temp(<boolean term>)
      | <boolean term>_2  and  <boolean element>
            Code(<boolean term>) ← concat(Code(<boolean term>_2),
                                          [(STO, temporary(Temp(<boolean term>)+1))],
                                          Code(<boolean element>,
                                          [(AND, temporary(Temp(<boolean term>)+1))])
            Temp(<boolean term>_2) ← Temp(<boolean term>)
            Temp(<boolean element>) ← Temp(<boolean term>)+1

<boolean element> ::=
        false
            Code(<boolean element>) ← [(LOAD, 0)]
      | true
            Code(<boolean element>) ← [(LOAD, 1)]
      | <variable>
            Code(<boolean element>) ← [(LOAD, Name(<variable>))]
      | <comparison>
            Code(<boolean element>) ← Code(<comparison>)
            Temp(<comparison>) ← Temp(<boolean element>)
      | ( <boolean expr> )
            Code(<boolean element>) ← Code(<boolean expr>)
            Temp(<boolean expr>) ← Temp(<boolean element>)
      | not ( <boolean expr> )
            Code(<boolean element>) ← concat(Code(<boolean expr>), [NOT]),
            Temp(<boolean expr>) ← Temp(<boolean element>)

<comparison> ::= <integer expr>_1 <relation> <integer expr>_2
            Code(<comparison>) ← concat( Code(<integer expr>_1),
                    optimize(Code(<integer expr>_2),Temp(<comparison>),SUB),
                    [TestCode(<relation>)])
            Temp(<integer expr>_1) ← Temp(<comparison>)
            Temp(<integer expr>_2) ← Temp(<comparison>)+1
```

Figure 7.8: Complete Attribute Grammar for Wren (Part 4)

<relation> ::=
>
TestCode(<relation>) ← TSTGT
| >=
TestCode(<relation>) ← TSTGE
| <>
TestCode(<relation>) ← TSTNE
| =
TestCode(<relation>) ← TSTEQ
| <=
TestCode(<relation>) ← TSTLE
| <
TestCode(<relation>) ← TSTLT

<variable> ::= <identifier>
N*ame*(<variable>) ← *Name*(<identifier>)

<identifier> ::=
<letter>
Name(<identifier>) ← *Name*(<letter>)
| <identifier>$_2$ <letter>
Name(<identifier>) ← concat(*Name*(<identifier>$_2$),*Name*(<letter>))
| <identifier>$_2$ <digit>
Name(<identifier>) ← concat(*Name*(<identifier>$_2$),*Name*(<digit>))

<letter> ::=
a
Name(<letter>) ← 'A'
: :
| **z**
Name(<letter>) ← 'Z'

<numeral> ::=
<digit>
Name(<numeral>) ← *Name*(<digit>)
| <numeral>$_2$ <digit>
Name(<numeral>) ← concat(*Name*(<numeral>$_2$),*Name*(<digit>))

<digit> ::=
0
Name(<digit>) ← '0'
: :
| **9**
Name(<digit>) ← '9'

Figure 7.8: Complete Attribute Grammar for Wren (Part 5)

Auxiliary Functions

```
optimize(code, temp, opcode) =
    if length(code) = 1 then                    -- a variable or numeral
                                  [(opcode, secondField(first(code)))]
                              else
                                  concat([(STO, temporary(temp+1))],
                                          code,
                                          [(STO, temporary(temp+2))],
                                          [(LOAD, temporary(temp+1))],
                                          [(opcode, temporary(temp+2))])

temporary(integer) =  concat('T',string(integer))

label(integer) = concat('L', string(integer))

string(n) =   if n = 0 then '0'
                :  :          :
              else if n = 9 then '9'
              else concat(string(n/10), string(n mod 10))
```

Figure 7.8: Complete Attribute Grammar for Wren (Part 6)

Exercises

1. The negation of an element was not specified in the attribute grammar of Figure 7.8. Add this alternative to the production for <element> without adding any new instructions to the object code.

2. Draw the complete, decorated tree for the arithmetic expression

   2 * x * y + z / 3

3. Draw the complete, decorated tree for the command sequence in the following program:

```
program mod is
  var m, n : integer;
begin
  read m;  read n;
  while m > n do
        m := m – n
  end while;
  write m
end
```

4. Without drawing the complete tree, show the code generated by attribute grammar for the following Wren program:

```
program multiply is
  var m, n, product : integer;
begin
  read m;  read n;
  product := 0;
  while n > 0 do
        if 2 * (n / 2) < > n then  (* if n is odd *)
                product := product + m
        end if;
        m := 2 * m;  n := n / 2
  end while;
  write product
end
```

Compare the answer for this problem with the answer for exercise 1 in section 7.1. Is there any difference in efficiency between hand-generated and machine-generated code?

5. Change the semantic rules for the **write** command so that the code is optimized when the expression being printed is a variable.

6. The Boolean expressions in the grammar given are fully evaluated. Some programming languages short-circuit Boolean expression evaluation once the final result is known. Is it possible to modify the grammar in a *simple way* so that short-circuit evaluation is performed?

7. We did not include an optimization for Boolean expressions. Will such an optimization be possible? If it is, add it to the attribute grammar; if it is not, explain why.

8. Add the command

   repeat <command sequence> **until** <boolean expr>

   to Wren and modify the attribute grammar so that the generated code causes the loop to be exited when the Boolean expression is true.

9. Add the conditional integer expression

   if <boolean expr> **then** <integer expr>$_1$ **else** <integer expr>$_2$

   to Wren and modify the attribute grammar accordingly.

10. Add integer expressions with side effects

    begin <command sequence> **return** <integer expr> **end**

    to Wren and modify the attribute grammar so that the value returned is evaluated with the state produced by executing the command sequence.

11. Reverse Polish notation is used on some calculators to evaluate arithmetic expressions. For a binary operation, the first operand is pushed on a stack, the second operand is pushed on the stack, and, when the operation is performed, both operands are popped off the stack and the result of the operation is pushed back onto the stack. Introduce appropriate machine instructions for a stack architecture for arithmetic expression evaluation and modify the attribute grammar for Wren accordingly.

12. The following BNF grammar defines an expression language with binary operations +, –, *, and /, unary -, and the variables a, b, and c.

    <expr> ::= <term> | <expr> + <term> | <expr> – <term>

    <term> ::= <elem> | <term> * <elem> | <term> / <elem>

    <elem> ::= a | b | c | (<expr>) | - <expr>

    Convert this definition into an attribute grammar whose main attribute *Val* is an abstract syntax tree for the expression represented in the form of a tagged structure similar to a Prolog structure. For example,

    Val("(a–b)*-(b+c)/a") = times(minus(a,b),divides(negative(plus(b,c)),a)).

7.3 LABORATORY: IMPLEMENTING CODE GENERATION

As in Chapter 3, we will be developing an attribute grammar written in Prolog, but unlike that previous laboratory project, our goal now is the generation of intermediate code, as described in section 7.1. Although we parse the declaration section of the program, we do not use this information in the generation of code.

As before, we assume "front end" code to read the text from a file and convert it into a sequence of tokens. We also add a pretty-printing capability on the "back end" so that the resulting program looks like assembly code. An example illustrates the code generator.

```
>>> Translating Wren <<<
Enter name of source file: gcd.wren
    program gcd is
    var m,n: integer;
    begin
      read m; read n;
      while m <> n do
        if m < n then n := n - m
                 else m := m - n
        end if
```

