APPENDIX A

# Answers to Selected Exercises

Specimen answers to about half of the exercises are given here. Some of the answers are given only in outline.

## Answers 1

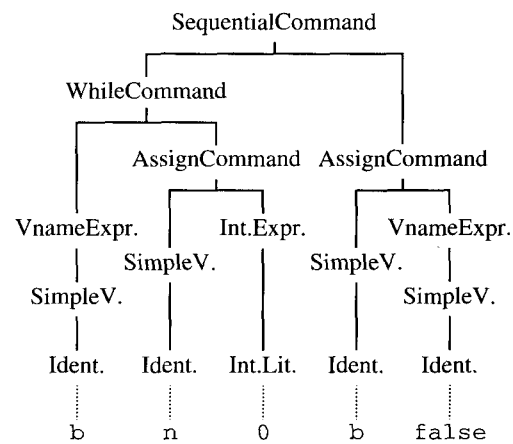
**1.1** Other kinds of language processor: syntax checkers, cross-referencers, pretty-printers, high-level translators, program transformers, symbolic debuggers, etc.

**1.4** Mini-Triangle expressions: (a) and (e) only. (Mini-Triangle has no functions, no unary operators, and no operator '>='.)

Commands: (f) and (j) only. (Mini-Triangle procedures have exactly one parameter each, and there is no if-command without an else-part.)

Declarations: (l), (m), and (o). (Mini-Triangle has no real-literals, and no multiple variable declarations.)

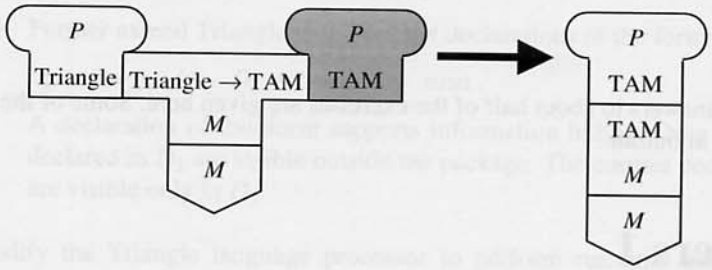
**1.5** AST:



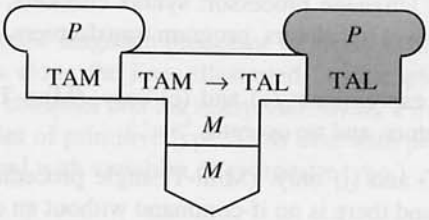
**1.6** The value written is 10.

# Answers 2

**2.2** (a) Compiling and (b) running a Triangle program:

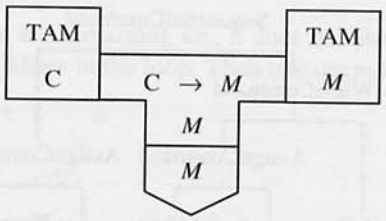



(c) Disassembling the object program:

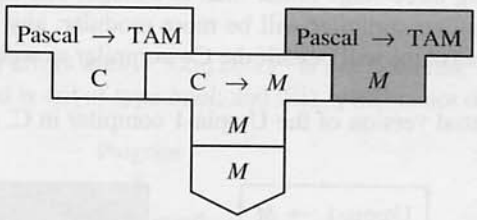



The purpose of the disassembler is to allow the programmer to read the compiler's object code.

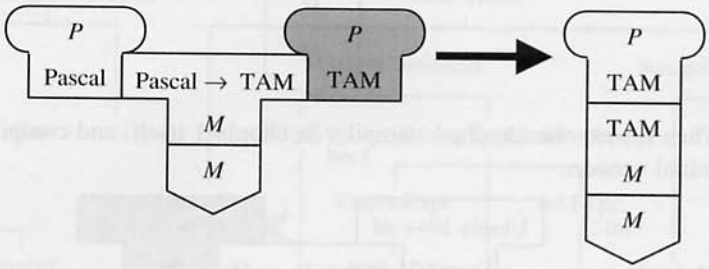
**2.4** (a) Compiling the TAM interpreter:



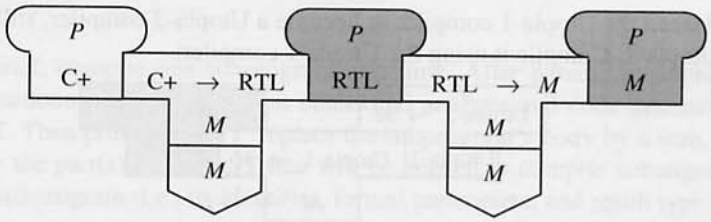
(b) Compiling the Pascal compiler:



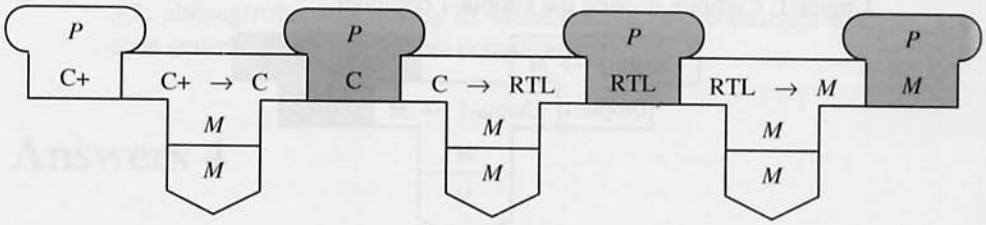

(c) Compiling and running a Pascal program:




**2.8** *Strategy 1:* Extend the C-into-RTL translator to become a C+-into-RTL translator, and compile it. Composed with the RTL-into-*M* translator, this gives a two-stage C+ compiler (similar to the given C compiler):



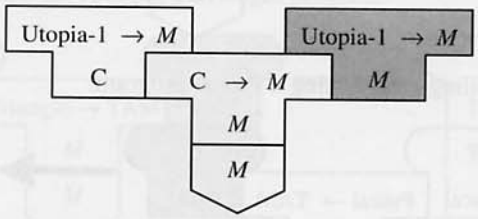

*Strategy 2:* Write a C+-into-C translator (a *preprocessor*) in C, and compile it. Composed with the two-stage C-into-*M* compiler, this gives a three-stage C+ compiler:



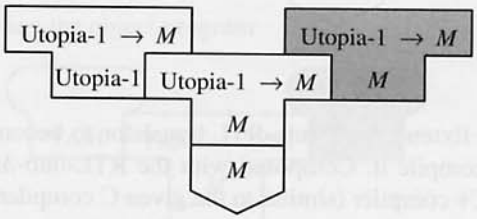

Strategy 2 requires more work, because the preprocessor must not only analyze the C+ source program, but also generate a C object program (requiring good

layout if intended to be readable). Moreover the resulting compiler will be slower, being three-stage rather than two-stage. The advantage of strategy 2 is that the resulting compiler will be more modular: any improvements in the C-into-RTL translator will benefit the C+ compiler as well as the C compiler.

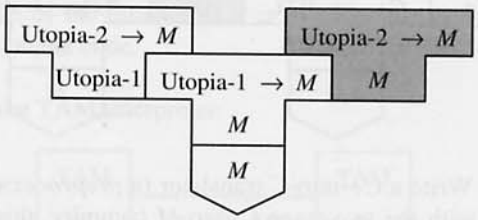
**2.9** Write an initial version of the Utopia-1 compiler in C, and compile it using the C compiler:



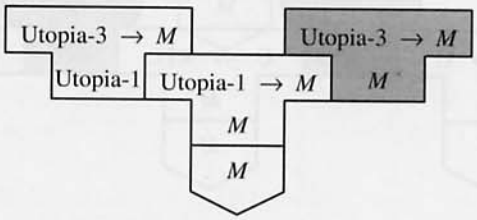

Then rewrite the Utopia-1 compiler in Utopia-1 itself, and compile it using the initial version:




Extend the Utopia-1 compiler to become a Utopia-2 compiler, still expressed in Utopia-1. Compile it using the Utopia-1 compiler:



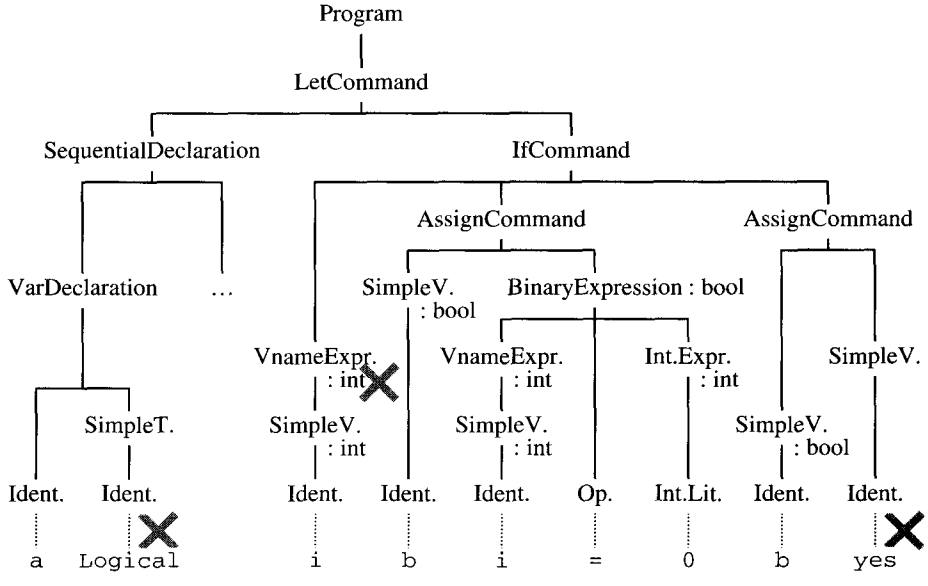

Extend the Utopia-2 compiler to become a Utopia-3 compiler, still expressed in Utopia-1. Compile it using the Utopia-1 compiler:




(Alternatively, the Utopia-3 compiler could be written in Utopia-2.)

# Answers 3

**3.3** The contextual errors are (i) '`Logical`' is not declared; (ii) the expression of the if-command is not of type *bool*; and (iii) '`yes`' is not declared:




**3.5** In brief, compile one subprogram at a time. After parsing a subprogram and constructing its AST, perform contextual analysis and code generation on the AST. Then prune the AST: replace the subprogram's body by a stub, and retain only the part(s) of the AST that will be needed to compile subsequent calls to the subprogram (i.e., its identifier, formal parameters, and result type if any).

The maximum space requirement will be for the largest subprogram's AST, plus the pruned ASTs of all the subprograms.

**3.6** This restructuring would be feasible. It would be roughly similar to Answer 3.5, although the interleaving of syntactic analysis, contextual analysis, and code generation would be more complicated.

# Answers 4

**4.3** After repeated left factorization and elimination of left recursion:

Numeral ::= Digits ( **.** Digits | ε) (**e** Sign Digits | ε)

Digits ::= Digit Digit*

**4.4** (a) {**C**, **J**, **P**}

(b) {**0**, **1**, **2**, **a**, **b**}

(c) *starters*[[Digit]] = *starters*[[Digits]] = *starters*[[Numeral]] = {**0**, **1**, **2**, **3**}

(d) *starters*[[Subject]] = {**I**, **a**, **the**}; *starters*[[Object]] = {**me**, **a**, **the**}.

**4.9** Parsing methods (with enhancements italicized):

```
private void parseCommand () {
   int expval = parseExpression();
   accept('=');
   print(expval);
}

private int parseExpression () {
   int expval = parseNumeral();
   while (currentChar == '+'
          || currentChar == '-'
          || currentChar == '*') {
      char op = currentChar;
      acceptIt();
      int numval = parseNumeral();
      switch (op) {
      case '+':  expval += numval;  break;
      case '-':  expval -= numval;  break;
      case '*':  expval *= numval;  break;
      }
   }
   return expval;
}

private int parseNumeral () {
   int numval = parseDigit();
   while (isDigit(currentChar))
      numval = 10*numval + parseDigit();
   return numval;
}

private byte parseDigit () {
   if ('0' <= currentChar && currentChar <= '9')
      byte digval = currentChar - '0';
      currentChar = next input character;
      return digval;
   } else
      report a lexical error
```

**4.11** (a) Refine 'parse $X \mid Y$' to:

```
if (currentToken.kind is in starters[[X]])
   parse X
else if (currentToken.kind is in starters[[Y]])
   parse Y
else
   report a syntactic error
```

This is correct if and only if *starters*[[$X$]] and *starters*[[$Y$]] are disjoint.

(b) Refine 'parse [$X$]' to:

```
if (currentToken.kind is in starters[[X]])
   parse X
```

This is correct if and only if *starters*[[$X$]] is disjoint from the set of tok that can follow [$X$] in this particular context.

(c) Refine 'parse $X^+$' to:

```
do
   parse X
while (currentToken.kind is in starters[[X]]);
```

This is correct if and only if *starters*[[$X$]] is disjoint from the set of tok that can follow $X^+$ in this particular context.

**4.12** After left factorization:

```
single-Command ::=  ...
                  |  if Expression then single-Command
                        (else single-Command | ε)
                  |  ...
```

The tokens that can follow a single-command are {`else`, `end`}. This set is disjoint from *starters*[[`else` single-Command]] = {`else`}, so the gramma not LL(1). (In fact, no ambiguous grammar is LL(1).)

The parsing method obtained by converting this production rule would be:

```
private void parseSingleCommand () {
  switch (currentToken.kind) {
  ...

  case Token.IF: {
         acceptIt();
         parseExpression();
         accept(Token.THEN);
         parseSingleCommand();
```

```
            if (currentToken.kind == Token.ELSE) {
               acceptIt();
               parseSingleCommand();
            }
         }
         break;

      ...
      }
   }
```

Given (4.34) as input, `parseSingleCommand` would accept '`if` $E_1$ `then`', and then call itself recursively. The recursive activation of `parseSingleCommand` would accept '`if` $E_2$ `then` $C_1$ `else` $C_2$', and then return. The original activation would then also return.

This behavior is exactly what Pascal and C specify.

