

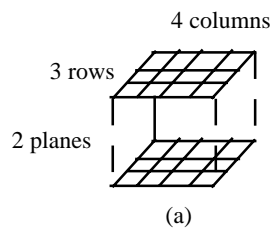
### 5.4 Arrays

Although arrays are not included in our definition of Decaf, they are of such great importance to programming languages and computing in general, that we would be remiss not to mention them at all in a compiler text. We will give a brief description of how multi-dimensional array references can be implemented and converted to atoms, but for a more complete and efficient implementation the student is referred to Parsons [1992] or Aho [1986].

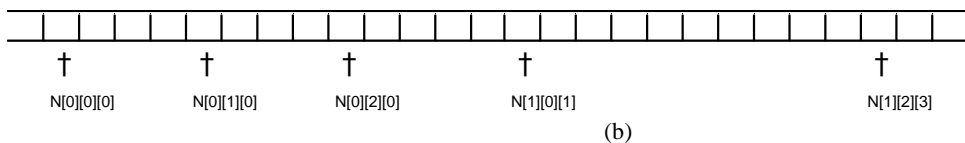
The main problem that we need to solve when referencing an array element is that we need to compute an offset from the first element of the array. Though the programmer may be thinking of multi-dimensional arrays (actually arrays of arrays) as existing in two, three, or more dimensions, they must be physically mapped to the computer's memory, which has one dimension. For example, an array declared as `int n[ ][ ] = new int [2][3][4];` might be envisioned by the programmer as a structure having three rows and four columns in each of two planes as shown in Figure 5.10 (a). In reality, this array is mapped into a sequence of twenty-four ( $2*3*4$ ) contiguous memory locations as shown in Figure 5.10 (b). The problem which the compiler must solve is to convert an array reference such as `n[1][1][0]` to an offset from the beginning of the storage area allocated for `n`. For this example, the offset would be sixteen memory cells (assuming that each element of the array occupies one memory cell).

To see how this is done, we will begin with a simple one-dimensional array and then proceed to two and three dimensions. For a vector, or one-dimensional array, the offset is simply the subscripting value, since subscripts begin at 0 in Java. For example, if `v` were declared to contain twenty elements, `char v[] = new char [20];`, then the offset for the fifth element, `v[4]`, would be 4, and in general the offset for a reference `v[i]` would be `i`. The simplicity of this formula results from the fact that array indexing begins with 0 rather than 1. A vector maps directly to the computer's memory.

Now we introduce arrays of arrays, which, for the purposes of this discussion, we call multi-dimensional arrays; suppose `m` is declared as a matrix, or two-dimensional



**Figure 5.10** A Three-Dimensional Array `N[2][3][4]` (a) Mapped into a One-Dimensional Memory (b).



array, `char m[] [] = new char [10] [15];`. We are thinking of this as an array of 10 rows, with 15 elements in each row. A reference to an element of this array will compute an offset of fifteen elements for each row after the first. Also, we must add to this offset the number of elements in the selected row. For example, a reference to `m[4][7]` would require an offset of  $4*15 + 7 = 67$ . The reference `m[r][c]` would require an offset of  $r*15 + c$ . In general, for a matrix declared as `char m[] [] = new char [ROWS] [COLS]`, the formula for the offset of `m[r][c]` is  $r*COLS + c$ .

For a three-dimensional array, `char a[] [] [] = new char [5] [6] [7];`, we must sum an offset for each plane ( $6*7$  elements), an offset for each row (7 elements), and an offset for the elements in the selected row. For example, the offset for the reference `a[2][3][4]` is found by the formula  $2*6*7 + 3*7 + 4$ . The reference `a[p][r][c]` would result in an offset computed by the formula  $p*6*7 + r*7 + c$ . In general, for a three-dimensional array, `new char [PLANES] [ROWS] [COLS]`, the reference `a[p][r][c]` would require an offset computed by the formula  $p*ROWS*COLS + r*COLS + c$ .