```
      end while;
      write m
    end
Scan successful
[program,ide(gcd),is,var,ide(m),comma,ide(n),colon,integer,
 semicolon,begin,read,ide(m),semicolon,read,ide(n),semicolon,
 while,ide(m),neq,ide(n),do,if,ide(m),less,ide(n),then,ide(n),
 assign,ide(n),minus,ide(m),else,ide(m),assign,ide(m),minus,
 ide(n),end,if,end,while,semicolon,write,ide(m),end,eop]
Parse successful
[[GET,m],[GET,n],[L1,LABEL],[LOAD,m],[SUB,n],TSTNE,[JF,L2],[LOAD,m],[SUB,n],
 TSTGT,[JF,L3],[LOAD,n],[SUB,m],[STO,n],[J,L4],[L3,LABEL],
 [LOAD,m],[SUB,N],[STO,m],[L4,LABEL],[J,L1],[L2,LABEL],
 [LOAD,m],[STO,T1],[PUT,T1],HALT]
      GET   M
      GET   N
L1    LABEL
      LOAD  M
      SUB   N
      TSTNE
      JF    L2
      LOAD  M
      SUB   N
      TSTGT
      JF    L3
      LOAD  N
      SUB   M
      STO   N
      J     L4
L3    LABEL
      LOAD  M
      SUB   N
      STO   M
L4    LABEL
      J     L1
L2    LABEL
      LOAD  M
      STO   T1
      PUT   T1
      HALT
yes
```

The transcript above shows the token list produced by the scanner and the list of assembly language instructions constructed by the attribute grammar

woven throughout the parser. A pretty-print routine capitalizes symbols in the code and formats the output. The program above is the gcd program in Wren that was discussed in section 7.1.

This example illustrates the code generated by the Prolog translator once it is fully implemented. As in previous laboratory sections, we provide only a partial implementation and leave the unimplemented components as exercises.

The generated code for the synthesized attribute *Code* is maintained as a Prolog list of assembly language instructions, each of which is a Prolog list itself.

The program clause adds the instruction 'HALT' to the generated code; at the same time it ignores the program identifier since that value does not affect the code generated. Because uppercase has particular significance in Prolog, generated opcodes must be enclosed in apostrophes. At the block level, the synthesized *Code* attribute is passed to the program level and the inherited attributes for *Temp* and *InhLabel* are initialized to zero.

```
program(Code) -->   [program, ide(Ident), is], block(Code1),
                    { concat(Code1, ['HALT'], Code) }.

block(Code) --> decs, [begin], commandSeq(Code,0,0,SynLabel), [end].
```

Commands

Implementing decs following the example in Chapter 2 is left as an exercise. We break a command sequence into the first command followed by the rest of the commands, if any. The *Temp* attribute is passed to both children, the *InhLabel* attribute from the command sequence is inherited by the first command, the *SynLabel* attribute from the first command becomes the *InhLabel* attribute of the rest of the commands, and the *SynLabel* of the rest of the commands is passed to the parent command sequence. The two code sequences are concatenated in a list structure. The rest of the commands are handled in a similar manner except that when no more commands remain, the resulting code list is empty.

```
commandSeq(Code,Temp,InhLab,SynLab) -->
                              command(Code1,Temp,InhLab,SynLab1),
                              restcmds(Code2,Temp,SynLab1,SynLab),
                              { concat(Code1, Code2, Code) }.

restcmds(Code,Temp,InhLab,SynLab) -->
                              [semicolon],
                              command(Code1,Temp,InhLab,SynLab1),
                              restcmds(Code2,Temp,SynLab1,SynLab),
                              { concat(Code1, Code2, Code) }.

restcmds([ ],Temp,InhLab,InhLab) --> [ ].
```

The input and **skip** commands do not use the *Temp* attribute and simply turn the label attribute around and feed it back out by placing the same variable Label in both argument places. The assignment and output commands use the *Temp* attribute and turn around the label attribute. Some of these commands appear below, others are left as exercises.

```
command([['GET', Var]], Temp, Label, Label) --> [read,ide(Var)].

command(Code, Temp, Label, Label) -->
                                        [ide(Var), assign], expr(Code1,Temp),
                                        { concat(Code1, [['STO',Var]], Code) }.
```

The input of a variable is translated into the GET of the same variable. The output of an expression is the code generated for the expression, followed by the store of its result in a temporary location and a PUT of this location. The **skip** command generates a NO-OP. The assignment command concatenates a STO of the target variable after the code generated by expression. The reader is encouraged to complete the **write** and **skip** commands.

The single alternative **if** command consists of the code for the Boolean expression that includes a test operation, a conditional jump, the code for the body, and a label instruction that is the target of the conditional jump. Notice the use of the built-in Prolog predicate is to evaluate an arithmetic expression and bind the result. We have also used a utility predicate label to combine L with the label number. Note that we need to define a concat predicate that concatenates three lists (see Appendix A).

