

5.4 Arrays

Although arrays are not included in our definition of MiniC, 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 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[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 C++. For example, if `V` were declared to contain twenty elements, `char V[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`. A vector maps directly to the computer's memory.

Now we introduce a second dimension; suppose `M` is declared as a matrix, or two-dimensional array, `char M[10][15];`. 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

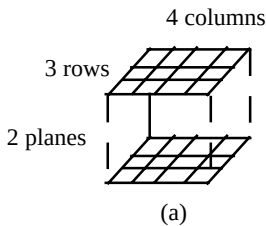
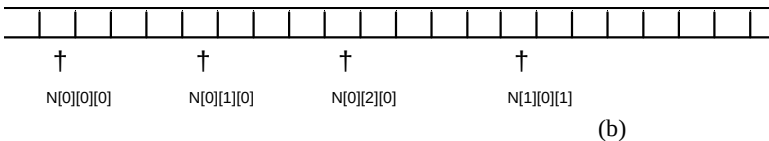


Figure 5.10 A Three-Dimensional Array `N[2][3][4]` (a) Mapped into a One-Dimensional Memory (b).



this offset the number of columns 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[Rows][Cols]`, the formula for the offset of $M[R][C]$ is $R*Cols + C$.

For a three-dimensional array, `char A[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, `A[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 $A[D_1][D_2][D_3] \dots [D_n]$, then a reference to $A[S_1][S_2][S_3] \dots [S_n]$ 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 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, assuming that arrays must be 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 added rules 8-10.

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 G23, 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 **Factor** in that rule.

Grammar G23:

7. $\text{Factor}_e \rightarrow \text{var}_v \{ \text{MOV} \}_{0,, \text{sum}} \text{Subs}_{v, \text{sum}, i}$

$$\begin{aligned} e &\leftarrow v[\text{sum}] \\ i &\leftarrow 1 \\ \text{sum} &\leftarrow \text{Alloc} \end{aligned}$$
8. $\text{Subs}_{v, \text{sum}, i1} \rightarrow [\text{Expr}_e] \{ \text{MUL} \}_{e, =D, T} \{ \text{ADD} \}_{\text{sum}, T, \text{sum}} \text{Subs}_{v, \text{sum}, i2}$

$$\begin{aligned} D &\leftarrow \text{Prod}(v, i1) \\ i2 &\leftarrow i1 + 1 \\ T &\leftarrow \text{Alloc} \end{aligned}$$
9. $\text{Subs}_{v, \text{sum}, i} \rightarrow \{ \text{check} \}_{i, v}$

The nonterminal **Subs** has three attributes: **v** (inherited) represents a pointer to the symbol table for the array being referenced, **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 function $\text{Prod}(v, i)$. This function 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 **sum** is used as an offset to the address of the variable **v**, which then 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 $\text{check}(i, v)$ which is specified by the action symbol $\{ \text{check} \}_{i, v}$ in rule 9. This action symbol represents a

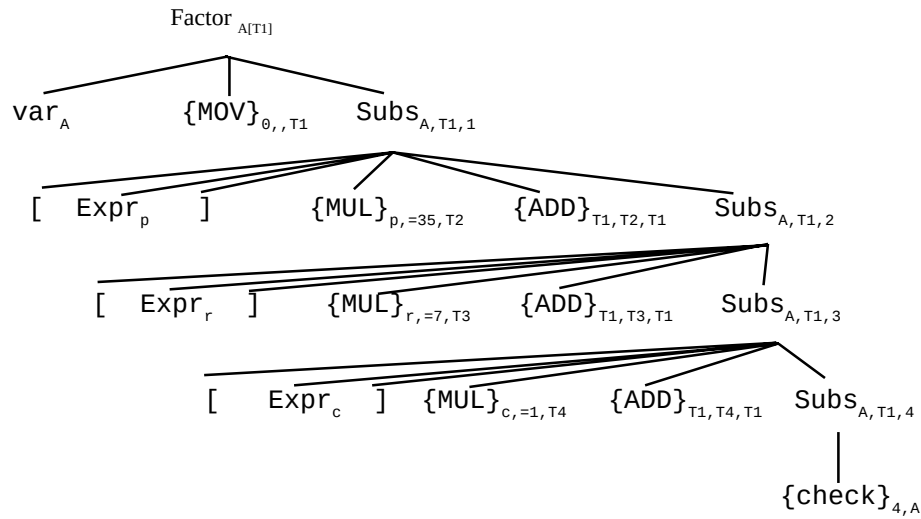


Figure 5.11 A Derivation Tree for the Array Reference $A[p][r][c]$, Which is Declared as `int A[3][5][7]`.