We now generalize what we have done to an array that has any number of dimensions. Each subscript is multiplied by the total number of elements in all higher dimensions. If an  $n$  dimensional array is declared as `char a[] [] ... [] = new char [D1] [D2] [D3] ... [Dn]`, then a reference to `a[S1][S2][S3] ... [Sn]` will require an offset computed by the following formula:

$$S_1 * D_2 * D_3 * D_4 * \dots * D_n + S_2 * D_3 * D_4 * \dots * D_n + S_3 * D_4 * \dots * D_n + \dots + S_{n-1} * D_n + S_n.$$

In this formula,  $D_i$  represents the number of elements in the  $i$ th dimension and  $S_i$  represents the  $i$ th subscript in a reference to the array. Note that in some languages, such as Java and C, all the subscripts are not required. For example, the array of three dimensions `a[2][3][4]`, may be referenced with two, one, or even zero subscripts. `a[1]` refers to the address of the first element in the second plane; i.e. all missing subscripts are assumed to be zero.

Notice that some parts of the formula shown above can be computed at compile time. For example, for arrays which are dimensioned with constants, the product of dimensions  $D_2 * D_3 * D_4$  can be computed at compile time. However, since subscripts can be arbitrary expressions, the complete offset may have to be computed at run time.

The atoms which result from an array reference must compute the offset as described above. Specifically, for each dimension,  $i$ , we will need a MUL atom to multiply  $S_i$  by the product of dimensions from  $D_{i+1}$  through  $D_n$ , and we will need an ADD atom to add the term for this dimension to the sum of the previous terms. Before showing a translation grammar for this purpose, however, we will first show a grammar without action symbols or attributes, which defines array references. Grammar G22 is an extension to the grammar for simple arithmetic expressions, G5, given in Section 3.1. Here we have changed rule 7 and added rules 8,9.

G22

1.  $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
2.  $\text{Expr} \rightarrow \text{Term}$
3.  $\text{Term} \rightarrow \text{Term} * \text{Factor}$
4.  $\text{Term} \rightarrow \text{Factor}$
5.  $\text{Factor} \rightarrow (\text{Expr})$
6.  $\text{Factor} \rightarrow \text{const}$
7.  $\text{Factor} \rightarrow \text{var Subs}$
8.  $\text{Subs} \rightarrow [\text{Expr}] \text{Subs}$
9.  $\text{Subs} \rightarrow \epsilon$

This extension merely states that a variable may be followed by a list of subscripting expressions, each in square brackets (the nonterminal Subs represents a list of subscripts).

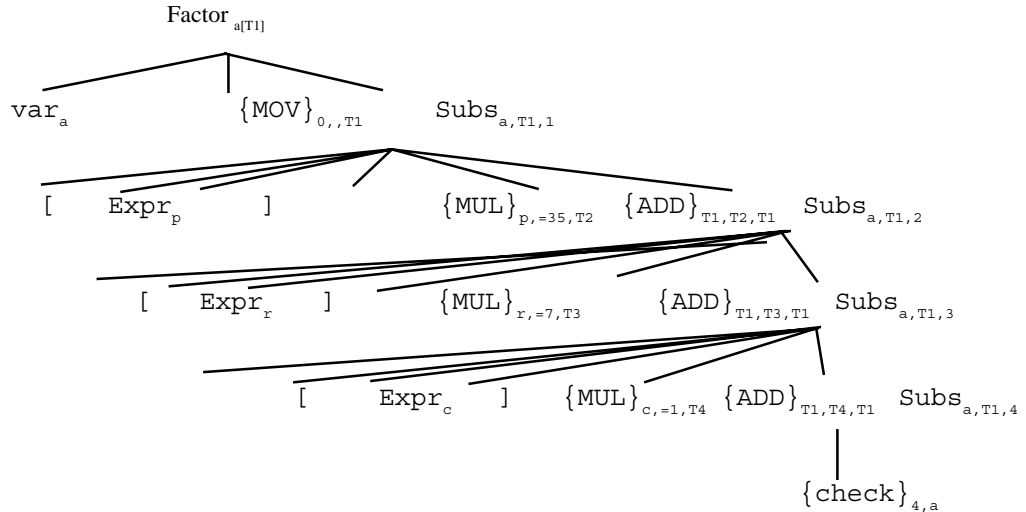
Grammar G23 shows rules 7-9 of grammar G22, with attributes and action symbols. Our goal is to come up with a correct offset for a subscripted variable in grammar rule 8, and provide its address for the attribute of the Subs defined in that rule.