**4.14** After eliminating left recursion:

| | | |
|---|---|---|
| Expression | ::= | secondary-Expression (add-Operator secondary-Expression)* |
| secondary-Expression | ::= | primary-Expression (mult-Operator primary-Expression)* |

Parsing procedures (with AST enhancements in italics):

```
private Expression parseExpression () {
   Expression e1AST =
         parseSecondaryExpression();
   while (currentToken.kind ==
         Token.ADDOPERATOR) {
      Operator opAST = parseAddOperator();
      Expression e2AST =
         parseSecondaryExpression ();
      e1AST = new BinaryExpression(
         e1AST, opAST, e2AST);
   }
   return e1AST;
}

private Expression parseSecondaryExpression () {
   Expression e1AST = parsePrimaryExpression();
   while (currentToken.kind ==
         Token.MULTOPERATOR) {
      Operator opAST = parseMultOperator();
      Expression e2AST =
         parsePrimaryExpression();
```

```
            e1AST = new BinaryExpression(
                    e1AST, opAST, e2AST);
        }
        return e1AST;
    }
```

Procedure parsePrimaryExpression would be similar to the corresponding method in Example 4.12.

**4.16** (a) Define an abstract class AST together with concrete subclasses NonterminalAST and TerminalAST:

```
public abstract class AST {
  public byte tag;

  public static final byte  // tag values
     IDENTIFIER          =  0,
     INTLITERAL          =  1,
     OPERATOR            =  2,
     PROGRAM             =  3,
     ASSIGNCOMMAND       =  4,
     CALLCOMMAND         =  5,
     ...
     CONSTDECLARATION    = 15,
     VARDECLARATION      = 16,
     SEQDECLARATION      = 17,
     SIMPLETYPEDENOTER   = 18;
}

////////////////////////////////////////////////

public class NonterminalAST extends AST {
  public AST[] children;

  public NonterminalAST (byte tag,
          AST[] children) {
     this.tag = tag;  this.children = children;
  }
}

////////////////////////////////////////////////

public class TerminalAST extends AST {
  public String spelling;

  public TerminalAST (byte tag,
          String spelling) {
     this.tag = tag;  this.spelling = spelling;
  }
}
```

(b) To display an AST:

```
public abstract class AST {
  ...
  public void display (byte level);
}

////////////////////////////////////////////

public class NonterminalAST extends AST {
  ...
  public void display (byte level) {
     for (int i = 0; i < level; i++)
        print("  ");
     switch (this.tag) {
     case AST.PROGRAM:
        println("Program");  break;
     ...
     }
     for (int i = 0;
              i < this.children.length; i++)
        this.children[i].display(level+1);
  }
}

////////////////////////////////////////////

public class TerminalAST extends AST {
  ...
  public void display (byte level) {
     for (int i = 0; i < level; i++)
        print("  ");
     switch (this.tag) {
     case AST.IDENTIFIER:
        print("Identifier ");  break;
     ...
     }
     println(this.spelling);
  }
}
```

**4.18** This lexical grammar is ambiguous. The scanning procedure would turn ( follows:

```
private byte scanToken () {
  switch (currentChar) {
```

```
        case 'a': case 'b': case 'c': case 'd':
        ...
        case 'y': case 'z':
            takeIt();
            while (isLetter(currentChar)
                    || isDigit(currentChar))
                takeIt();
            return Token.IDENTIFIER;

        ...
        case 'i':
            takeIt(); take('f');
            return Token.IF;
        case 't':
            takeIt(); take('h'); take('e'); take('n');
            return Token.THEN;
        case 'e':
            takeIt(); take('l'); take('s'); take('e');
            return Token.ELSE;
        ...
        }
    }
```

This method will not compile. Moreover, there is no reasonable way to fix it.

# Answers 5

**5.2** One possibility would be a pair of subtables, one for globals and one for locals. (Each subtable could be an ordered binary tree or a hash table.) There would also be a variable, the *current level*, set to either *global* or *local*. Constructor `IdentificationTable` would set the current level to *global*, and would empty both subtables. Method `enter` would add the new entry to the global or local subtable, according to the current level. Method `retrieve` would search the local subtable first, and if unsuccessful would search the global subtable second. Method `openScope` would change the current level to *local*. Method `closeScope` would change it to *global*, and would also empty the local subtable.

**5.3** Constructor `IdentificationTable` would make the stack contain a single empty binary tree. Method `enter` would add the new entry to the topmost binary tree. Method `retrieve` would search the binary trees in turn, starting with the topmost, and stopping as soon as it finds a match. Method `open-`

`Scope` would push an empty binary tree on to the stack. Method `closeScope` would pop the topmost binary tree.

This implementation would be more efficient than the simple stack implementation, because it would replace linear search by binary search. In terms of time complexity, retrieval would be $O(\log n)$ rather than $O(n)$.

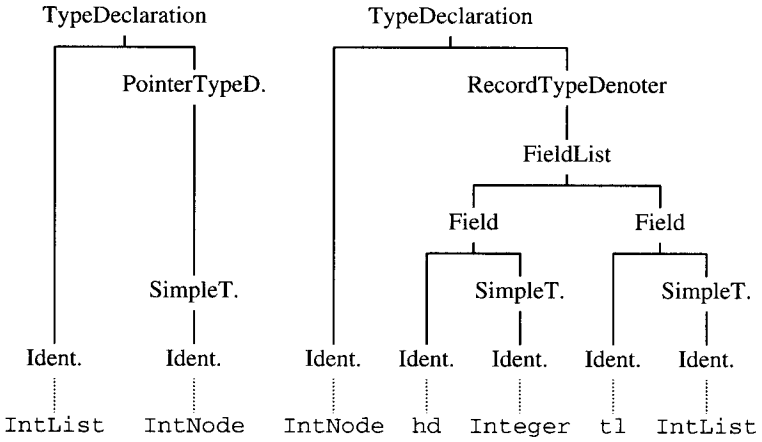
**5.4** Constructor `IdentificationTable` would make the table contain a single (empty) column. Method `enter` would add the new entry to the leftmost column, and to the row indexed by the given identifier. Method `retrieve` would access the first entry in the row indexed by the given identifier. Method `openScope` would insert a new leftmost (empty) column. Method `closeScope` would remove the leftmost column.

This implementation would be more efficient than the simple stack implementation or the stack of binary trees, because retrieval would be essentially a constant-time operation. In terms of time complexity, retrieval would be $O(1)$ rather than $O(n)$ or $O(\log n)$. (This assumes that identifiers are hashed to index the table rows efficiently.)

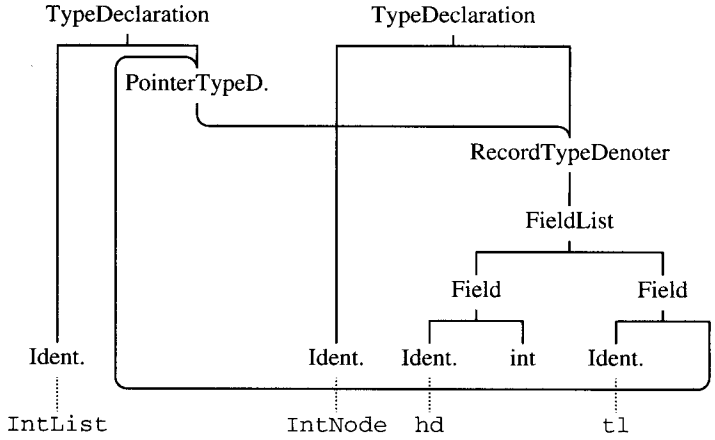
**5.7** Since each type is represented by an *object*, simply compare the object references. This will require changing the existing type checking code from using the `equals` method to using the '`==`' operator. For example, the code for `visitAssignCommand` in Example 5.11 would become:

```
public Object visitAssignCommand
                    (AssignCommand com,
                     Object arg) {
  Type vType = (Type) com.V.visit(this, null);
  Type eType = (Type) com.E.visit(this, null);
  if (! com.V.variable)
      ... // Report an error – the left side is not a variable.
  if (eType != vType)
      ... // Report an error – the left and right sides
          // are not of equivalent type.
  return null;
}
```

**5.9** Undecorated AST:




After elimination of type identifiers:




The AST has been transformed to a directed graph, with the mutually recursive types giving rise to a cycle.

The complication is that the `equals` method must be able to compare two (possibly cyclic) graphs for structural equivalence. It must be implemented carefully to avoid nontermination.

**5.10** Consider the function call '*I* (*E*)'. Check that *I* has been declared by a function declaration, say '`func` *I* (*I'*: *T'*) : *T* ~ *E'*'. Check that the type of the actual parameter *E* is equivalent to the formal parameter type *T'*. Infer that the type of the function call is *T*.

# Answers 6

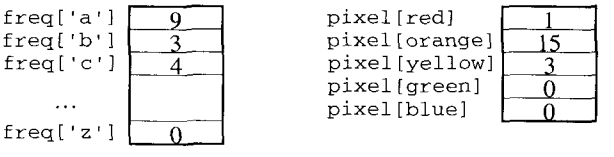
**6.3** Advantage of single-word representation:

- It is economical in storage.

Advantages of double-word representation:

- It is closer to the mathematical (unbounded) set of integers.
- Overflow is less likely.

**6.5** (a)




(b) Every $T_{\text{index}}$ has a minimum value, *min* $T_{\text{index}}$; a maximum value, *m* $T_{\text{index}}$; and an *ord* function that maps the values of the type consecutive integers. Thus:

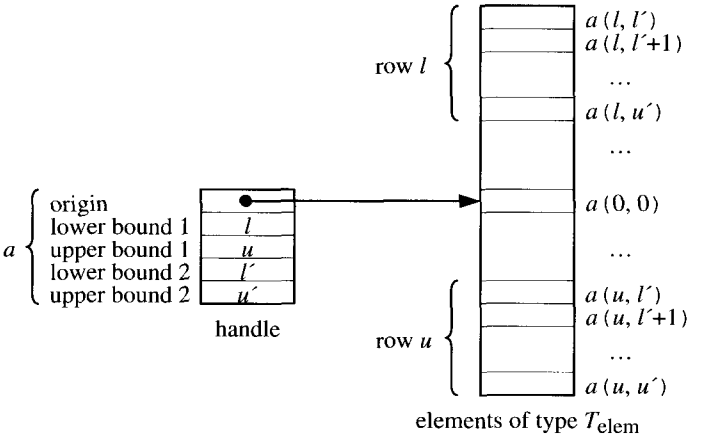
$$\begin{aligned} size\ T &= (u - l + 1) \times size\ T_{\text{elem}} \\ address[\![\ a\,[\,0\,]\ ]\!] &= address\ a - (l \times size\ T_{\text{elem}}) \\ address[\![\ a\,[\,i\,]\ ]\!] &= address[\![\ a\,[\,0\,]\ ]\!] + (ord\ (i) \times size\ T_{\text{elem}}) \end{aligned}$$

where $l = ord(min\ T_{\text{index}})$ and $u = ord(max\ T_{\text{index}})$.

**6.6** For two-dimensional arrays:

$$\begin{aligned} size\ T &= m \times n \times size\ T_{\text{elem}} \\ address[\![\ a\,[\,i\,]\,[\,j\,]\ ]\!] &= address\ a + (i \times (n \times size\ T_{\text{elem}})) + \\ &\qquad (j \times size\ T_{\text{elem}}) \end{aligned}$$

**6.8** Make the handle contain the lower and upper bounds in both dimensions, well as a pointer to the elements. Store the elements themselves row by row ( in Example 6.6). If $l$, $u$, $l'$, and $u'$ are the values of $E_1$, $E_2$, $E_3$, and $E_4$, respe tively, then we get:




**6.11** (a) Evaluate subexpression '`1 - (c * 2)`' before '`a * b`':

```
LOAD R1 c
MULT R1 #2
LOAD R2 #1
SUB  R2 R1
LOAD R1 a
MULT R1 b
ADD  R1 R2
```

(b) Save the accumulator's contents to a temporary location (say *temp*) whenever the accumulator is needed to evaluate something else:

```
LOAD  c
MULT  #2
STORE temp
LOAD  #1
SUB   temp
STORE temp
LOAD  a
MULT  b
ADD   temp
```

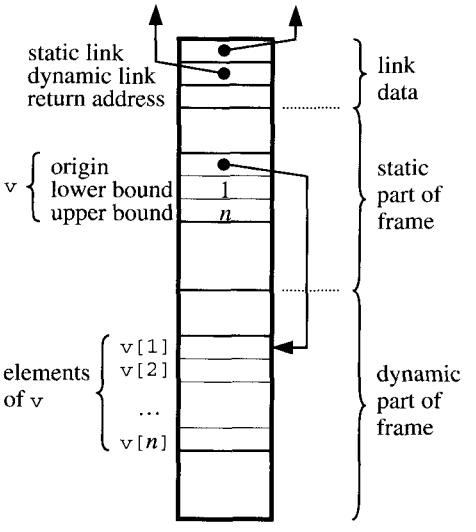
(In general, more than one temporary location might be needed.)

**6.13** Address of global variable $v_i$ is:

$$address\ v_i = size\ T_1 + \ldots + size\ T_{i-1}$$

Only the addresses allocated to the variables are affected by the order of the variable declarations. The *net* behavior of the object program is not affected.

**6.16** Let each frame consist of a static part and a dynamic part. The static part accommodates variables of primitive type, and the handles of dynamic arrays. The dynamic part expands as necessary to accommodate elements of dynamic arrays. The frame containing v would look like this:




Since everything in the static part is of constant size, the compiler can determine each variable's address relative to the frame base. This is not true for the dynamic part, but the array elements there are always addressed indirectly through the handles.

**6.17** There are three cases of interest. If $n = m+1$, $S$ is local to the caller. If $n = m$, $S$ is at the same level as the caller. If $n < m$, $S$ encloses the caller.

(a) On call, push $S$'s frame on to the stack. In all cases, set D$n$ to point to the base of the new frame. (*Note:* If $n < m$, D($n$+1), ..., and D$m$ become undefined.)

(b) On return, pop $S$'s frame off the stack. If $n = m+1$, do nothing else. If $n = m$, reset D$m$ to point to the base of the (now) topmost frame. If $n < m$, reset other display registers using the static links: D($m$–1) ← *content* (D$m$); ...; D$n$ ← *content* (D($n$+1)). (*Note:* If $n = m+1$, D$n$ becomes undefined.)

There is no need to change D0, D1, ..., D($n$–1) at either the call or the return, since these registers point to the same frames before, during, and after the activation of $S$.

Advantages and disadvantages (on the assumption that D0, D1, etc., are all true registers):

•• Nonlocal variables can be accessed as efficiently as local or global variables.

- The display registers must be updated at every call and return.
- The caller must reset the display registers after return, since $S$ does not know where it was called from, i.e., it does not know $m$.
- Complications arise when $S$ is an unknown routine (e.g., an argument), because then the caller does not know $n$.