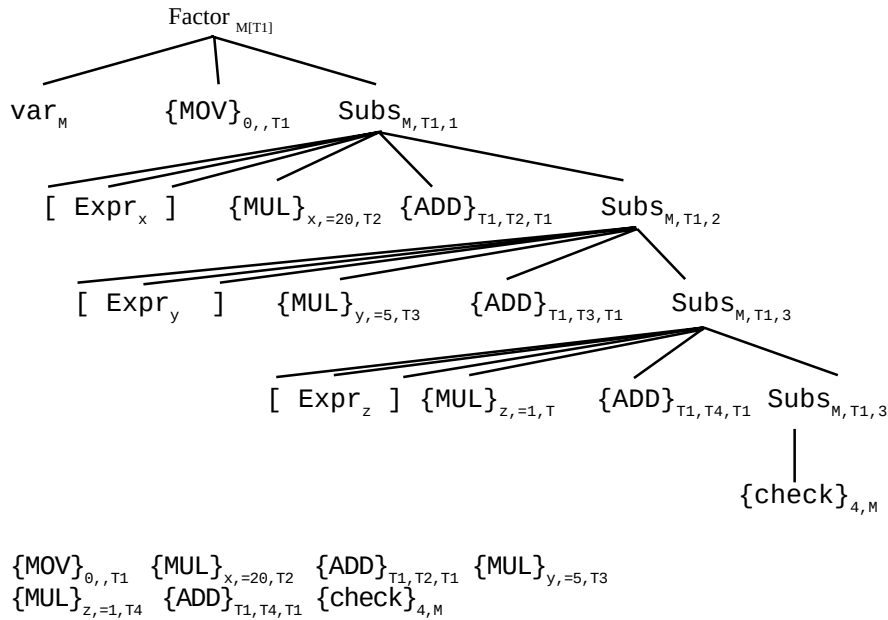
procedure call, not an atom. The purpose of the procedure is to compare the number of dimensions of the variable, v , as stored in the symbol table, with the value of i , the number of subscripts plus one. The `check(i, v)` procedure simply puts out an error message if the number of subscripts does not 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[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[13]
int M[12][17]
int A3[15][7][5]
int Z[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 C++ function –

```
int offSet (int D[max], int S[max]);
```

that computes the offset for an array reference $A[S_1, S_2, \dots, S_{\max}]$ where the array has been declared as $\text{char } A[D_1, D_2, \dots, D_{\max}]$.

(b) Improve your C++ function, if necessary, to minimize the number of run-time multiplications.

5.5 Case Study: Syntax Analysis for MiniC

In this section we continue the development of a compiler for MiniC, a small subset of the C++ programming language. We do this by implementing the syntax analysis phase of the compiler using the yacc utility as described in Section 5.3, above. The parser generated by yacc will obtain input tokens by calling `yylex()` (the function generated by lex as described in Section 2.5). The parser will then check the tokens for correct syntax.

In addition, we provide supporting functions which enable 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 MiniC compiler, syntax analysis and semantic analysis have been combined into one program.

The complete yacc source file is shown in Appendix B.2 and is explained in the following sections. The input to yacc is the file `MiniC.y` (divided into three sections by `%%` delimiters), and the output, stored in the file `y.tab.c`, is the `yyparse()` function. The `MiniC.y` file contains the main program (in the third section) which receives control when the compiler is invoked. It calls the `yyparse()` function, which in turn calls the `yylex()` function when it needs a token. As output, it produces a file of atoms. It then invokes the code generator `code_gen()` if no errors have been detected. Consequently, the program generated by yacc will ultimately contain all three phases of the compiler (the lexical phase and the code generator are brought in with `include` directives).

5.5.1 Header Files Included

Our yacc source file begins with three `include` statements enclosed in `%{` and `%}` braces, indicating that these directives are to be copied verbatim to the output file:

```
#include <stdio.h>
#include "mini.h"
#include "MiniC.h"
```

The first `include` is to support the necessary input/output function calls. The `mini.h` header file contains declarations which are needed for the back end of the compiler (code generation), but not the front end (lexical and syntax analysis).

The third header file included, `MiniC.h`, contains declarations which are needed by the front end of the compiler. It specifies constants determining the maximum size of the symbol table and the table of address labels. It also defines a type `ADDRESS`, which is intended to be a run-time address on the target machine. This header file also defines structures for the lexical tables, as described in Section 2.5.

The parser produces a file of atoms, the structure of which is defined in the `MiniC.h` header file. This record structure is shown in Figure 5.12. The remainder of the `MiniC.h` header file defines the atom classes which are not also used as machine op

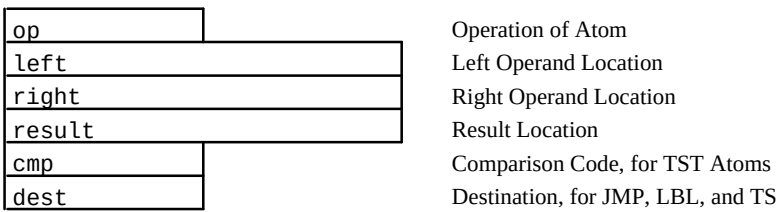


Figure 5.12 Record Structure of the File of Atoms

codes (these are defined in the `mini.h` header file because they are needed to compile the Mini machine simulator).