Grammar G23:

7.  $\text{Factor}_e \rightarrow \text{var}_v \{ \text{MOV} \}_{0, \text{sum}} \text{Subs}_{v, \text{sum}, i}$   
 $e \leftarrow v[\text{sum}]$   
 $i \leftarrow 1$   
 $\text{sum} \leftarrow \text{Alloc}$
8.  $\text{Subs}_{v, \text{sum}, i1} \rightarrow [\text{Expr}_e] \{ \text{MUL} \}_{e, =D, T} \{ \text{ADD} \}_{\text{sum}, T, \text{sum}} \text{Subs}_{v, \text{sum}, i2}$   
 $D \leftarrow \text{prod}(v, i1)$   
 $i2 \leftarrow i1 + 1$   
 $T \leftarrow \text{Alloc}$
9.  $\text{Subs}_{v, \text{sum}, i} \rightarrow \{ \text{check} \}_{i, v}$

The nonterminal Subs has three attributes:  $v$  (inherited) represents a reference to the symbol table for the array being referenced,  $\text{sum}$  (synthesized) represents the location storing the sum of the terms which compute the offset, and  $i$  (inherited) is the dimension being processed. In the attribute computation rules for grammar rule 8, there is a call to a method  $\text{prod}(v, i)$ . This method computes the product of the dimensions of the array  $v$ , above dimension  $i$ . As noted above, this product can be computed at compile time. Its value is then stored as a constant,  $D$ , and referred to in the grammar as  $=D$ .

The first attribute rule for grammar rule 7 specifies  $e \leftarrow v[\text{sum}]$ . This means that the value of  $\text{sum}$  is used as an offset to the address of the variable  $v$ , which then



**Figure 5.11** A Derivation Tree for the Array Reference  $A[p][r][c]$ , Which is Declared as `int A[3][5][7]`.

becomes the attribute of the `Factor` defined in rule 7.

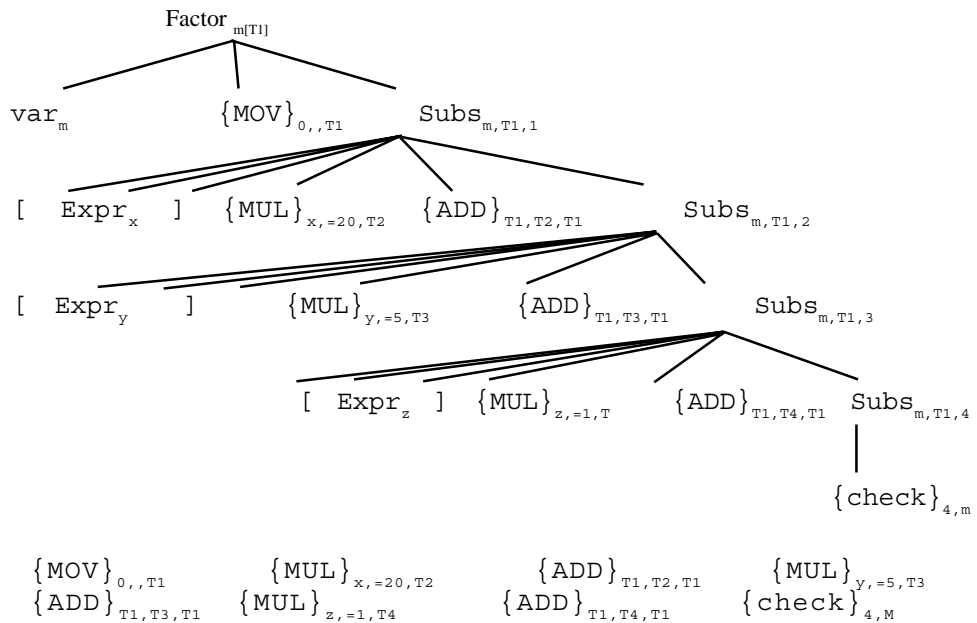
The compiler should ensure that the number of subscripts in the array reference is equal to the number of subscripts in the array declaration. If they are not equal, an error message should be put out. This is done by a procedure named `check(i, v)` which is specified by the action symbol  $\{check\}_{i,v}$  in rule 9. This action symbol represents a procedure call, not an atom. The purpose of the procedure is to compare the number of dimensions of the variable,  $v$ , as stored in the symbol table, with the value of  $i$ , the number of subscripts plus one. The `check(i, v)` method simply puts out an error message if the number of subscripts does not equal the number of dimensions, and the parse continues.