**6.19** (a)

| *Stack machine code* | *Register machine code* |
|---|---|
| `LOAD a` | `LOAD R1 a` |
| `LOAD b` | `LOAD R2 b` |
| `LOAD c` | `LOAD R3 c` |
| `CALL f` | `CALL f` |

(b)

| | |
|---|---|
| `LOAD a` | `LOAD R1 a` |
| `CALL g` | `CALL g` |
| | `STORE R0 temp` |
| `LOAD b` | `LOAD R1 b` |
| `LOAD c` | `LOAD R2 c` |
| `CALL h` | `CALL h` |
| | `LOAD R2 R0` |
| `LOAD d` | `LOAD R3 d` |
| | `LOAD R1 temp` |
| `CALL f` | `CALL f` |

**6.23** The following algorithm uses an address translation table, in which each entry $(a, a')$ consists of a heap variable's old address $a$ together with its new address $a'$ after compaction.

Procedure to compact the heap:
  calculate the new address of all heap variables;
  move and adjust all heap variables;
  adjust the heap top HT;
  adjust all pointers in the stack.

Procedure to calculate the new address of all heap variables:
  set the current new address `newAddr` to HB;
  for each heap variable `hv`, in order of distance from HB:
    subtract the size of `hv` from `newAddr`;
    add an entry for (address of `hv`, `newAddr`) to the table.

```
Procedure to move and adjust all heap variables:
    for each heap variable hv, in order of distance from HB:
        for each pointer p in hv:
            find an entry (p, q) in the table;
            replace p by q;
        find an entry (address of hv, r) in the table;
        copy hv to address r.

Procedure to adjust all pointers in the stack:
    for each pointer p in the stack:
        find an entry (p, q) in the table;
        replace p by q.
```

This algorithm assumes that the heap variables are linked (by hidden fields) i order of increasing distance from HB.

# Answers 7

**7.1** Code template (7.8e) gives rise to object code in which one jump instruction i executed per iteration. The code template of Exercise 7.1 gives rise to objec code in which two jump instructions are executed per iteration, which i slower.

The code template of Exercise 7.1 is commonly used because it is suitable fo one-pass compilation.

**7.2** (a) *execute*⟦$V_1$ , $V_2$ := $E_1$ , $E_2$⟧ =

```
        evaluate E1
        evaluate E2
        assign V2
        assign V1
```

(b) *execute*⟦$C_1$ , $C_2$⟧ =

```
        execute C1
        execute C2
```

(c) *execute*⟦`if` *E* `then` *C*⟧ =

```
        evaluate E
        JUMPIF(0) g
        execute C
    g:
```

(d) *execute*⟦`repeat` *C* `until` *E*⟧ =

```
    g:  execute C
        evaluate E
        JUMPIF(0) g
```

(e) *execute*⟦`repeat` $C_1$ `while` $E$ `do` $C_2$⟧ =

```
        JUMP  h
g:      execute C2
h:      execute C1
        evaluate E
        JUMPIF(1)  g
```

**7.3** (a) *evaluate*⟦`if` $E_1$ `then` $E_2$ `else` $E_3$⟧ =

```
        evaluate E1
        JUMPIF(0)  g
        evaluate E2
        JUMP  h
g:      evaluate E3
h:
```

(b) *evaluate*⟦`let` $D$ `in` $E$⟧ =

```
        elaborate D
        evaluate E
        POP(n)  s       if s > 0
```

where $s$ = amount of storage allocated by $D$,
$n$ = *size* (type of $E$)

(c) *evaluate*⟦`begin` $C$ `;` `yield` $E$ `end`⟧ =

```
        execute C
        evaluate E
```

**7.5** Selected encoding methods:

(a)
```
public Object visitSimAssignCommand
             (SimAssignCommand com, Object arg) {
   com.E1.visit(this, arg);
   com.E2.visit(this, arg);
   encodeAssign(com.V2);
   encodeAssign(com.V1);
   return null;
}
```

(c)
```
public Object visitIfOnlyCommand
             (IfOnlyCommand com, Object arg) {
   com.E.visit(this, arg);
   short i = nextInstAddr;
   emit(Instruction.JUMPIFop, 0,
        Instruction.CBr, 0);
   com.C.visit(this, arg);
   short g = nextInstrAddr;
   patch(i, g);
   return null;
}
```

(d)
```
public Object visitRepeatCommand
                  (RepeatCommand com, Object arg) {
   short g = nextInstrAddr;
   com.C.visit(this, arg);
   com.E.visit(this, arg);
   emit(Instruction.JUMPIFop, 0,
            Instruction.CBr, g);
   return null;
}
```

**7.7** (a) The most efficient solution is:

*execute*⟦`for` *I* `from` $E_1$ `to` $E_2$ `do` *C*⟧ =

| | | |
|---|---|---|
| | *evaluate* $E_2$ | – compute final value |
| | *evaluate* $E_1$ | – compute initial value of *I* |
| | `JUMP` *h* | |
| *g*: | *execute C* | |
| | `CALL` *succ* | – increment current value of *I* |
| *h*: | `LOAD -1[ST]` | – fetch current value of *I* |
| | `LOAD -3[ST]` | – fetch final value |
| | `CALL` *le* | – test current value ≤ final value |
| | `JUMPIF(1)` *g* | – if so, repeat |
| | `POP(0) 2` | – discard current and final values |

At *g* and at *h*, the current value of *I* is at the stack top (at address `-1[ST]`), and the final value is immediately underlying (at address `-2[ST]`).

(b) This solution requires the introduction of two new AST classes. The first is a Command AST used to represent the for-command itself. The second is a Declaration AST used to represent the (pseudo-)declaration of the for-command control variable. This is because the identification table stores Declaration ASTs as attributes.

```
public class ForCommand extends Command {
   ...
   // Declaration of control variable...
   public ForDeclaration D;

   // Subphrases of for-command...
   public Expression E1, E2;
   public Command C;
}

////////////////////////////////////////////////
```

```
public class ForDeclaration
                extends Declaration {
  // The contextual analyzer links all applied occurrences of the
  // control variable to the ForDeclaration AST in the
  // corresponding ForCommand AST.
  ...
  public Identifier I;
}

////////////////////////////////////////////

public Object visitForCommand
              (ForCommand com,
               Object arg) {
  short gs = shortValueOf(arg);
  com.D.entity = new UnknownValue(1,
                          gs + 1);
  com.E2.visit(this, arg);
  com.E1.visit(this, new short(gs + 1));
  short i = nextInstrAddr;
  emit(Instruction.JUMPop, 0,
       Instruction.CBr, 0);
  short g = nextInstrAddr;
  com.C.visit(this, new short(gs + 2));
  emit(Instruction.CALLop, Instruction.SBr,
       Instruction.PBr,
       address of primitive routine succ);
  short h = nextInstrAddr;
  patch(i, h);
  emit(Instruction.LOADop, 1,
       Instruction.STr, -1);
  emit(Instruction.LOADop, 1,
       Instruction.STr, -3);
  emit(Instruction.CALLop, Instruction.SBr,
       Instruction.PBr, address of primitive routine le);
  emit(Instruction.JUMPIFop, 1,
       Instruction.CBr, g);
  emit(Instruction.POPop, 0, 0, 2);
  return null;
}
```

The `ForCommand` visitor/encoding method first creates a run-time entity description for the control variable *I*, and attaches it to the corresponding `ForDeclaration`. Provided that the contextual analyzer has linked each applied occurrence of the control variable to the `For-Declaration`, the loop body *C* will be able to fetch (but not assign to) the control variable.

**7.10** (a) Reserve space for the result variable just above the link data in the function's frame (i.e., at address `3[LB]`):

*elaborate*⟦`func` *I* `(` *FP* `)` `:` *T* `~` *C*⟧ =

```
        JUMP  g
  e:    PUSH  n                 where n = size T
        execute C
        RETURN(n)  d            where d = size of FP
  g:
```

*execute*⟦`result` *E*⟧ =

```
        evaluate E
        STORE(n)  3[LB]         where n = size (type of E)
```

(b)

```
public Object visitFuncDeclaration
                  (FuncDeclaration decl,
                   Object arg) {
    Frame f = (Frame) arg;
    short i = nextInstrAddr;
    emit(Instruction.JUMPop, 0,
          Instruction.CBr, 0);
    short e = nextInstrAddr;
    decl.entity =
          new KnownRoutine(2, f.level, e);
    Frame f1 = new Frame(f.level + 1, 0);
    short d = shortValueOf(
          decl.FP.visit(this, f1));
    // ... creates a run-time entity for the formal parameter,
    // and returns the size of the parameter.
    short n = shortValueOf(
          decl.T.visit(this, null));
    emit(Instruction.PUSHop, 0, 0, n);
    Frame f2 = new Frame(f.level + 1, 3 + n);
    decl.C.visit(this, f2);
    emit(Instruction.RETURNop, n, 0, d);
    short g = nextInstrAddr;
    patch(i, g);
    return new Short(0);
}
```

```
public Object visitResultCommand
              (ResultCommand com,
               Object arg) {
  short n =
        shortValueOf(com.E.visit(this, arg));
  emit(Instruction.STOREop, n,
        Instruction.LBr, 3);
  return null;
}
```

# Answers 8

**8.3** In outline:

```
public abstract class UserCommand {
  public abstract void perform
                (HypoInterpreter interp);
}

////////////////////////////////////////////////

public class StepCommand extends UserCommand {
  public void perform
           (HypoInterpreter interp) {
     interp.step();
  }
}

////////////////////////////////////////////////

public class RunCommand extends UserCommand {
  public void perform
           (HypoInterpreter interp) {
     do {
        interp.step();
     } while (! interp.break[interp.CP]
           && (interp.status ==
              HypoState.RUNNING));
  }
}

////////////////////////////////////////////////
```

```
public class ShowRegistersCommand
             extends UserCommand {
  ...
}

////////////////////////////////////////////////

public class ShowDataStoreCommand
             extends UserCommand {
  ...
}

////////////////////////////////////////////////

public class TerminateCommand
             extends UserCommand {
  public void perform
           (HypoInterpreter interp) {
     interp.status = HypoState.HALTED;
  }
}

////////////////////////////////////////////////

public class ToggleBreakpointCommand
             extends UserCommand {

  public short point; // The breakpoint address to toggle

  public void perform
           (HypoInterpreter interp) {
     interp.break[this.point] =
        ! interp.break[this.point];
  }
}

////////////////////////////////////////////////

public class HypoInterpreter extends HypoState {
  ...
  public static boolean[] break =
             new boolean[CODESIZE];
  UserCommand command;

  private void clearBreakpoints () {
     for (int d = 0; d < CODESIZE; d++)
        break[d++] = false;
  }
```

```
    private UserCommand interact () {
        ... // Obtain a command from the user.
    }

    public void step () {
        // Fetch the next instruction ...
        HypoInstruction instr = code[PC++];
        // Analyze this instruction ...
        ...
        // Execute this instruction ...
        ...
    }

    public void emulate () {
        // initialize ...
        PC = 0; ACC = 0; status = RUNNING;
        clearBreakpoints();

        do {
            command = interact();
            command.perform(this);
        } while (status == RUNNING);
    }
}
```

**8.6** (a) Advantages and disadvantages of storing commands as *text*:

- Loading and editing are easy.
- Commands have to be scanned and parsed whenever fetched for exe tion, which is slow.

Advantages and disadvantages of storing commands as *tokens*:

- Loading and editing are fairly easy.
- Commands have to be parsed whenever fetched for execution, which moderately slow.

Advantages and disadvantages of storing commands as *ASTs*:

- Commands have to be scanned and parsed when loaded.
- Editing is awkward.
- Commands are immediately ready for execution.

(b) Answer implied by the above.

**8.8** In outline:

```
public class MiniShell extends MiniShellState {

  public MiniShellCommand readAnalyze () {
     // Read and analyze the next command from the user.
     ...
  }

  public MiniShellCommand readAnalyze
                  (FileInputStream script) {
     // Read and analyze the next command from file script.
     ...
  }

  public void execute (MiniShellCommand com) {

     if (com.name.equals("create")) {
        ...
     }

     else if (com.name.equals("call")) {
        File input = new File(com.args[0]);
        FileInputStream script =
              new FileInputStream(input);
        while (more commands in script) {
           MiniShellCommand subCom =
                 readAnalyze(script);
           execute(subCom);
        }

     } else  // executable program
        exec(com.name, com.args);
  }

  public void interpret () {

     // Initialize ...
     status = RUNNING;

     do {

        // Fetch and analyze the next instruction ...
        MiniShellCommand com = readAnalyze();

        // Execute this instruction ...
        execute(com);

     } while (status == RUNNING);
  }
}
```

# Answers 9

**9.5** In outline:

Common subexpressions are: 'i < j' at points (1); the *address* of a[i] points (2); the *address* of a[j] at points (3); the *address* of a[n] at points (4

```
var a: array ... of Integer
...
i := m - 1; j := n; pivot := a[n];
while i < j(1) do
  begin
  i := i + 1;
  while a[i](2) < pivot do i := i + 1;
  j := j - 1;
  while a[j](3) > pivot do j := j - 1;
  if i < j(1) then
     begin
     t := a[i](2);
     a[i](2) := a[j](3);
     a[j](3) := t
     end
  end;
t := a[i](2);
a[i](2) := a[n](4);
a[n](4) := t
```

APPENDIX B

# Informal Specification of the Programming Language Triangle

## B.1 Introduction

Triangle is a regularized extensible subset of Pascal. It has been designed as a model language to assist in the study of the concepts, formal specification, and implementation of programming languages.

The following sorts of entity can be declared and used in Triangle:

- A *value* is a truth value, integer, character, record, or array.
- A *variable* is an entity that may contain a value and that can be updated. Each variable has a well-defined lifetime.
- A *procedure* is an entity whose body may be executed in order to update variables. A procedure may have constant, variable, procedural, and functional parameters.
- A *function* is an entity whose body may be evaluated in order to yield a value. A function may have constant, variable, procedural, and functional parameters.
- A *type* is an entity that determines a set of values. Each value, variable, and function has a specific type.

Each of the following sections specifies part of the language. The subsection headed **Syntax** specifies its grammar in BNF (except for Section B.8 which uses EBNF). The subsection headed **Semantics** informally specifies the semantics (and contextual constraints) of each syntactic form. Finally, the subsection headed **Examples** illustrates typical usage.

## B.2 Commands

A command is executed in order to update variables. (This includes input–output.)