5.5.2 Other Declarations in the First Section

The parser stack needs to store three kinds of data: target machine addresses (`address`), integer codes for comparisons and type declarations (`code`), and label numbers (`labels`). We tell this to yacc with a `%union` statement:

```
%union
{
    ADDRESS address;
    int code;      /* comparison code 1-6 */
    struct
    {int L1;
     int L2;
     int L3; } labels;
}
```

This is similar in meaning to the `union` declaration in C; in effect, it allows the user to bind different interpretations (types) to the same storage location. When assigning a value to a syntactic type in the rules section, we can refer to any of these fields to indicate the type of the value pushed on the parser stack. For example, in the following rule:

```
Type:  terminal          {$$ .code = $1 .L1;}
;
```

the value assigned to the `Type` being defined (`$$`) is the type of `code`, whereas the type of the terminal (`$1`) is the type of `L1`. Notice that we did not need to give the structure name, `labels`, in the reference to `$1 .L1`. The reason is that yacc fills in the structure name for us.

The lexical tokens are declared with `%token` statements. We must declare a type for those tokens which have a value part (as one of the types listed in the `%union` statement) so the parser will know which type is being pushed on the stack. For ex-

In order to understand these actions the reader should refer back to Figure 4.17(a) in which the control structure of a `while` statement is diagrammed. The first atom put out is a `LBL` atom in which the destination field is a new label (`L2`). We can then refer to this in another action, but must be careful to refer to the correct grammar item (recall that actions are included when numbering the items in a grammar statement: `$1`, `$2`, `$3`, ...). The assignment to `$$ . L2` is done in the second item, so we refer to it below as `$2 . L2`. Since `$2` is an embedded action, we need to specify its type as `$<labels>2 . L1`. After the `' ) '` is seen, a `TST` atom is put out, which compares the value of the `Expr` with 0 (false). If the `Expr` is 0, it jumps out of the loop (to label `L2`). After the `Stmt`, which is the body of the loop, we put out a `JMP` atom to label `L1`, which is at the beginning of the loop.

The other control structures are handled in a similar fashion, and should be compared with Figure 4.17. If the student attempts to generate the parser for MiniC, there will be an error message from yacc: shift/reduce conflict. This error results from the ambiguous `if else` statement in the MiniC grammar. Fortunately, yacc resolves the ambiguity with a shift whenever possible, which means that all `else` clauses are matched with the closest previous unmatched `if`. In other words, the user need not be concerned about this shift/reduce error message.

Arithmetic expressions are handled with a grammar similar to grammar G4, which is ambiguous. Fortunately the ambiguity is resolved with the `%left` precedence directives described in Section 5.5.2, above.

In the processing of arithmetic expressions, each operation invokes `alloc(1)` to reserve temporary storage for the result of the operation. These temporary storage locations are never released or reused for other purposes, simply to keep the compiler short and simple. The student is encouraged to write a `release()` function which would return unused temporaries. One approach would be to locate these items after the program instructions, rather than in low memory locations, using high memory as a run-time stack. Each time a complete expression is evaluated, its result could be moved to the lowest stack location no longer needed.

5.5.4 Supporting Functions

The third section of the yacc program contains the main program as well as supporting functions. The main program opens the output file `atoms` and then installs the constants 1.0 and 0.0 (which are used as logical values True and False) in the table of constants before invoking the parser `yyparse()`.

The `yyerror()` function is called by `yyparse()` when there is a syntax error, and processing terminates. Yacc does have the capability of recovering from syntax errors, but that is beyond our scope. The `yyerror()` function is also called from `yylex()` when undeclared identifiers or multiply-declared identifiers are encountered. In this case, the program continues to look for additional errors, but no code is generated.

The most important supporting function is the `atom()` function, which generates an atom as a record in the file `atoms`, given the operation, the left and right operands, the result, the comparison code (for TST atoms), and the destination label

number (for TST and JMP atoms). The complete source input to yacc, in the file `minic.y`, is shown in appendix B.

Exercises 5.5

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

`DoStmt` $\rightarrow$ `do Stmt while '(' Expr ')'` `';' ;`

Modify the files `Minic.l` and `Minic.y`, 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 `Expr` are already defined. The tokens `do` and `while` need to be defined. 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.

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

`SwitchStmt` $\rightarrow$ `switch '(' Expr ')'` `CaseList`
`CaseList` $\rightarrow$ `case number ':'` `Stmt CaseList`
`CaseList` $\rightarrow$ `case number ':'` `Stmt`

Modify the files `Minic.l` and `Minic.y`, 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 MiniC language to include initializations in declarations, such as:

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

Modify the files `MiniC.l` and `MiniC.y`, 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:

▽ . . . HANDLE	Nt → 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

▽ . . . Nt

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 Unix utility, *yacc*, 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. *Lex*, the utility for generating lexical scanners (Section 2.4), and *yacc* are designed to work together.

Finally we looked at an *implementation of MiniC*, our case study language which is a subset of C++, using *yacc*. This compiler works with the lexical phase discussed in Section 2.4 and is shown in Appendix B.3.