To see how this translation grammar works, we take an example of a three-dimensional array declared as `int a[][][] = new int[3][5][7]`. An attributed derivation tree for the reference `a[p][r][c]` is shown, above, in Figure 5.11 (for simplicity we show only the part of the tree involving the subscripted variable, not an entire expression). To build this derivation tree, we first build the tree without attributes and then fill in attribute values where possible. Note that the first and third attributes of `Subs` are inherited and derive values from higher nodes or nodes on the same level in the tree. The final result is the offset stored in the attribute `sum`, which is added to the attribute of the variable being subscripted to obtain the offset address. This is then the attribute of the `Factor` which is passed up the tree.

**Sample Problem 5.4**

Assume the array *m* has been declared to have two planes, four rows, and five columns (`int m[2][4][5]`). Show the attributed derivation tree generated by grammar G23 for the array reference `m[x][y][z]`. Use `Factor` as the starting nonterminal, and show the subscripting expressions as `Expr`, as done in Figure 5.11. Also show the sequence of atoms which would be put out as a result of this array reference.

**Solution:**



**Exercises 5.4**

1. Assume the following array declarations:

```
int v[] = new int [13];
int m[] [] = new int [12] [17];
int a3[] [] [] = new int [15] [7] [5];
```

```
int z [] [] [] [] = new int [4] [7] [2] [3];
```

Show the *attributed derivation tree* resulting from grammar G23 for each of the following array references. Use `Factor` as the starting nonterminal, and show each subscript expression as `Expr`, as done in Figure 5.11. Also show the sequence of atoms that would be put out.

- (a) `v[7]`                      (b) `m[q][2]`                      (c) `a3[11][b][4]`  
 (d) `z[2][c][d][2]`                      (e) `m[1][1]`

2. The discussion in this section assumed that each array element occupied one memory cell. If each array element occupies `SIZE` memory cells, what changes would have to be made to the general *formula* given in this section for the *offset*? How would this affect grammar G23?
3. You are given two vectors: the first, `d`, contains the dimensions of a declared array, and the second, `s`, contains the subscripting values in a reference to that array.

- (a) Write a *Java method* –

```
int offset (int d [], int s []);
```

that computes the offset for an array reference `a [S0] [S1] . . . [Smax-1]` where the array has been declared as `char a [d0] [d1] . . . [dmax-1]`.

- (b) *Improve* your Java method, if possible, to minimize the number of run-time multiplications.