## Syntax

A single-command is a restricted form of command. (A command must be enclosed between `begin` ... `end` brackets in places where only a single-command is allowed.)

```
Command         ::=  single-Command
                   |  Command ; single-Command

single-Command  ::=
                   |  V-name := Expression
                   |  Identifier ( Actual-Parameter-Sequence )
                   |  begin Command end
                   |  let Declaration in single-Command
                   |  if Expression then single-Command
                         else single-Command
                   |  while Expression do single-Command
```

(The first form of single-command is empty.)

## Semantics

- The skip command ' ' has no effect when executed.
- The assignment command '*V* := *E*' is executed as follows. The expression *E* is evaluated to yield a value; then the variable identified by *V* is updated with this value. (The types of *V* and *E* must be equivalent.)
- The procedure calling command '*I*(*APS*)' is executed as follows. The actual-parameter-sequence *APS* is evaluated to yield an argument list; then the procedure bound to *I* is called with that argument list. (*I* must be bound to a procedure. *APS* must be compatible with that procedure's formal-parameter-sequence.)
- The sequential command '$C_1$ ; $C_2$' is executed as follows. $C_1$ is executed first; then $C_2$ is executed.
- The bracketed command '`begin` *C* `end`' is executed simply by executing *C*.
- The block command '`let` *D* `in` *C*' is executed as follows. The declaration *D* is elaborated; then *C* is executed, in the environment of the block command overlaid by the bindings produced by *D*. The bindings produced by *D* have no effect outside the block command.
- The if-command '`if` *E* `then` $C_1$ `else` $C_2$' is executed as follows. The expression *E* is evaluated; if its value is true, then $C_1$ is executed; if its value is false, then $C_2$ is executed. (The type of *E* must be `Boolean`.)
- The while-command '`while` *E* `do` *C*' is executed as follows. The expression *E* is evaluated; if its value is true, then *C* is executed, and then the while-command is executed again; if its value is false, then execution of the while-command is completed. (The type of *E* must be `Boolean`.)

### Examples

The following examples assume the standard environment (Section B.9), and also the following declarations:

```
var i: Integer;
var s: array 8 of Char;
var t: array 8 of Char;

proc sort (var a: array 8 of Char) ~ ...
```

(a)
```
s[i] := '*'; t := s
```

(b)
```
getint(var i); putint(i); puteol()
```

(c)
```
sort(var s)
```

(d)
```
if s[i] > s[i+1] then
   let var c : Char
   in
      begin
      c := s[i]; s[i] := s[i+1]; s[i+1] := c
      end
else ! skip
```

(e)
```
i := 7;
while (i > 0) /\ (s[i] = ' ') do
   i := i - 1
```

# B.3 Expressions

An expression is evaluated to yield a value. A record-aggregate is evaluated to construct a record value from its component values. An array-aggregate is evaluated to construct an array value from its component values.

### Syntax

A secondary-expression and a primary-expression are progressively more restricted forms of expression. (An expression must be enclosed between parentheses in places where only a primary-expression is allowed.)

| | | |
|---|---|---|
| Expression | ::= | secondary-Expression |
| | \| | **let** Declaration **in** Expression |
| | \| | **if** Expression **then** Expression **else** Expression |
| secondary-Expression | ::= | primary-Expression |
| | \| | secondary-Expression Operator primary-Expression |

```
primary-Expression    ::=  Integer-Literal
                         |  Character-Literal
                         |  V-name
                         |  Identifier ( Actual-Parameter-Sequence )
                         |  Operator primary-Expression
                         |  ( Expression )
                         |  { Record-Aggregate }
                         |  [ Array-Aggregate ]
Record-Aggregate      ::=  Identifier ~ Expression
                         |  Identifier ~ Expression , Record-Aggregate
Array-Aggregate       ::=  Expression
                         |  Expression , Array-Aggregate
```

## Semantics

- The expression '*IL*' yields the value of the integer-literal *IL*. (The type of the expression is `Integer`.)
- The expression '*CL*' yields the value of the character-literal *CL*. (The type of the expression is `Char`.)
- The expression '*V*', where *V* is a value-or-variable-name, yields the value identified by *V*, or the current value of the variable identified by *V*. (The type of the expression is the type of *V*.)
- The function calling expression '*I*(*APS*)' is evaluated as follows. The actual-parameter-sequence *APS* is evaluated to yield an argument list; then the function bound to *I* is called with that argument list. (*I* must be bound to a function. *APS* must be compatible with that function's formal-parameter-sequence. The type of the expression is the result type of that function.)
- The expression '*O E*' is, in effect, equivalent to a function call '*O*(*E*)'.
- The expression '$E_1$ *O* $E_2$' is, in effect, equivalent to a function call '*O*($E_1$, $E_2$)'.
- The expression '(*E*)' yields just the value yielded by *E*.
- The block expression '`let` *D* `in` *E*' is evaluated as follows. The declaration *D* is elaborated; then *E* is evaluated, in the environment of the block expression overlaid by the bindings produced by *D*. The bindings produced by *D* have no effect outside the block expression. (The type of the expression is the type of *E*.)
- The if-expression '`if` $E_1$ `then` $E_2$ `else` $E_3$' is evaluated as follows. The expression $E_1$ is evaluated; if its value is true, then $E_2$ is evaluated; if its value is false, then $E_3$ is evaluated. (The type of $E_1$ must be `Boolean`. The type of the expression is the same as the types of $E_2$ and $E_3$, which must be equivalent.)

- The expression '{*RA*}' yields just the value yielded by the record-aggregate *RA*. (The type of '{$I_1$ ~ $E_1$, ..., $I_n$ ~ $E_n$}' is '`record` $I_1$`:` $T_1$`,` ... `,` $I_n$`:` $T_n$ `end`', where the type of each $E_i$ is $T_i$. The identifiers $I_1$, ..., $I_n$ must all be distinct.)
- The expression '`[`*AA*`]`' yields just the value yielded by the array-aggregate *AA*. (The type of '`[`$E_1$`,` ... `,` $E_n$`]`' is '`array` *n* `of` *T*', where the type of every $E_i$ is *T*.)
- The record-aggregate '*I* ~ *E*' yields a record value, whose only field has the identifier *I* and the value yielded by *E*.
- The record-aggregate '*I* ~ *E* , *RA*' yields a record value, whose first field has the identifier *I* and the value yielded by *E*, and whose remaining fields are those of the record value yielded by *RA*.
- The array-aggregate '*E*' yields an array value, whose only component (with index 0) is the value yielded by *E*.
- The array-aggregate '*E* , *AA*' yields an array value, whose first component (with index 0) is the value yielded by *E*, and whose remaining components (with indices 1, 2, ...) are the components of the array value yielded by *AA*.

## Examples

The following examples assume the standard environment (Section B.9), and also the following declarations:

```
var current: Char;
type Date ~ record
              y: Integer, m: Integer, d: Integer
            end;
var today: Date;

func multiple (m: Integer, n: Integer) : Boolean ~
        ...;

func leap (yr: Integer) : Boolean ~ ...
```

(a)
```
{y ~ today.y + 1, m ~ 1, d ~ 1}
```

(b)
```
[31, if leap(today.y) then 29 else 28,
 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

(c)
```
eof()
```

(d)
```
(multiple(yr, 4) /\ \multiple(yr, 100))
   \/ multiple(yr, 400)
```

(e)
```
let
   const shift ~ ord('a') - ord('A');
   func capital (ch : Char) : Boolean ~
            (ord('A') <= ord(ch))
               /\ (ord(ch) <= ord('Z'))
```

```
in
   if capital(current)
   then chr(ord(current) + shift)
   else current
```

# B.4 Value-or-variable names

A value-or-variable-name identifies a value or variable.

## Syntax

| | | |
|---|---|---|
| V-name | ::= | Identifier |
| | \| | V-name . Identifier |
| | \| | V-name [ Expression ] |

## Semantics

- The simple value-or-variable-name '*I*' identifies the value or variable bound to *I*. (*I* must be bound to a value or variable. The type of the value-or-variable-name is the type of that value or variable.)
- The qualified value-or-variable-name '*V*.*I*' identifies the field *I* of the record value or variable identified by *V*. (The type of *V* must be a record type with a field *I*. The type of the value-or-variable-name is the type of that field.)
- The indexed value-or-variable-name '*V*[*E*]' identifies that component, of the array value or variable identified by *V*, whose index is the value yielded by the expression *E*. If the array has no such index, the program fails. (The type of *E* must be `Integer`, and the type of *V* must be an array type. The type of the value-or-variable-name is the component type of that array type.)

## Examples

The following examples assume the standard environment (Section B.9), and also the following declarations:

```
type Date ~ record
               m : Integer, d : Integer
            end;
const xmas ~ {m ~ 12, d ~ 25};
var easter : Date;
var holiday : array 10 of Date
```

(a) `easter`

(b) `xmas`

(c) `xmas.m`

(d) `holiday`

(e) `holiday[7]`

(f) `holiday[2].m`

# B.5 Declarations

A declaration is elaborated to produce bindings. Elaborating a declaration may also have the side effect of creating and updating variables.

## Syntax

A single-declaration is just a restricted form of declaration.

```
Declaration        ::= single-Declaration
                     | Declaration ; single-Declaration

single-Declaration ::= const Identifier ~ Expression
                     | var Identifier : Type-denoter
                     | proc Identifier ( Formal-Parameter-Sequence ) ~
                           single-Command
                     | func Identifier ( Formal-Parameter-Sequence )
                           : Type-denoter ~ Expression
                     | type Identifier ~ Type-denoter
```

## Semantics

- The constant declaration '`const` *I* ~ *E*' is elaborated by binding *I* to the value yielded by the expression *E*. (The type of *I* will be the type of *E*.)
- The variable declaration '`var` *I* : *T*' is elaborated by binding *I* to a newly created variable of type *T*. The variable's current value is initially undefined. The variable exists only during the activation of the block that caused the variable declaration to be elaborated.
- The procedure declaration '`proc` *I* (*FPS*) ~ *C*' is elaborated by binding *I* to a procedure whose formal-parameter-sequence is *FPS* and whose body is the command *C*. The effect of calling that procedure with an argument list is determined as follows: *FPS* is associated with the argument list; then *C* is executed, in the environment of the procedure declaration overlaid by the bindings of the formal-parameters.
- The function declaration '`func` *I* (*FPS*) : *T* ~ *E*' is elaborated by binding *I* to a function whose formal-parameter-sequence is *FPS* and whose body is the expression *E*. The effect of calling that function with an argument list is determined as follows: *FPS* is associated with the argument list; then *E* is evaluated to yield a value, in the

environment of the function declaration overlaid by the bindings of the formal-parameters. (The type of $E$ must be equivalent to the type denoted by $T$.)

- The type declaration '`type` $I$ `~` $T$' is elaborated by binding $I$ to the type denoted by $T$.
- The sequential declaration '$D_1$ `;` $D_2$' is elaborated by elaborating $D_1$ followed by $D_2$, and combining the bindings they produce. $D_2$ is elaborated in the environment of the sequential declaration, overlaid by the bindings produced by $D_1$. ($D_1$ and $D_2$ must not produce bindings for the same identifier.)

### Examples

The following examples assume the standard environment (Section B.9):

(a)
```
const minchar ~ chr(0)
```

(b)
```
var name: array 20 of Char;
var initial: Char
```

(c)
```
proc inc (var n: Integer) ~ n := n + 1
```

(d)
```
func odd (n: Integer) : Boolean ~
         (n // 2) \= 0
```

(e)
```
func power (a: Integer, n: Integer) : Integer ~
         if n = 0
         then 1
         else a * power(a, n - 1)
```

(f)
```
type Rational ~
         record num: Integer, den: Integer end
```

## B.6 Parameters

Formal-parameters are used to parameterize a procedure or function with respect to (some of) the free identifiers in its body. On calling a procedure or function, the formal-parameters are associated with the corresponding arguments, which may be values, variables, procedures, or functions. These arguments are yielded by actual-parameters.

### Syntax

```
Formal-Parameter-Sequence
                ::=
                  |   proper-Formal-Parameter-Sequence
proper-Formal-Parameter-Sequence
                ::=   Formal-Parameter
                  |   Formal-Parameter , proper-Formal-Parameter-Sequence
```

```
Formal-Parameter  ::=  Identifier : Type-denoter
                   |   var Identifier : Type-denoter
                   |   proc Identifier ( Formal-Parameter-Sequence )
                   |   func Identifier ( Formal-Parameter-Sequence )
                           : Type-denoter

Actual-Parameter-Sequence
                  ::=
                   |   proper-Actual-Parameter-Sequence

proper-Actual-Parameter-Sequence
                  ::=  Actual-Parameter
                   |   Actual-Parameter , proper-Actual-Parameter-Sequence

Actual-Parameter  ::=  Expression
                   |   var V-name
                   |   proc Identifier
                   |   func Identifier
```

(The first form of actual-parameter-sequence and the first form of formal-parameter-sequence are empty.)

## Semantics

- A formal-parameter-sequence '$FP_1$, ..., $FP_n$' is associated with a list of arguments, by associating each $FP_i$ with the $i$th argument. The corresponding actual-parameter-sequence '$AP_1$, ..., $AP_n$' yields a list of arguments, with each $AP_i$ yielding the $i$th argument. (The number of actual-parameters must equal the number of formal-parameters, and each actual-parameter must be compatible with the corresponding formal-parameter. The actual-parameter-sequence must be empty if the formal-parameter-sequence is empty .)
- The formal-parameter '*I* : *T*' is associated with an argument value by binding *I* to that argument. The corresponding actual-parameter must be of the form '*E*', and the argument value is obtained by evaluating *E*. (The type of *E* must be equivalent to the type denoted by *T*.)
- The formal-parameter '`var` *I* : *T*' is associated with an argument variable by binding *I* to that argument. The corresponding actual-parameter must be of the form '`var` *V*', and the argument variable is the one identified by *V*. (The type of *V* must be equivalent to the type denoted by *T*.)
- The formal-parameter '`proc` *I* ( *FPS* )' is associated with an argument procedure by binding *I* to that argument. The corresponding actual-parameter must be of the form '`proc` *I*', and the argument procedure is the one bound to *I*. (*I* must be bound to a procedure, and that procedure must have a formal-parameter-sequence equivalent to *FPS*.)
- The formal-parameter '`func` *I* ( *FPS* ) : *T*' is associated with an argument function by binding *I* to that argument. The corresponding actual-parameter must be of the

form '`func` *I*', and the argument function is the one bound to *I*. (*I* must be bound to a function, and that function must have a formal-parameter-sequence equivalent to *FPS* and a result type equivalent to the type denoted by *T*.)

## Examples

The following examples assume the standard environment (Section B.9):

(a)
```
while \eol() do
   begin get(var ch); put(ch) end;
geteol(); puteol()
```

(b)
```
proc increment (var count: Integer) ~
          count := count + 1
...
increment(var freq[n])
```

(c)
```
func uppercase (letter: Char) : Char ~
          if (ord('a') <= ord(letter))
                /\ (ord(letter) <= ord('z'))
          then chr(ord(letter)-ord('a')+ord('A'))
          else letter

