One aspect of code generation that we have not mentioned is the automatic generation of a code generator using a formal description of the machine architecture, in a manner similar to the way parsers and semantic analyzers can be automatically generated. Such methods vary from purely syntactic (Glanville and Graham [1978]) to attribute based (Ganapathi and Fischer [1985]) to intermediate code based (Davidson and Fraser [1984a]). A survey of these and other methods may be found in Fischer and LeBlanc [1991].

Appendix A

Compiler Project

We define here a programming language called **C-Minus** (or C—, for short), which is a suitable language for a compiler project, which is more complex than the TINY language in that it includes functions and arrays. It is essentially a subset of C, but is missing some important pieces, hence its name. This appendix consists of five sections. In the first, we list the lexical conventions of the language, including a description of the tokens of the language. In the second, we give a BNF description of each language construct, together with an English description of the associated semantics. In the third section, we give two sample programs in C—. In the fourth, we describe a Tiny Machine runtime environment for C—. The last section describes a number of programming projects using C— and TM, suitable for a compiler course.

A.1 LEXICAL CONVENTIONS OF C—

1. The keywords of the language are the following:

```
else if int return void while
```

All keywords are reserved, and must be written in lowercase.

2. Special symbols are the following:

```
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
```

3. Other tokens are ID and NUM, defined by the following regular expressions:

```
ID = letter letter*
NUM = digit digit*
letter = a|..|z|A|..|z
digit = 0|..|9
```

4. Lower- and uppercase letters are distinct.
   White space consists of blanks, newlines, and tabs. White space is ignored except that it must separate ID's, NUM's, and keywords.
5. Comments are surrounded by the usual C notations /*...*/. Comments can be placed anywhere white space can appear (that is, comments cannot be placed within tokens) and may include more than one line. Comments may not be nested.

A.2 SYNTAX AND SEMANTICS OF C—

A BNF grammar for C— is as follows:

1. program → declaration-list
2. declaration-list → declaration-list declaration | declaration
3. declaration → var-declaration | fun-declaration
4. var-declaration → type-specifier ID ; | type-specifier ID [NUM] ;
5. type-specifier → int | void
6. fun-declaration → type-specifier ID (params) compound-stmt
7. params → param-list | void
8. param-list → param-list , param | param
9. param → type-specifier ID | type-specifier ID [ ]
10. compound-stmt → { local-declarations statement-list }
11. local-declarations → local-declarations var-declaration | empty
12. statement-list → statement-list statement | empty
13. statement → expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt
14. expression-stmt → expression ; | ;
15. selection-stmt → if (expression) statement | if (expression) statement else statement
16. iteration-stmt → while (expression) statement
17. return-stmt → return ; | return expression ;
18. expression → var = expression | simple-expression
19. var → ID | ID [expression]
20. simple-expression → additive-expression relop additive-expression | additive-expression
21. relop → <= | < | > | >= | == | !=
22. additive-expression → additive-expression addop term | term
23. addop → + | -
24. term → term mulop factor | factor
25. mulop → * | /
26. factor → (expression) | var | call | NUM

27. call → ID (args)
28. args → arg-list | empty
29. arg-list → arg-list , expression | expression

For each of these grammar rules we give a short explanation of the associated semantics.

1. program → declaration-list
2. declaration-list → declaration-list declaration | declaration
3. declaration → var-declaration | fun-declaration

A program consists of a list (or sequence) of declarations, which may be function or variable declarations, in any order. There must be at least one declaration. Semantic restrictions are as follows (these do not occur in C). All variables and functions must be declared before they are used (this avoids backpatching references). The last declaration in a program must be a function declaration of the form void main(void). Note that C— lacks prototypes, so that no distinction is made between declarations and definitions (as in C).

4. var-declaration → type-specifier ID ; | type-specifier ID [NUM] ;
5. type-specifier → int | void

A variable declaration declares either a simple variable of integer type or an array variable whose base type is integer, and whose indices range from 0 .. NUM—1. Note that in C— the only basic types are integer and void. In a variable declaration, only the type specifier int can be used. void is for function declarations (see below). Note, also, that only one variable can be declared per declaration.

