C-Minus (C-)

It is essentially a subset of C in which the primary atomic data type is only **integer**, hence its name.

The lexical conventions of the language, including a description of the tokens of the language are listed.

A BNF description of each language construct, together with an English description of the associated semantics is provided.

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

Lower- and uppercase letters are distinct.

4: 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.

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 a short explanation of the associated semantics is now given

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***- I. 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 he 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 retums no value (i.e., is a procedure). Parameters of a function are cither **void** (i.e., there are no parameters) or a list representing the function's parameters. Parameters followed by brackets are array parameters whose size can vary. Simple integer parameters are passed by value. Array parameters are passed by reference (ie., 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 monterminal empty stands for the empty string, sometimes written as &)

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 substracture 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 retum a value or not. Functions not declared **void** must return values. Functions declared **void** must not return values. A retum 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 stop-age 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 eof 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. Such functions must be included in the definition of C, since unlike C, C has no separate compilation or linking facilities. Two functions to be predefined in the global environment are therefore included, 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.