...
if uppercase(request) = 'Q' then quit
```

(d)
```
type Point ~ record x: Integer, y: Integer end;
proc shiftright (var pt: Point, xshift: Integer) ~
          pt.x := pt.x + xshift
...
shiftright(var penposition, 10)
```

(e)
```
proc iteratively (proc p (n: Integer),
                  var a: array 10 of Integer) ~
          let var i: Integer
          in
             begin
             i := 0;
             while i < 10 do
                begin p(a[i]); i := i + 1 end
             end;
var v : array 10 of Integer
...
iteratively(proc putint, var v)
```

# B.7 Type-denoters

A type-denoter denotes a data type. Every value, constant, variable, and function has a specified type.

A record-type-denoter denotes the structure of a record type.

## Syntax

| | | |
|---|---|---|
| Type-denoter | ::= | Identifier |
| | \| | **array** Integer-Literal **of** Type-denoter |
| | \| | **record** Record-Type-denoter **end** |
| Record-Type-denoter | ::= | Identifier **:** Type-denoter |
| | \| | Identifier **:** Type-denoter **,** Record-Type-denoter |

## Semantics

- The type-denoter '*I*' denotes the type bound to *I*.
- The type-denoter '`array` *IL* `of` *T*' denotes a type whose values are arrays. Each array value of this type has an index range whose lower bound is zero and whose upper bound is one less than the integer-literal *IL*. Each array value has one component of type *T* for each value in its index range.
- The type-denoter '`record` *RT* `end`' denotes a type whose values are records. Each record value of this type has the record structure denoted by *RT*.
- The record-type-denoter '*I* : *T*' denotes a record structure whose only field has the identifier *I* and the type *T*.
- The record-type-denoter '*I* : *T*, *RT*' denotes a record structure whose first field has the identifier *I* and the type *T*, and whose remaining fields are determined by the record structure denoted by *RT*. *I* must not be a field identifier of *RT*.

(Type equivalence is structural:

- Two primitive types are equivalent if and only if they are the same type.
- The type `record` ..., $I_i$`:` $T_i$, ... `end` is equivalent to `record` ..., $I_i'$`:` $T_i'$, ... `end` if and only if each $I_i$ is the same as $I_i'$ and each $T_i$ is equivalent to $T_i'$.
- The type `array` *n* `of` *T* is equivalent to `array` *n'* `of` *T'* if and only if $n = n'$ and *T* is equivalent to *T'*.)

## Examples

(a) `Boolean`

(b) `array 80 of Char`

(c)
```
record y: Integer, m: Month, d: Integer end
```

(d)
```
record
    size: Integer,
    entry: array 100 of
               record
                   name: array 20 of Char,
                   number: Integer
               end
end
```

# B.8 Lexicon

At the lexical level, the program text consists of tokens, comments, and blank space.

The tokens are literals, identifiers, operators, various reserved words, and various punctuation marks. No reserved word may be chosen as an identifier.

Comments and blank space have no significance, but may be used freely to improve the readability of the program text. However, two consecutive tokens that would otherwise be confused must be separated by comments and/or blank space.

## Syntax

| | | |
|---|---|---|
| Program | ::= | (Token \| Comment \| Blank)* |
| Token | ::= | Integer-Literal \| Character-Literal \| Identifier \| Operator \| **array** \| **begin** \| **const** \| **do** \| **else** \| **end** \| **func** \| **if** \| **in** \| **let** \| **of** \| **proc** \| **record** \| **then** \| **type** \| **var** \| **while** \| **.** \| **:** \| **;** \| **,** \| **:=** \| **~** \| **(** \| **)** \| **[** \| **]** \| **{** \| **}** |
| Integer-Literal | ::= | Digit Digit* |
| Character-Literal | ::= | **'**Graphic**'** |
| Identifier | ::= | Letter (Letter \| Digit)* |
| Operator | ::= | Op-character Op-character* |
| Comment | ::= | **!** Graphic* end-of-line |
| Blank | ::= | space \| tab \| end-of-line |
| Graphic | ::= | Letter \| Digit \| Op-character \| space \| tab \| **.** \| **:** \| **;** \| **,** \| **~** \| **(** \| **)** \| **[** \| **]** \| **{** \| **}** \| **_** \| **\|** \| **!** \| **'** \| **`** \| **"** \| **#** \| **$** |

```
Letter        ::=  a|b|c|d|e|f|g|h|i|j|k|l|m|
                   n|o|p|q|r|s|t|u|v|w|x|y|z|
                   A|B|C|D|E|F|G|H|I|J|K|L|M|
                   N|O|P|Q|R|S|T|U|V|W|X|Y|Z

Digit         ::=  0|1|2|3|4|5|6|7|8|9

Op-character  ::=  +|-|*|/|=|<|>|\|&|@|%|^|?
```

(*Note:* The symbols space, tab, and end-of-line stand for individual characters that cannot stand for themselves in the syntactic rules.)

### Semantics

- The value of the integer-literal $d_n...d_1d_0$ is $d_n \times 10^n + ... + d_1 \times 10 + d_0$.
- The value of the character-literal `'c'` is the graphic character *c*.
- Every character in an identifier is significant. The cases of the letters in an identifier are also significant.
- Every character in an operator is significant. Operators are, in effect, a subclass of identifiers (but they are bound only in the standard environment, to unary and binary functions).

### Examples

(a) Integer-literals: `0  1987`

(b) Character-literals: `'%'  'Z'  '''`

(c) Identifiers: `x  pi  v101  Integer  get  gasFlowRate`

(d) Operators: `+  *  <=  \/`

## B.9 Programs

A program communicates with the user by performing input–output.

### Syntax

```
Program  ::=  Command
```

### Semantics

- The program '*C*' is run by executing the command *C* in the standard environment.

## Example

```
let
   type Line ~
            record
               length: Integer,
               content: array 80 of Char
            end;
   proc getline (var l: Line) ~
            begin
            l.length := 0;
            while eol() do
               begin
               get(var l.content[l.length]);
               l.length := l.length + 1
               end;
            geteol()
            end;
   proc putreversedline (l: Line) ~
            let var i : Integer
            in
               begin
               i := l.length;
               while i > 0 do
                  begin
                  i := i - 1;
                  put(l.content[i])
                  end;
               puteol()
               end;
   var currentline: Line
in
   while eof() do
      begin
      getline(var currentline);
      putreversedline(currentline)
      end
```

## Standard environment

The standard environment includes the following constant, type, procedure, and function declarations:

```
type Boolean ~ ...; ! truth values

const false ~ ...;  ! the truth value false

const true ~ ...;   ! the truth value true

type Integer ~ ...; ! integers up to maxint in magnitude

const maxint ~ ...; ! implementation-defined maximum integer

type Char ~ ...; ! implementation-defined characters

func \  (b: Boolean) : Boolean ~
          ...; ! not b, i.e., logical negation

func /\ (b1: Boolean, b2: Boolean) : Boolean ~
          ...; ! b1 and b2, i.e., logical conjunction

func \/ (b1: Boolean, b2: Boolean) : Boolean ~
          ...; ! b1 or b2, i.e., logical disjunction

func +  (i1: Integer, i2: Integer) : Integer ~
          ...; ! i1 plus i2,
               ! failing if the result exceeds maxint in magnitude

func -  (i1: Integer, i2: Integer) : Integer ~
          ...; ! i1 minus i2,
               ! failing if the result exceeds maxint in magnitude

func *  (i1: Integer, i2: Integer) : Integer ~
          ...; ! i1 times i2,
               ! failing if the result exceeds maxint in magnitude

func /  (i1: Integer, i2: Integer) : Integer ~
          ...; ! i1 divided by i2, truncated towards zero,
               ! failing if i2 is zero

func // (i1: Integer, i2: Integer) : Integer ~
          ...; ! i1 modulo i2, failing unless i2 is positive

func <  (i1: Integer, i2: Integer) : Boolean ~
          ...; ! true iff i1 is less than i2

func <= (i1: Integer, i2: Integer) : Boolean ~
          ...; ! true iff i1 is less than or equal to i2

func >  (i1: Integer, i2: Integer) : Boolean ~
          ...; ! true iff i1 is greater than i2
```

```
func >= (i1: Integer, i2: Integer) : Boolean ~
        ...; ! true iff i1 is greater than or equal to i2

func chr (i: Integer) : Char ~
        ...; ! character whose internal code is i,
             ! failing if no such character exists

func ord (c: Char) : Integer ~
        ...; ! internal code of c

func eof () : Boolean ~
        ...; ! true iff end-of-file has been reached in input

func eol () : Boolean ~
        ...; ! true iff end-of-line has been reached in input

proc get (var c: Char) ~
        ...; ! read the next character from input and assign it to c,
             ! failing if end-of-file already reached

proc put (c: Char) ~
        ...; ! write character c to output

proc getint (var i: Integer) ~
        ...; ! read an integer literal from input and assign its value
             ! to i, failing if the value exceeds maxint in magnitude,
             ! or if end-of-file is already reached

proc putint (i: Integer) ~
        ...; ! write to output the integer literal whose value is i

proc geteol () ~
        ...; ! skip past the next end-of-line in input,
             ! failing if end-of-file is already reached

proc puteol () ~
        ...; ! write an end-of-line to output
```

In addition, the following functions are available for every type *T*:

```
func =  (val1: T, val2: T) : Boolean ~
        ...; ! true iff val1 is equal to val2