6. fun-declaration → type-specifier ID (params) compound-stmt
7. params → param-list | void
8. param-list → param-list , param | param
9. param → type-specifier ID | type-specifier ID [ ]

A function declaration consists of a return type specifier, an identifier, and a comma-separated list of parameters inside parentheses, followed by a compound statement with the code for the function. If the return type of the function is void, then the function returns no value (i.e., is a procedure). Parameters of a function are either void (i.e., there are no parameters) or a list representing the function's parameters. Parameters are passed by value. Array parameters whose size can vary. Simple integer parameters are passed by reference (i.e., as pointers) and must be matched by an array variable during a call. Note that there are no parameters of type "function." The parameters of a function have scope equal to the compound statement of the function declaration, and each invocation of a function has a separate set of parameters. Functions may be recursive (to the extent that declaration before use allows).

10. compound-stmt → { local-declarations statement-list }

A compound statement consists of curly brackets surrounding a set of declarations and statements. A compound statement is executed by executing the statement

sequence in the order given. The local declarations have scope equal to the statement list of the compound statement and supersede any global declarations.

11. *local-declarations → local-declarations var-declaration* | *empty*
12. *statement-list → statement-list statement* | *empty*

Note that both declarations and statement lists may be empty. (The nonterminal *empty* stands for the empty string, sometimes written as ε.)

13. *statement → expression-stmt*
      | *compound-stmt*
      | *selection-stmt*
      | *iteration-stmt*
      | *return-stmt*

14. *expression-stmt → expression* **;** | **;**

An expression statement has an optional expression followed by a semicolon. Such expressions are usually evaluated for their side effects. Thus, this statement is used for assignments and function calls.

15. *selection-stmt →* **if** **(** *expression* **)** *statement*
      | **if** **(** *expression* **)** *statement* **else** *statement*

The if-statement has the usual semantics: the expression is evaluated; a nonzero value causes execution of the first statement; a zero value causes execution of the second statement, if it exists. This rule results in the classical dangling else ambiguity, which is resolved in the standard way: the else part is always parsed immediately as a substructure of the current if (the "most closely nested" disambiguating rule).

16. *iteration-stmt →* **while** **(** *expression* **)** *statement*

The while-statement is the only iteration statement in C−. It is executed by repeatedly evaluating the expression and then executing the statement if the expression evaluates to a nonzero value, ending when the expression evaluates to 0.

17. *return-stmt →* **return ;** | **return** *expression* **;**

A return statement may either return a value or not. Functions not declared **void** must return values. Functions declared **void** must not return values. A return causes transfer of control back to the caller (or termination of the program if it is inside **main**).

18. *expression → var* **=** *expression* | *simple-expression*
19. *var →* **ID** | **ID** **[** *expression* **]**

An expression is a variable reference followed by an assignment symbol (equal sign) and an expression, or just a simple expression. The assignment has the usual storage semantics: the location of the variable represented by *var* is found, then the subexpression to the right of the assignment is evaluated, and the value of the subexpression is stored at the given location. This value is also returned as the value of the entire expression. A *var* is either a simple (integer) variable or a subscripted array variable. A negative subscript causes the program to halt (unlike C). However, upper bounds of subscripts are not checked.

Vars represent a further restriction of C− from C. In C the target of an assignment must be an **l-value**, and l-values are addresses that can be obtained by many operations. In C− the only l-values are those given by the *var* syntax, and so this category is checked syntactically, instead of during type checking as in C. Thus, pointer arithmetic is forbidden in C−.

20. *simple-expression → additive-expression relop additive-expression*
      | *additive-expression*
21. *relop →* **<=** | **<** | **>** | **>=** | **==** | **!=**

A simple expression consists of relational operators that do not associate (that is, an unparenthesized expression can only have one relational operator). The value of a simple expression is either the value of its additive expression if it contains no relational operators, or 1 if the relational operator evaluates to true, or 0 if it evaluates to false.