### 5.5 Case Study: Syntax Analysis for Decaf

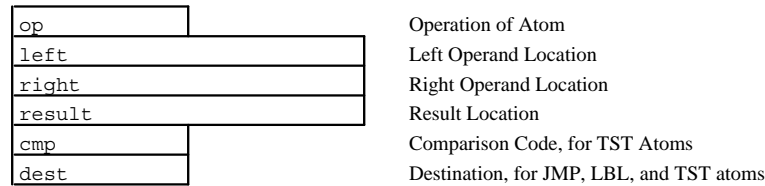
In this section we continue the development of a compiler for Decaf, a small subset of the Java programming language. We do this by implementing the syntax analysis phase of the compiler using SableCC as described in Section 5.3, above. The parser generated by SableCC will obtain input tokens from the standard input stream. The parser will then check the tokens for correct syntax.

In addition, we provide a Translation class which enables our parser to put out atoms corresponding to the run-time operations to be performed. This aspect of compilation is often called *semantic analysis*. For more complex languages, semantic analysis would also involve type checking, type conversions, identifier scopes, array references, and symbol table management. Since these will not be necessary for the Decaf compiler, syntax analysis and semantic analysis have been combined into one program.

The complete SableCC grammar file and Translation source code is shown in Appendix B and is explained here. The input to SableCC is the file `decaf.grammar`, which generates classes for the parser, nodes, lexer, and analysis. In the Tokens section, we define the two types of comments; `comment1` is a single-line comment, beginning with `//` and ending with a newline character. `comment2` is a multi-line comment, beginning with `/*` and ending with `*/`. Neither of these tokens requires the use of states, which is why there is no States section in our grammar. Next each keyword is defined as a separate token taking care to include these before the definition of identifiers. These are followed by special characters `'+' , '-' , ';' , . . .`. Note that relational operators are defined collectively as a `compare` token. Finally we define identifiers and numeric constants as tokens. The Ignored Tokens are `space` and the two comment tokens.

The Productions section is really the Decaf grammar with some modifications to allow for bottom-up parsing. The major departure from what has been given previously and in Appendix A, is the definition of the `if` statement. We need to be sure to handle the *dangling else* appropriately; this is the ambiguity problem discussed in section 3.1 caused by the fact that an `if` statement has an optional `else` part. This problem was relatively easy to solve when parsing top-down, because the ambiguity was always resolved in the correct way simply by checking for an `else` token in the input stream. When parsing bottom-up, however, we get a shift-reduce conflict from this construct. If we rewrite the grammar to eliminate the ambiguity, as in section 3.1 (Grammar G7), we still get a shift-reduce conflict. Unfortunately, in SableCC there is no way to resolve this conflict always in favor of a shift (this is possible with `yacc`). Therefore, we will need to rewrite the grammar once again; we use a grammar adapted from Appel [2002]. In this grammar a `no_short_if` statement is one which does not contain an `if` statement without a matching `else`. The EBNF capabilities of SableCC are used, for example, in the definition of `compound_stmt`, which consists of a pair of braces enclosing 0 or more statements. The complete grammar is shown in appendix B. An array of Doubles named 'memory' is used to store the values of numeric constants.

The Translation class, also shown in appendix B, is written to produce atoms for the arithmetic operations and control structures. The structure of an atom is shown in



**Figure 5.12** Record Structure of the File of Atoms

Figure 5.12. The Translation class uses a few Java maps: the first map, implemented as a `HashMap` and called 'hash', stores the temporary memory location associated with each sub-expression (i.e. with each node in the syntax tree). It also stores label numbers for the implementation of control structures. Hence, the keys for this map are nodes, and the values are the integer run-time memory locations, or label numbers, associated with them. The second map, called 'nums', stores the values of numeric constants, hence if a number occurs several times in a Decaf program, it need be stored only once in this map. The third map is called 'identifiers'. This is our Decaf symbol table. Each identifier is stored once, when it is declared. The Translation class checks that an identifier is not declared more than once (local scope is not permitted), and it checks that an identifier has been declared before it is used. For both numbers and identifiers, the value part of each entry stores the run-time memory location associated with it. The implementation of control structures for `if`, `while`, and `for` statements follows that which was presented in section 4.9. A boolean expression always results in a TST atom which branches if the comparison operation result is false. Whenever a new temporary location is needed, the method `alloc` provides the next available location (a better compiler would re-use previously allocated locations when possible). Whenever a new label number is needed, it is provided by the `lalloc` method. Note that when an integer value is stored in a map, it must be an object, not a primitive. Therefore, we use the wrapper class for integers provided by Java, `Integer`. The complete Translation class is shown in appendix B and is available at <http://www.rowan.edu/~bergmann/books>.