```
command(Code,Temp,InhLab,SynLab) -->
                    [if], { InhLab1 is InhLab+1, label(InhLab1,Lab) },
                    booleanExpr(Code1,Temp),
                    [then], commandSeq(Code2,Temp,InhLab1,SynLab), [end,if],
                    { concat(Code1, [['JF',Lab]|Code2], [[Lab,'LABEL']], Code) }.

label(Number,Label) :-
          name('L',L1), name(Number,L2), concat(L1,L2,L), name(Label,L).
```

The two-alternative **if** command has the most complex code sequence:

- The code from the Boolean expression
- A conditional jump to the false task
- The code from the true task
- An unconditional jump to the label instruction following the entire command
- A label instruction for entry into the false task
- The code for the false task itself
- The final label instruction for the jump out of the true task.

The same T*emp* attribute is passed to all three children. Since the two-alternative **if** command requires two unique labels, the *InhLabel* for the true command sequence has been incremented by two. The *SynLabel* out of the true command sequence is threaded into the false command sequence. The *SynLabel* of the false command sequence is passed to the parent. Here we need a concat predicate that concatenates four lists.

```
command(Code,Temp,InhLab,SynLab) -->
                [if], { InhLab1 is InhLab+1, InhLab2 is InhLab+2,
                        label(InhLab1,Lab1), label(InhLab2,Lab2) },
                booleanExpr(Code1,Temp),
                [then], commandSeq(Code2,Temp,InhLab2,SynLab2),
                [else], commandSeq(Code3,Temp,SynLab2,SynLab), [end,if],
                { concat(Code1, [['JF',Lab1]|Code2],
                   [['J',Lab2], [Lab1,'LABEL']|Code3], [[Lab2,'LABEL']], Code) }.
```

The **while** command begins with a label instruction that is the target for the unconditional jump at the bottom of the loop, which is followed by the code for the Boolean expression, a conditional jump out of the **while**, the code for the loop body, an unconditional jump to the top of the loop, and a final label instruction for exiting the **while** loop. The *Temp* attribute is inherited down to the Boolean expression and loop body. Since two labels are used, the InhLabel to the loop body is incremented by two and the *SynLabel* from the loop body is passed back up to the parent. Completion of the code for a **while** command is left as an exercise.

Expressions

The code generated by arithmetic expressions does not involve labels, so the label attributes are not used at all. As we saw earlier in Chapter 2, we have to transform our left recursive attribute grammar into a right recursive format when implemented as a logic grammar. If an expression goes directly to a single term, then *Temp* is passed in and Code is passed out. If an expression is a term followed by one or more subsequent terms, then the inherited *Temp* value is passed down to the left-hand term and this value incremented by one is passed to the right-hand term. There may be still more terms to the right, but since the additive operations are left associative, we have completed the operation on the left two terms and the temporary locations can be used again. Therefore the original *Temp* value is passed down to the clause for the remaining terms.

The generated code for an integer expression is the code from the first term followed by the optimized code from any remaining terms. If the code from the right-hand term is simply the load of a variable or a numeral, the code is

optimized by having the opcode associated with the binary operation applied directly to the simple operand. If this is not the case, the result from the left operand is stored in a temporary, the code is generated for the right operand that is stored in a second temporary, the first temporary is loaded, and the operation is applied to the second temporary. The predicate optimize allows two forms due to the two possible list structures. Notice the use of the utility predicate temporary to build temporary location names. The resulting code from an expression with multiple terms is the code from the first term, the code from the second term, and the code from the remaining terms, if any.

```
integerExpr(Code,Temp) -->   term(Code1,Temp), restExpr(Code2,Temp),
                             { concat(Code1, Code2, Code) }.

restExpr(Code,Temp) -->   weakop(Op), { Temp1 is Temp+1 },
                          term(Code1,Temp1),
                          { optimize(Code1,OptCode1,Temp,Op) },
                          restExpr(Code2,Temp),
                          { concat(OptCode1, Code2, Code) }.

restExpr([ ],Temp) --> [ ].

weakop('ADD') --> [plus].

weakop('SUB') --> [minus].

optimize([['LOAD',Operand]],[[Opcode,Operand]],Temp,Opcode).
optimize(Code,OptCode,Temp,Op) :-
        Temp1 is Temp+1, Temp2 is Temp+2,
        temporary(Temp1,T1), temporary(Temp2,T2),
        concat([['STO',T1]|Code], [['STO',T2], ['LOAD',T1], [Op,T2]], OptCode).

temporary(Number,Temp) :-
        name('T',T1), name(Number,T2), concat(T1,T2,T), name(Temp,T).
```

Terms are similar to expressions and are left as an exercise. For now, we give a clause for terms that enables the current specification of the attribute grammar to work correctly on a restrict subset of Wren with only the "weak" arithmetic operators. This clause will have to be replaced to produce correct translations of terms in Wren.