func \= (val1: T, val2: T) : Boolean ~
        ... ! true iff val1 is not equal to val2
```

APPENDIX C

# Description of the Abstract Machine TAM

TAM is an abstract machine whose design makes it especially suitable for executing programs compiled from a block-structured language (such as Algol, Pascal, or Triangle). All evaluation takes place on a stack. Primitive arithmetic, logical, and other operations are treated uniformly with programmed functions and procedures.

## C.1 Storage and registers

TAM has two separate stores:

- *Code Store*, consisting of 32-bit instruction words (read only).
- *Data Store*, consisting of 16-bit data words (read–write).

The layouts of both stores are illustrated in Figure C.1.

Each store is divided into *segments*, whose boundaries are pointed to by dedicated registers. Data and instructions are always addressed relative to one of these registers.

While a program is running, the segmentation of Code Store is fixed, as follows:

- The *code segment* contains the program's instructions. Registers CB and CT point to the base and top of the code segment. Register CP points to the next instruction to be executed, and is initially equal to CB (i.e., the program's first instruction is at the base of the code segment).
- The *primitive segment* contains 'microcode' for elementary arithmetic, logical, input–output, heap, and general-purpose operations. Registers PB and PT point to the base and top of the primitive segment.

While a program is running, the segmentation of Data Store may vary:

- The *stack* grows from the low-address end of Data Store. Registers SB and ST point to the base and top of the stack, and ST is initially equal to SB.
- The *heap* grows from the high-address end of Data Store. Registers HB and HT point to the base and top of the heap, and HT is initially equal to HB.

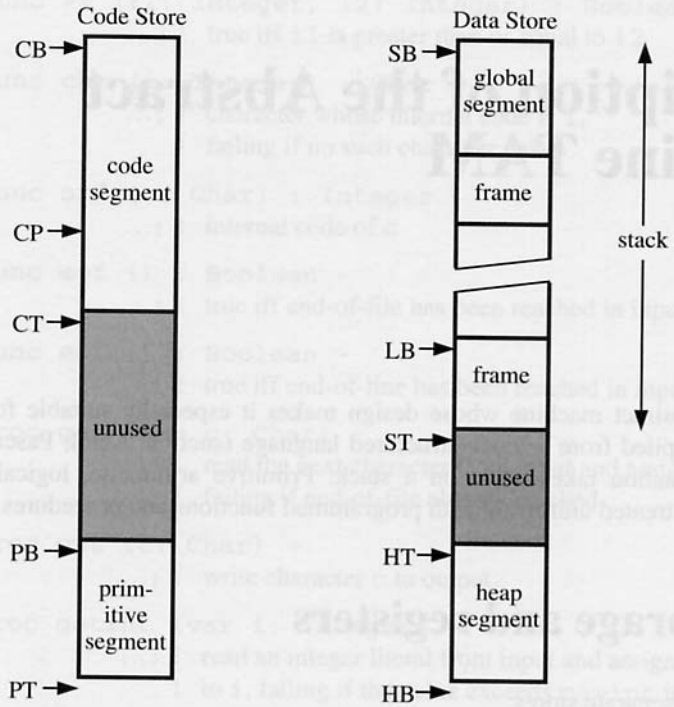



**Figure C.1** Layout of the TAM Code Store and Data Store.

Both stack and heap can expand and contract. Storage exhaustion arises when ST and HT attempt to cross over.

The stack itself consists of one or more segments:

- The *global segment* is always at the base of the stack, and contains global data used by the program.
- The stack may contain any number of other segments, known as *frames*. Each frame contains data local to an activation of some routine. Calling a routine causes a new frame to be pushed on to the stack; return from a routine causes the topmost frame to be popped. The topmost frame may expand and contract, but the underlying frames are (temporarily) fixed in size. Register LB points to the base of the topmost frame.

Figure C.2 shows the outline of a source program in some block-structured language. Figure C.3 shows successive stack snapshots while this program is running:

(1) The main program has called procedure `P`. Register LB points to the topmost frame, which belongs to `P`.

(2) Procedure `P` has called procedure `S`. Register LB points to the topmost frame, which belongs to `S`; register L1 points to a frame belonging to `P`.

(3) Procedure S has called procedure Q. Register LB points to the topmost frame, which belongs to Q; register L1 still points to a frame belonging to P.

(4) Procedure Q has called procedure R. Register LB points to the topmost frame, which belongs to R; register L1 now points to a frame belonging to Q; register L2 points to a frame belonging to P.

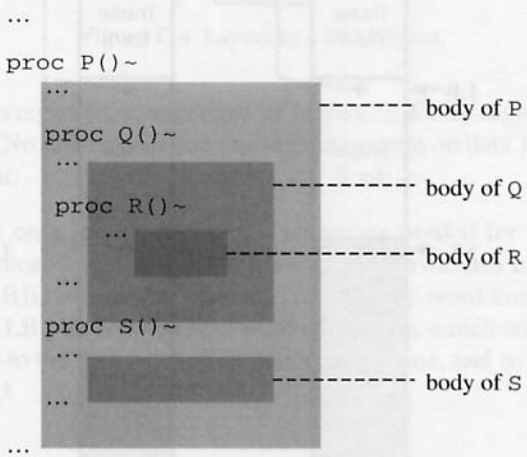



**Figure C.2** Outline of a source program.

Global, local, and nonlocal data can be accessed as follows:

| | |
|---|---|
| LOAD(*n*) *d*[SB] | – for any procedure to load global data |
| LOAD(*n*) *d*[LB] | – for any procedure to load its own local data |
| LOAD(*n*) *d*[L1] | – for procedure Q or S to load data local to P |
| LOAD(*n*) *d*[L1] | – for procedure R to load data local to Q |
| LOAD(*n*) *d*[L2] | – for procedure R to load data local to P |

In each case an *n*-word object is loaded from address *d* relative to the base of the appropriate segment. Storing is analogous to loading.

In general, register LB points to the topmost frame, which is always associated with the routine *R* whose code is currently being executed; register L1 points to a frame associated with the routine *R´* that textually encloses *R* in the source program; register L2 points to a frame associated with the routine *R´´* that textually encloses *R´*; and so on.

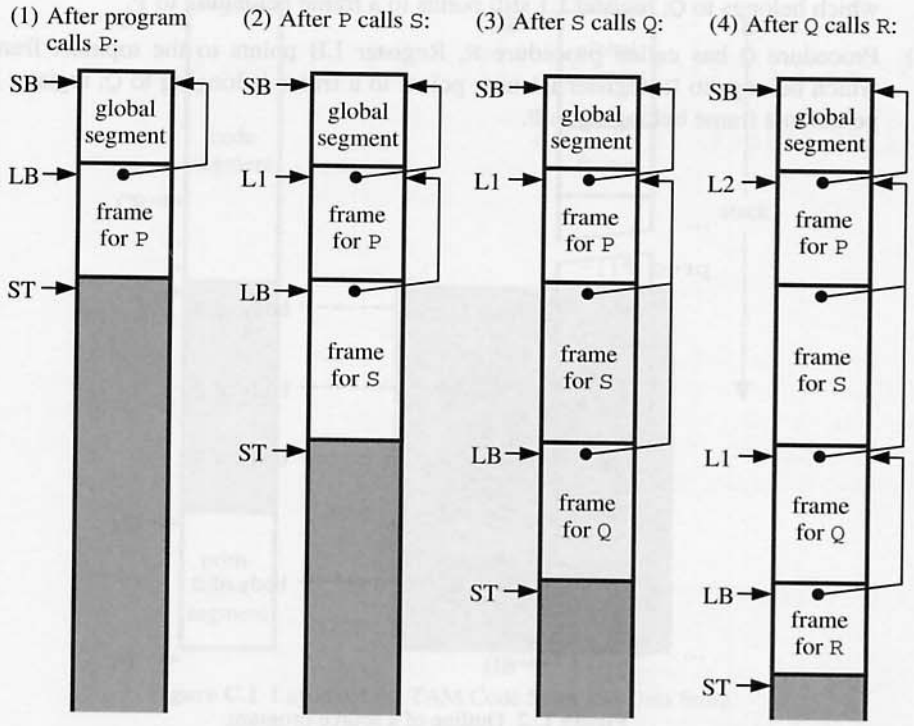



**Figure C.3** Snapshots of the TAM stack (showing static links but not dynamic links).

All accessible data in the stack may be addressed relative to registers SB, LB, L1, L2, etc., as follows:

| | |
|---|---|
| *d*[SB] | – for any routine to access global data |
| *d*[LB] | – for any routine to access its own local data |
| *d*[L1] | – for routine *R* to access data local to *R′* |
| *d*[L2] | – for routine *R* to access data local to *R″* |

In each case an object is accessed at address *d* relative to the base of the appropriate segment. (In practice, registers L1, L2, etc., are used far less often than LB and SB.)

The layout of a frame is illustrated in Figure C.4. Consider a frame associated with routine *R*:

- The *static link* points to an underlying frame associated with the routine that textually encloses *R* in the source program.
- The *dynamic link* points to the frame immediately underlying this one in the stack.
- The *return address* is the address of the instruction immediately following the call instruction that activated *R*.

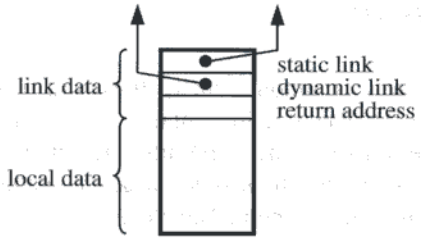



**Figure C.4** Layout of a TAM frame.

There are sixteen registers, summarized in Table C.1. Every register is dedicated to a particular purpose. (No instructions use registers to operate on data.) Some of the registers are constant.

L1, L2, etc., are only pseudo-registers – whenever needed for addressing nonlocal data, they are dynamically evaluated from LB using the invariants L1 = *content*(LB), L2 = *content*(*content*(LB)), etc., where *content*(*a*) means the word contained at address *a*. This works because LB points to the first word of a frame, which contains its static link, which in turn points to the first word of an underlying frame, and so on.

**Table C.1** Summary of TAM registers.

| Register number | Register mnemonic | Register name | Behavior |
|---|---|---|---|
| 0 | CB | Code Base | constant |
| 1 | CT | Code Top | constant |
| 2 | PB | Primitives Base | constant |
| 3 | PT | Primitives Top | constant |
| 4 | SB | Stack Base | constant |
| 5 | ST | Stack Top | changed by most instructions |
| 6 | HB | Heap Base | constant |
| 7 | HT | Heap Top | changed by heap routines |
| 8 | LB | Local Base | changed by call and return instructions |
| 9 | L1 | Local base 1 | L1 = *content*(LB) |
| 10 | L2 | Local base 2 | L2 = *content*(*content*(LB)) |
| 11 | L3 | Local base 3 | L3 = *content*(*content*(*content*(LB))) |
| 12 | L4 | Local base 4 | L4 = *content*(*content*(*content*(*content*(LB)))) |
| 13 | L5 | Local base 5 | L5 = *content*(*content*(*content*(*content*(*content*(LB))))) |
| 14 | L6 | Local base 6 | L6 = *content*(*content*(*content*( *content*(*content*(*content*(LB)))))) |
| 15 | CP | Code Pointer | changed by all instructions |

# C.2 Instructions

All TAM instructions have a common format, illustrated in Figure C.5. The *op* field contains the operation code. The *r* field contains a register number, and the *d* field usually contains an address displacement (possibly negative); together these define the operand's address (*d* + register *r*). The *n* field usually contains the size of the operand. The TAM instruction set is summarized in Table C.2.

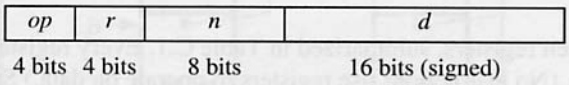



**Figure C.5** TAM instruction format.

# C.3 Routines

Every TAM routine must strictly respect the protocol illustrated in Figure C.6. Assume that routine *R* accepts *d* words of arguments and returns an *n*-word result. Immediately before *R* is called, its arguments must be at the stack top. (If *R* takes no arguments, *d* = 0.) On return from *R*, its arguments must be replaced at the stack top by its result. (If *R* does not return a result, *n* = 0.)

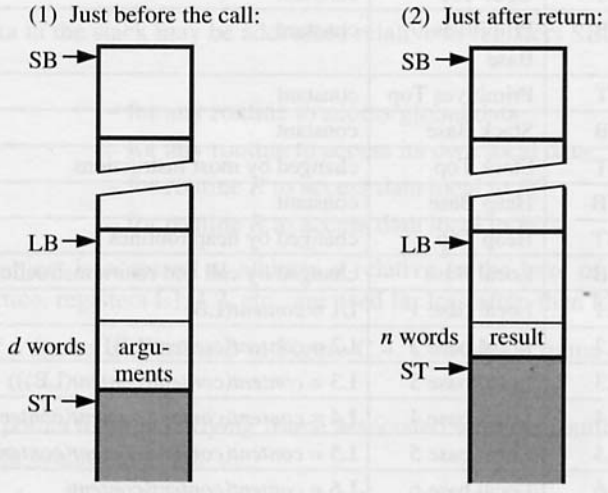



**Figure C.6** Stack layout before and after calling a TAM routine.

There are two kinds of routine in TAM:

- code routines
- primitive routines

A *code routine* consists of a sequence of instructions stored in the code segment. Control is transferred to the first instruction of that sequence by a `CALL` or `CALLI` instruction, and subsequently transferred back by a `RETURN` instruction. Figure C.7 illustrates the layout of the stack during a call to a code routine *R*.

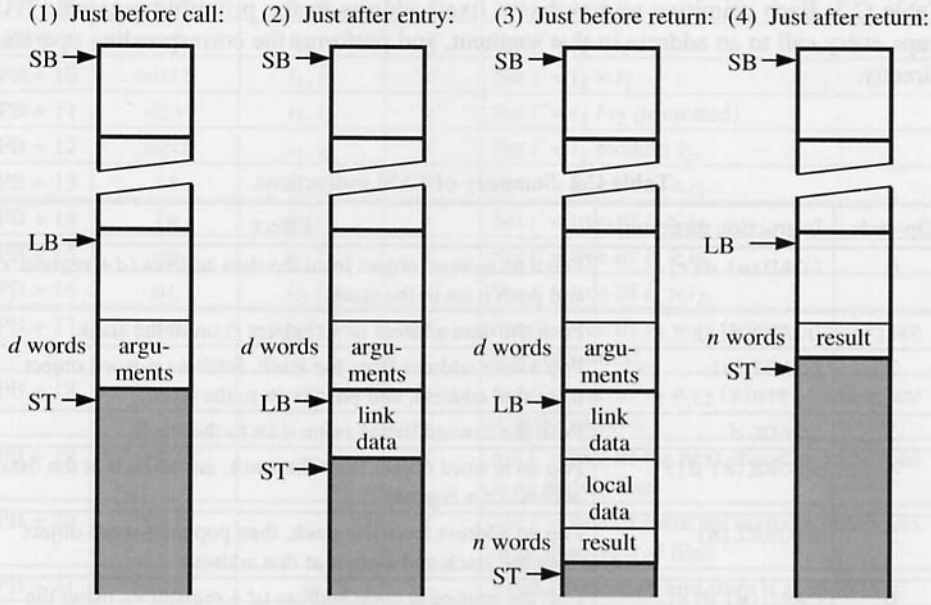



**Figure C.7** Stack layout before, during, and after calling a TAM code routine.

(1) Immediately before the call, *R*'s arguments (if any) must be at the stack top.

(2) The call instruction pushes a new frame on top of the arguments and makes LB point to the base of that frame. The frame's static link is supplied by the call instruction. (The instruction '`CALL(n) ...`' takes the address in register *n* as the static link. The instruction '`CALLI`' takes the static link from the closure at the stack top.) The frame's dynamic link is the address formerly in register LB. The frame's return address is the address of the instruction immediately following the call instruction. At this stage the new frame contains only these three words.

(3) The instructions of *R* may expand the topmost frame, e.g., by allocating space for local data. Immediately before the return, *R* must place any result at the stack top.

(4) The return instruction 'RETURN(*n*) *d*' pops the topmost frame and replaces the *d* words of arguments by the *n*-word result. LB is reset using the dynamic link, and control is transferred to the instruction at the return address.

Since *R*'s arguments lie immediately below its frame, *R* can access the arguments using negative displacements relative to LB. For example:

```
LOAD(1) -d[LB]      – for R to load its first argument (1 word)
LOAD(1) -1[LB]      – for R to load its last argument (1 word)
```

A *primitive routine* is one that performs an elementary arithmetic, logical, input–output, heap, or general-purpose operation. The primitive routines are summarized in Table C.3. Each primitive routine has a fixed address in the primitive segment. TAM traps every call to an address in that segment, and performs the corresponding operation directly.

**Table C.2** Summary of TAM instructions.

| Op-code | Instruction mnemonic | Effect |
|---|---|---|
| 0 | LOAD(*n*) *d*[*r*] | Fetch an *n*-word object from the data address (*d* + register *r*), and push it on to the stack. |
| 1 | LOADA *d*[*r*] | Push the data address (*d* + register *r*) on to the stack. |
| 2 | LOADI(*n*) | Pop a data address from the stack, fetch an *n*-word object from that address, and push it on to the stack. |
| 3 | LOADL *d* | Push the 1-word literal value *d* on to the stack. |
| 4 | STORE(*n*) *d*[*r*] | Pop an *n*-word object from the stack, and store it at the data address (*d* + register *r*). |
| 5 | STOREI(*n*) | Pop an address from the stack, then pop an *n*-word object from the stack and store it at that address. |
| 6 | CALL(*n*) *d*[*r*] | Call the routine at code address (*d* + register *r*), using the address in register *n* as the static link. |
| 7 | CALLI | Pop a closure (static link and code address) from the stack, then call the routine at that code address. |
| 8 | RETURN(*n*) *d* | Return from the current routine: pop an *n*-word result from the stack, then pop the topmost frame, then pop *d* words of arguments, then push the result back on to the stack. |
| 9 | – | (unused) |
| 10 | PUSH *d* | Push *d* words (uninitialized) on to the stack. |
| 11 | POP(*n*) *d* | Pop an *n*-word result from the stack, then pop *d* more words, then push the result back on to the stack. |
| 12 | JUMP *d*[*r*] | Jump to code address (*d* + register *r*). |
| 13 | JUMPI | Pop a code address from the stack, then jump to that address. |
| 14 | JUMPIF(*n*) *d*[*r*] | Pop a 1-word value from the stack, then jump to code address (*d* + register *r*) if and only if that value equals *n*. |
| 15 | HALT | Stop execution of the program. |

**Table C.3** Summary of TAM primitive routines.

| Address | Mnemonic | Arguments | Result | Effect |
|---|---|---|---|---|
| PB + 1 | `id` | $w$ | $w'$ | Set $w' = w$. |
| PB + 2 | `not` | $t$ | $t'$ | Set $t' = \neg t$. |
| PB + 3 | `and` | $t_1, t_2$ | $t'$ | Set $t' = t_1 \wedge t_2$. |
| PB + 4 | `or` | $t_1, t_2$ | $t'$ | Set $t' = t_1 \vee t_2$. |
| PB + 5 | `succ` | $i$ | $i'$ | Set $i' = i + 1$. |
| PB + 6 | `pred` | $i$ | $i'$ | Set $i' = i - 1$. |
| PB + 7 | `neg` | $i$ | $i'$ | Set $i' = -i$. |
| PB + 8 | `add` | $i_1, i_2$ | $i'$ | Set $i' = i_1 + i_2$. |
| PB + 9 | `sub` | $i_1, i_2$ | $i'$ | Set $i' = i_1 - i_2$. |
| PB + 10 | `mult` | $i_1, i_2$ | $i'$ | Set $i' = i_1 \times i_2$. |
| PB + 11 | `div` | $i_1, i_2$ | $i'$ | Set $i' = i_1 / i_2$ (truncated). |
| PB + 12 | `mod` | $i_1, i_2$ | $i'$ | Set $i'$ = $i_1$ modulo $i_2$. |
| PB + 13 | `lt` | $i_1, i_2$ | $t'$ | Set $t'$ = true iff $i_1 < i_2$. |
| PB + 14 | `le` | $i_1, i_2$ | $t'$ | Set $t'$ = true iff $i_1 \leq i_2$. |
| PB + 15 | `ge` | $i_1, i_2$ | $t'$ | Set $t'$ = true iff $i_1 \geq i_2$. |
| PB + 16 | `gt` | $i_1, i_2$ | $t'$ | Set $t'$ = true iff $i_1 > i_2$. |
| PB + 17 | `eq` | $v_1, v_2, n$ | $t'$ | Set $t'$ = true iff $v_1 = v_2$ (where $v_1$ and $v_2$ are $n$-word values). |
| PB + 18 | `ne` | $v_1, v_2, n$ | $t'$ | Set $t'$ = true iff $v_1 \neq v_2$ (where $v_1$ and $v_2$ are $n$-word values). |
| PB + 19 | `eol` | – | $t'$ | Set $t'$ = true iff the next character to be read is an end-of-line. |
| PB + 20 | `eof` | – | $t'$ | Set $t'$ = true iff there are no more characters to be read (end of file). |
| PB + 21 | `get` | $a$ | – | Read a character, and store it at address $a$. |
| PB + 22 | `put` | $c$ | – | Write the character $c$. |
| PB + 23 | `geteol` | – | – | Read characters up to and including the next end-of-line. |
| PB + 24 | `puteol` | – | – | Write an end-of-line. |
| PB + 25 | `getint` | $a$ | – | Read an integer-literal (optionally preceded by blanks and/or signed), and store its value at address $a$. |
| PB + 26 | `putint` | $i$ | – | Write an integer-literal whose value is $i$. |