22. *additive-expression → additive-expression addop term* | *term*
23. *addop →* **+** | **-**
24. *term → term mulop factor* | *factor*
25. *mulop →* ***** | **/**

Additive expressions and terms represent the typical associativity and precedence of the arithmetic operators. The **/** symbol represents integer division; that is, any remainder is truncated.

26. *factor →* **(** *expression* **)** | *var* | *call* | **NUM**

A factor is an expression enclosed in parentheses, a variable, which evaluates to the value of its variable; a call of a function, which evaluates to the returned value of the function; or a NUM, whose value is computed by the scanner. An array variable must be subscripted, except in the case of an expression consisting of a single ID and used in a function call with an array parameter (see below).

27. *call →* **ID** **(** *args* **)**
28. *args → arg-list* | *empty*
29. *arg-list → arg-list* **,** *expression* | *expression*

A function call consists of an **ID** (the name of the function), followed by parentheses enclosing its arguments. Arguments are either empty or consist of a comma-separated list of expressions, representing the values to be assigned to parameters during a call. Functions must be declared before they are called, and the number of parameters in a declaration must equal the number of arguments in a call. An array parameter in a function declaration must be matched with an expression consisting of a single identifier representing an array variable.

Finally, the above rules give no input or output statement. We must include such functions in the definition of C−, since unlike C, C− has no separate compilation or linking facilities. We, therefore, consider two functions to be predefined in the global environment, as though they had the indicated declarations:

```
int input(void) {...}
void output(int x) {...}
```

The input function has no parameters and returns an integer value from the standard input device (usually the keyboard). The output function takes one integer parameter, whose value it prints to the standard output (usually the screen), together with a newline.

A.3 SAMPLE PROGRAMS IN C—

The following is a program that inputs two integers, computes their greatest common divisor, and prints it:

```
/* A program to perform Euclid's
   Algorithm to compute gcd. */

int gcd (int u, int v)
{ if (v == 0) return u ;
  else return gcd(v,u-u/v*v);
  /* u-u/v*v == u mod v */
}

void main(void)
{ int x; int y;
  x = input(); y = input();
  output(gcd(x,y));
}
```

The following is a program that inputs a list of 10 integers, sorts them by selection sort, and outputs them again:

```
/* A program to perform selection sort on a 10
   element array. */

int x[10];

int minloc ( int a[], int low, int high )
{ int i; int x; int k;
  k = low;
  x = a[low];
  i = low + 1;
  while (i < high)
  { if (a[i] < x)
    { x = a[i];
      k = i; }
    i = i + 1;
  }
  return k;
}
```

```
void sort( int a[], int low, int high)
{ int i; int k;
  i = low;
  while (i < high-1)
  { int t;
    k = minloc(a,i,high);
    t = a[k];
    a[k] = a[i];
    a[i] = t;
    i = i + 1;
  }
}

void main(void)
{ int i;
  i = 0;
  while (i < 10)
  { x[i] = input();
    i = i + 1; }
  sort(x,0,10);
  i = 0;
  while (i < 10)
  { output(x[i]);
    i = i + 1; }
}
```

A.4 A TINY MACHINE RUNTIME ENVIRONMENT FOR THE C— LANGUAGE

The following description assumes a knowledge of the Tiny Machine as given in Section 8.7 and an understanding of stack-based runtime environments from Chapter 7. Since C— (unlike TINY) has recursive procedures, the runtime environment must be stack based. The environment consists of a global area at the top of dMem, and the stack just below it, growing downward toward 0. Since C— contains no pointers or dynamic allocation, there is no need for a heap. The basic organization of each activation record (or stack frame) in C— is

```
                  ┌───────────┐
 fp points  ────► │    ofp    │         ofpFO = 0
 here             │    ret    │         retFO = -1
                  ├───────────┤         initFO = -2
                  │    ...     │
                  │   params   │
                  │    ...     │
                  ├───────────┤
                  │    ...     │
                  │ local vars │
                  │    ...     │
                  ├───────────┤
                  │    ...     │
                  │   temps    │
                  │    ...     │
                  └───────────┘
```