```
term(Code,Temp) --> element(Code,Temp).
```

An element can expand to a number, an identifier, or a parenthesized expression, in which case the ***Temp*** attribute is passed in. The negation of an element is left as an exercise.

```
element(['LOAD',Number], Temp) --> [num(Number)].

element(['LOAD',Name], Temp) --> [ide(Name)].

element(Code,Temp) --> [lparen], expression(Code,Temp), [rparen].
```

The code for expressions, Boolean expressions, Boolean terms, and Boolean elements is left as an exercise.

The final task is to generate the code for comparisons. We generate code for the left-side expression, recognize the relation, generate code for the right-side expression, and then call optimize for the right side using the subtract operation. The code for the comparison is the concatenation of the code for the left-side expresion, the optimized code for the right-side expression, and the test instruction. In the code below, the variable Tst holds the value of the test operation returned by testcode. The reader is encouraged to write the clauses for testcode.

```
comparison(Code,Temp) -->
        { Temp1 is Temp+1 },
        integerExpr(Code1,Temp),
        testcode(Tst), integerExpr(Code2,Temp1),
        { optimize(Code2,OptCode2,Temp,'SUB') },
        { concat(Code1,OptCode2,[Tst], Code) }.
```

This completes our partial code generator for Wren. The exercises below describe the steps needed to complete this Wren translator. Other exercises deal with extensions to Wren that can be translated into intermediate code.

Exercises

1. Complete the implementation given in this section by adding the following featues:
    - The output and **skip** commands
    - The **while** command
    - Clauses for term, remterm, and strongop
    - Clauses for expression, boolExpr, boolTerm, and boolElement
    - Clauses for testcode

2. Write a pretty-print routine for the intermediate code. Add a routine to capitalize all identifiers. All commands, except for labels, are indented by one tab and there is a tab between an opcode and its argument. A tab character is generated by put(9) and a return is accomplished by nl. Recursion in the pretty-print predicate can stop once the halt instruction is encountered and printed.

3. The negation of an element (unary minus) was not specified in the production that defines elements. Add this alternative using the *existing* intermediate code instructions.

4. Modify the output command to print a list of expression values. Add a new intermediate code command for a line feed that is generated after the list of expressions is printed.

5. Add the repeat .. until command, as described in exercise 8, section 7.2.

6. Add a conditional expression, as described in exercise 9, section 7.2.

7. Add expressions with side effects, as described in exercise 10, section 7.2.

8. Follow exercise 11 in section 7.2 to change the code generation for a stack architecture machine and to implement these changes in the Prolog code generator.

7.4 FURTHER READING

The use of attribute grammars for code generation is of primary interest to the compiler writer. Lewis, Rosenkrantz, and Stearns presented an early paper on attributed translations [Lewis74]. Several references in compiler construction have already been noted in Section 3.4 [Aho86], [Fischer91], [Parsons92], and [Pittman92].

Frank Pagan presents a code-generating attribute grammar for the language Pam [Pagan81]. Pam is somewhat simpler than Wren since all variables are of type integer. Because there is no need to generate Boolean values, as our TST instructions do, his target language has six conditional jumps that use the same instruction both to test and jump.

We have assigned programs for students to use the Synthesizer-Generator [Reps89] to implement code generation for Wren. Our code generator in Prolog operates in batch mode whereas the Synthesizer-Generator code-generating editor operates in incremental mode. Two windows appear on the screen, one for the source code and one for the object code. As the code is entered in the source window, the corresponding object code appears immediately. Changes in the source code, including deletions, result in “instantaneous” changes in the object code, even when this involves changes in label numbers and temporary location numbers.