| PB + 27 | `new` | $n$ | $a'$ | Set $a'$ = address of a newly allocated $n$-word object in the heap. |
| PB + 28 | `dispose` | $n, a$ | – | Deallocate the $n$-word object at address $a$ in the heap. |

(See notes overleaf.)

Notes for Table C.3:

- *a* denotes a data address
- *c* denotes a character
- *i* denotes an integer
- *n* denotes a non-negative integer
- *t* denotes a truth value (0 for *false* or 1 for *true*)
- *v* denotes a value of any type
- *w* denotes any 1-word value

APPENDIX D

# Class Diagrams for the Triangle Compiler

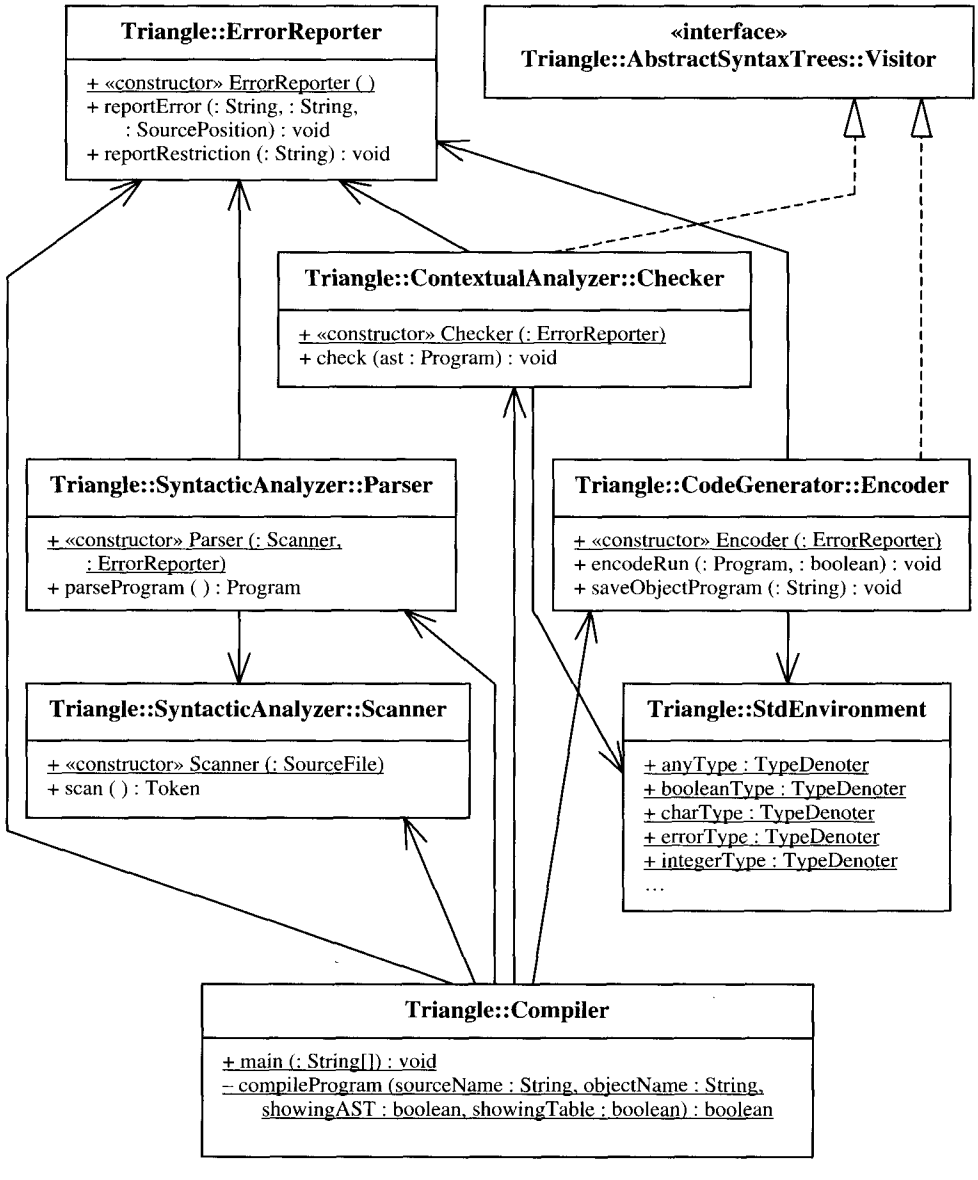
This appendix uses class diagrams to summarize the structure of the Triangle compiler, which is available from our Web site (see Preface, page xv).

The Triangle compiler has broadly the same structure as the Mini-Triangle compiler used throughout the text of this book. It is discussed in more detail in Sections 3.3, 4.6, 5.4, and 7.5.

The class diagrams are expressed in UML (Unified Modeling Language). UML is described in detail in Booch *et al.* (1999). However, the following points are worth noting. The name of an abstract class is shown in *italics*, whereas the name of a concrete class is shown in **bold**. Private attributes and methods are prefixed by a minus sign (–), whereas public attributes and methods are prefixed by a plus sign (+). The definition of a class attribute or method is underlined. The name of a method parameter is omitted where it is of little significance.

# D.1 Compiler

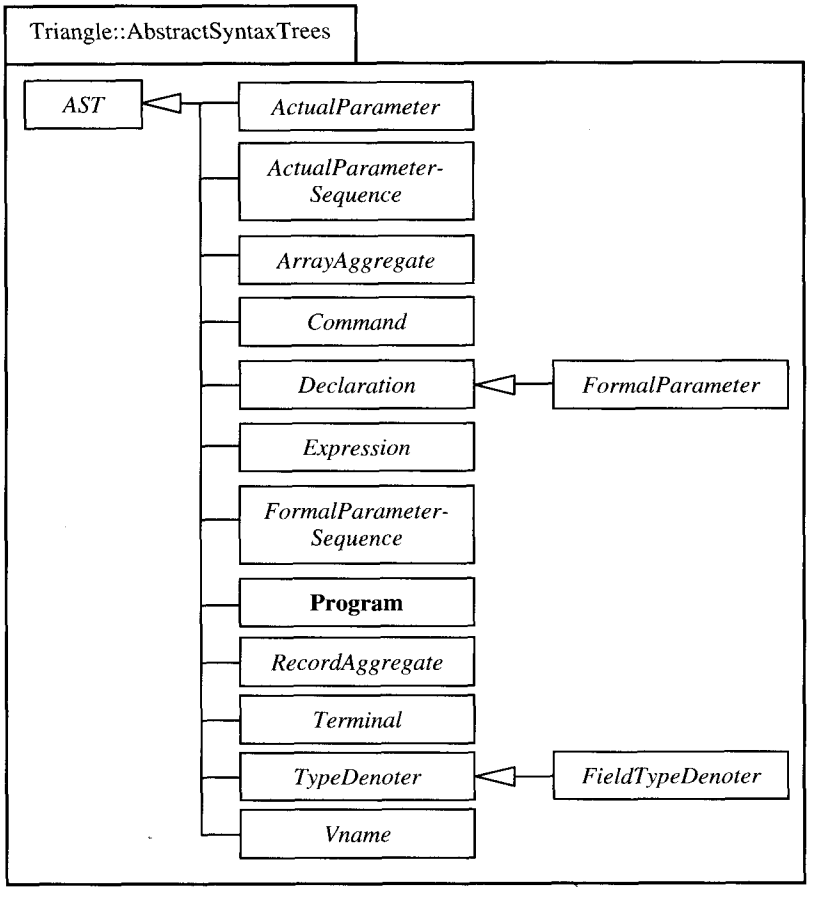
The following diagram shows the overall structure of the compiler, including the syntactic analyzer (scanner and parser), the contextual analyzer, and the code generator:



# D.2 Abstract syntax trees

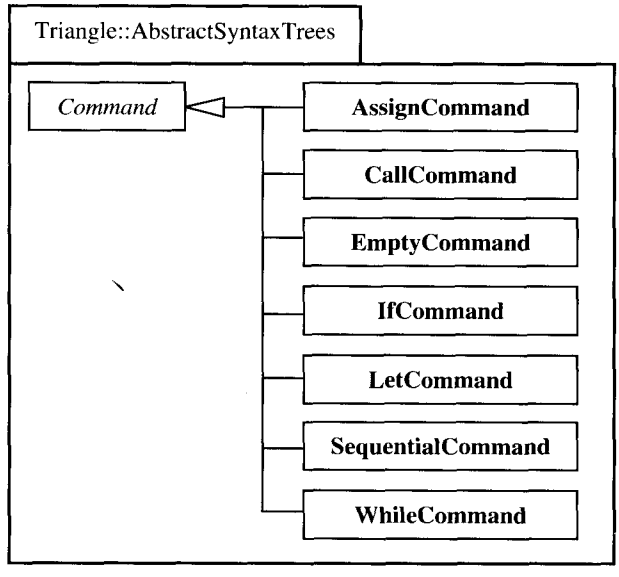
The diagrams in this section show the class hierarchy used in the representation of Triangle ASTs. Each major syntactic class is presented in a separate diagram. These diagrams show class names only, omitting constructors and methods.

The following diagram shows the immediate subclasses of the AST class. Most of these are abstract classes representing the main syntactic phrases. Note that *Formal-Parameter* is a subclass of *Declaration* in order that formal parameters may be included in the identification table during contextual analysis.



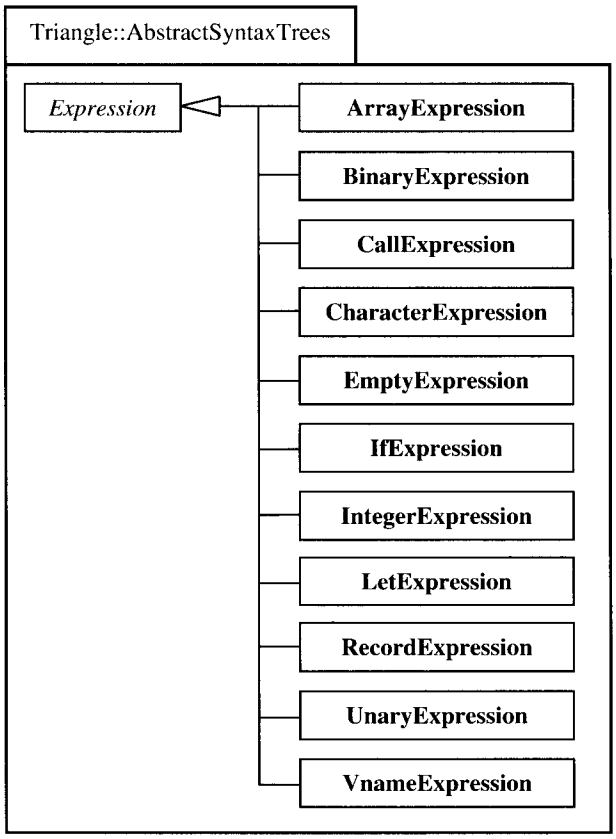
## D.2.1 Commands

The following diagram shows the individual concrete classes for each form of command:

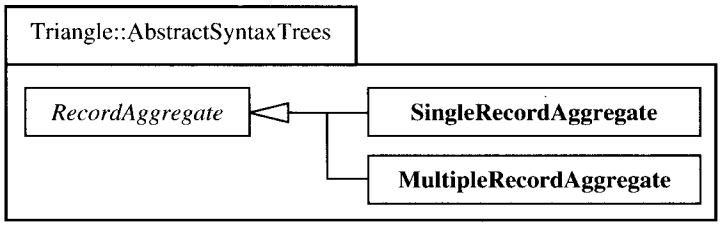


## D.2.2 Expressions

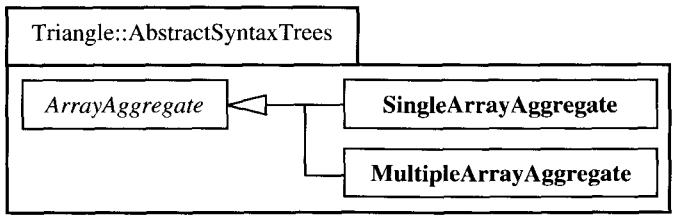
The following diagram shows the individual concrete classes for each form of expression:




The following diagram shows the individual concrete subclasses for a record aggregate:

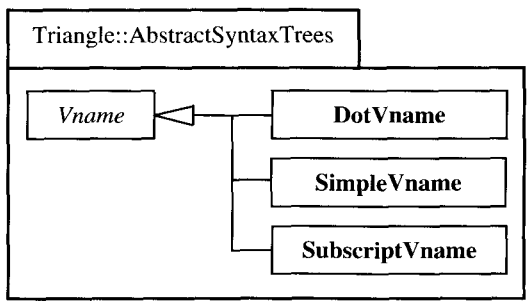


The following diagram shows the individual concrete subclasses for an array aggregate:



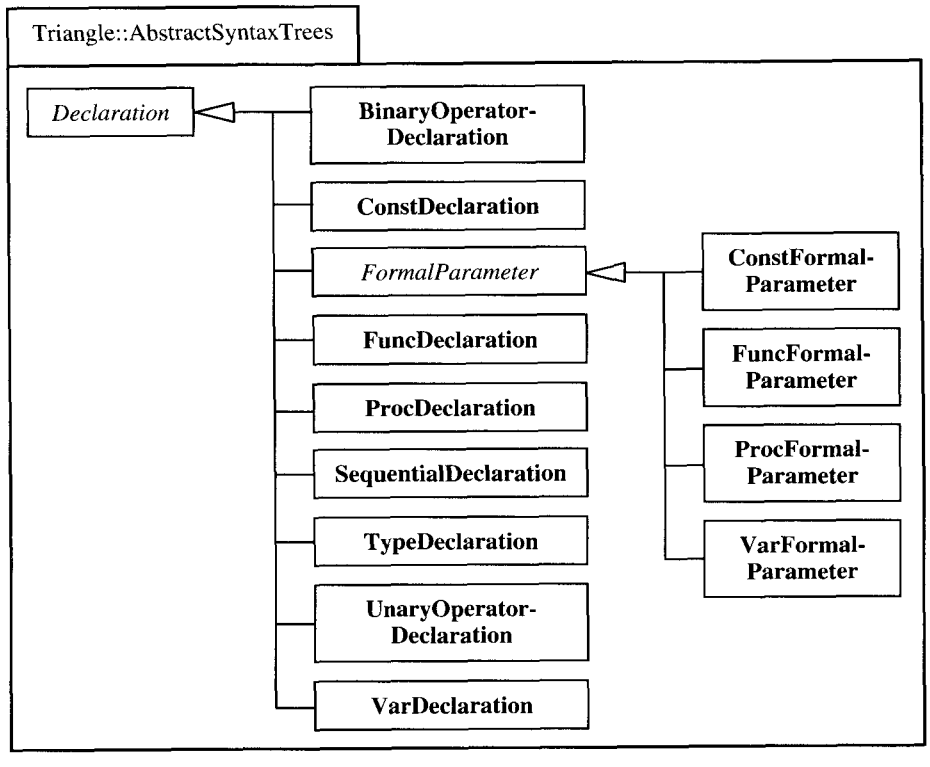

## D.2.3 Value-or-variable names

The following diagram shows the individual concrete subclasses for each form of value-or-variable name:



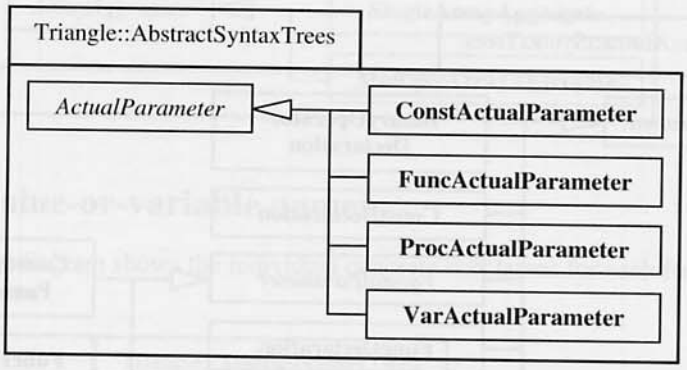
## D.2.4 Declarations

The following diagram shows the individual concrete classes for each form of declaration:

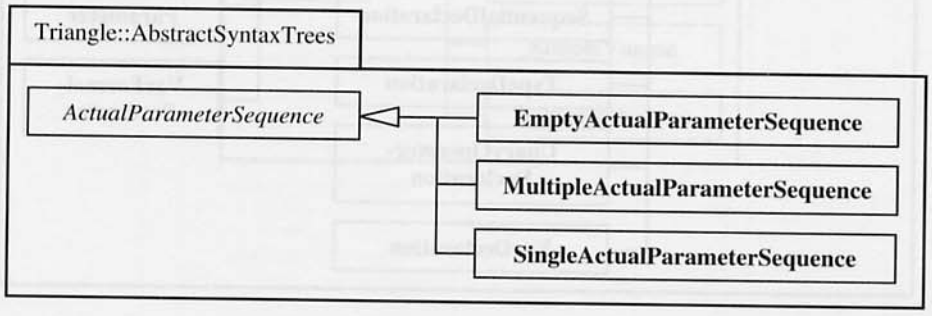


## D.2.5 Parameters

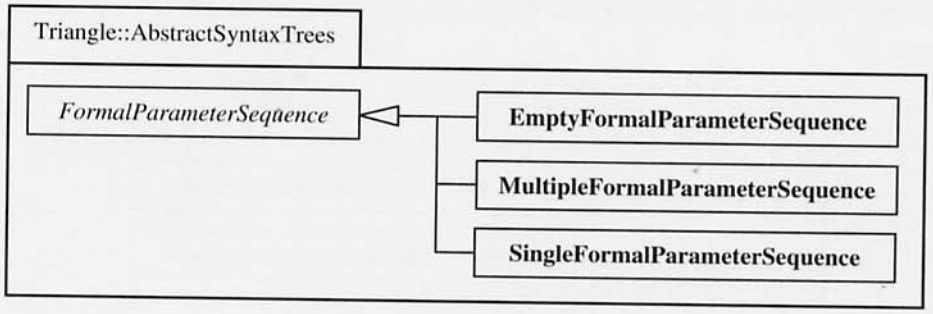
The following diagram shows the individual concrete subclasses for each form of actual parameter:




The following diagram shows the individual concrete subclasses for each form of actual parameter sequence:

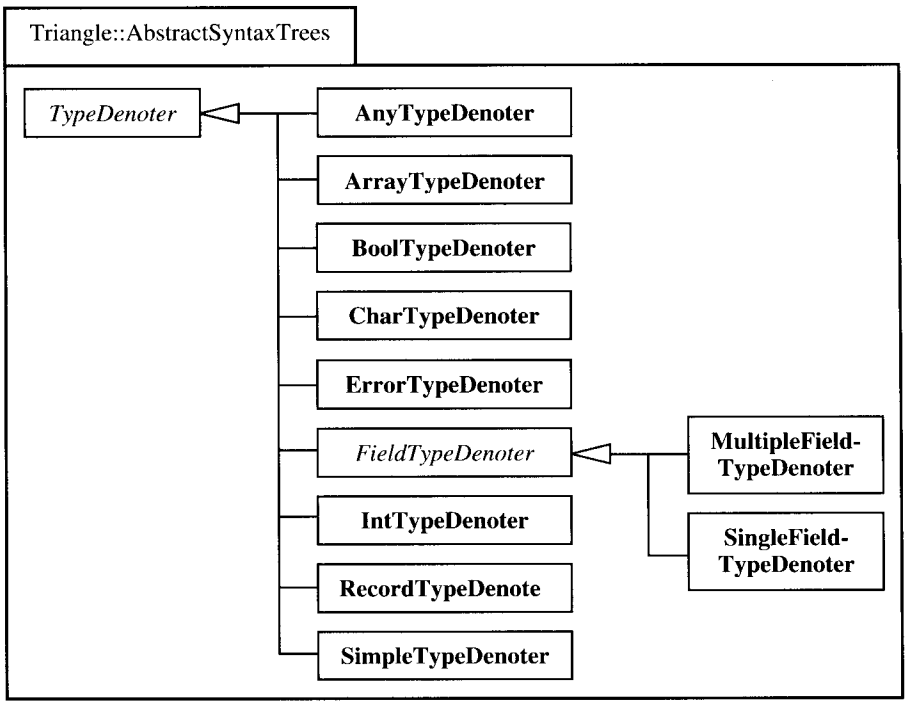



The following diagram shows the individual concrete subclasses for each form of formal parameter sequence:



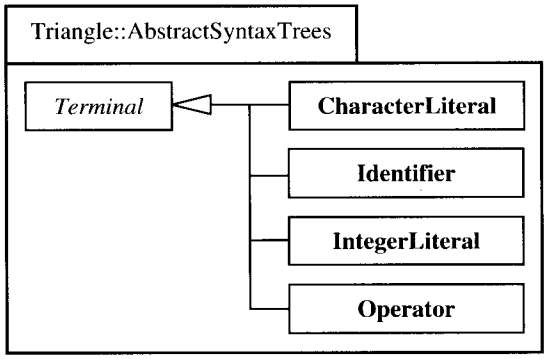
## D.2.6 Type-denoters

The following diagram shows the individual concrete subclasses for each form of type-denoter:



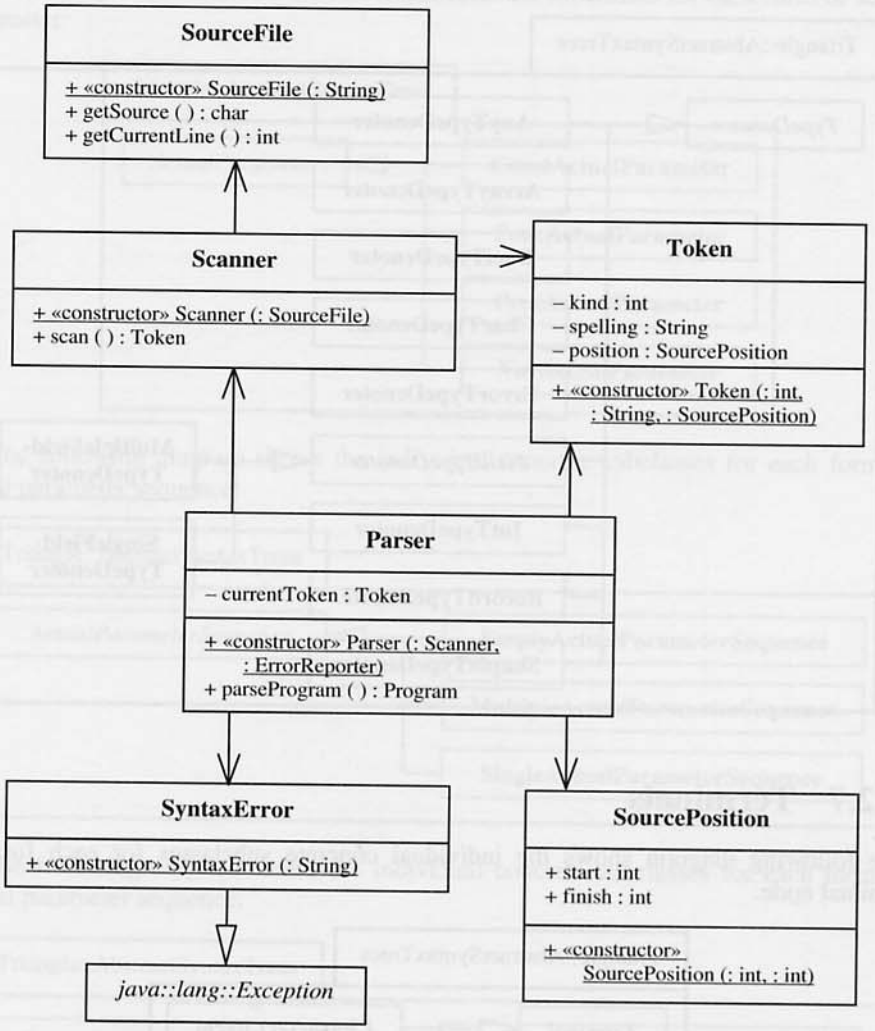

## D.2.7 Terminals

The following diagram shows the individual concrete subclasses for each form of terminal node:



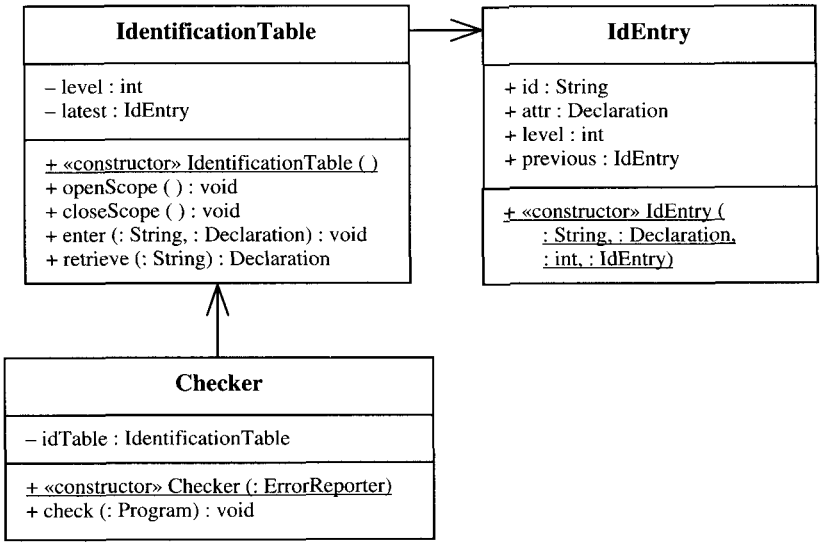
# D.3 Syntactic analyzer

The following diagram shows the internal structure of the syntactic analyzer.



# D.4 Contextual analyzer

The following diagram shows the internal structure of the contextual analyzer.



# D.5 Code generator

The following diagram shows the internal structure of the code generator.