### Exercises 5.5

1. Extend the Decaf language to include a `do` statement defined as:

```
DoStmt → do Stmt while ( BoolExpr ) ;
```

Modify the files `decaf.grammar` and `Translation.java`, shown in Appendix B so that the compiler puts out the correct *atom* sequence implementing this control structure, in which the test for termination is made after the body of the loop is executed. The nonterminals `Stmt` and `BoolExpr` are already defined. For purposes of

this assignment you may alter the `atom` method so that it prints out its arguments to `stdout` rather than building a file of atoms.

2. Extend the Decaf language to include a `switch` statement defined as:

```
SwitchStmt → switch ( Expr ) CaseList
CaseList → case number ':' Stmt CaseList
CaseList → case number ':' Stmt
```

Modify the files `decaf.grammar` and `Translation.java`, shown in Appendix B, so that the compiler puts out the correct *atom* sequence implementing this control structure. The nonterminals `Expr` and `Stmt` are already defined, as are the tokens `number` and `end`. The token `switch` needs to be defined. Also define a `break` statement which will be used to transfer control out of the `switch` statement. For purposes of this assignment, you may alter the `atom()` function so that it prints out its arguments to `stdout` rather than building a file of atoms, and remove the call to the code generator.

3. Extend the Decaf language to include initializations in declarations, such as:

```
int x=3, y, z=0;
```

Modify the files `decaf.grammar` and `Translation.java`, shown in Appendix B, so that the compiler puts out the correct *atom* sequence implementing this feature. You will need to put out a `MOV` atom to assign the value of the constant to the variable.

## 5.6 Chapter Summary

This chapter describes some *bottom up parsing algorithms*. These algorithms recognize a sequence of grammar rules in a derivation, corresponding to an *upward direction* in the *derivation tree*. In general, these algorithms begin with an empty stack, read input symbols, and apply grammar rules, until left with the starting nonterminal alone on the stack when all input symbols have been read.

The most general class of bottom up parsing algorithms is called *shift reduce parsing*. These parsers have two basic operations: (1) a *shift operation* pushes the current input symbol onto the stack, and (2) a *reduce operation* replaces zero or more top-most stack symbols with a single stack symbol. A reduction can be done only if a *handle* can be identified on the stack. A *handle* is a string of symbols occurring on the right side of a grammar rule, and matching the symbols on top of the stack, as shown below:

$\nabla \dots \text{HANDLE}$	$N\tau \rightarrow \text{HANDLE}$
------------------------------	-----------------------------------

The reduce operation applies the *rewriting rule in reverse*, by replacing the handle on the stack with the nonterminal defined in the corresponding rule, as shown below

$\nabla \dots N\tau$
----------------------

When writing the grammar for a shift reduce parser, one must take care to avoid *shift/reduce conflicts* (in which it is possible to do a reduce operation when a shift is needed for a correct parse) and *reduce/reduce conflicts* (in which more than one grammar rule matches a handle).

A special case of shift reduce parsing, called *LR parsing*, is implemented with a pair of tables: an *action table* and a *goto table*. The *action table* specifies whether a shift or reduce operation is to be applied. The *goto table* specifies the stack symbol to be pushed when the operation is a reduce.

We studied a parser generator, *SableCC*, which generates an LR parser from a specification grammar. It is also possible to include *actions* in the grammar which are to be applied as the input is parsed. These actions are implemented in a Translation class designed to be used with SableCC.

Finally we looked at an *implementation of Decaf*, our case study language which is a subset of Java, using SableCC. This compiler works with the lexical phase discussed in Section 2.4 and is shown in Appendix B.