The C-- Programming Language

(including User Manual for compiler)

Aman Wardak

**Introduction**

C Minus Minus (C--) is a language based on C/C++. It’s procedural and not object-oriented. It is meant to allow the programmer to have access to the most basic of the programming constructs required to develop something useful.

**Types**

Two data types are supported: int and char. One-dimensional arrays of these two types are supported as well. The size of the array must be known at compile time.

Literal strings must be enclosed in double quotes. Literal numbers may appear naked and be in decimal (base 10) format. Floating-point numbers are not supported.

**Functions**

Functions may accept any number of parameters and return a simple int, char, or be declared void. All parameters are passed by reference.

Execution begins in the function named main. It shall return an int and not take any parameters.

**Control Constructs**

If-else-if-else statements are supported. There may be any number of else-if statements after the first if. An optional else may follow at the end.

While loops are the only types of loops supported. They follow the same general syntax as the parent language. The break and continue keywords are supported.

**Input / Output**

Input and output is provided via the cin and cout keywords. Chaining is allowed, for example:

cout << “Please enter item “ << i << endl;

cin >> name >> quantity;

**Operators**

Expressions may make use of the following relational operators: ==, <, >

The following logical operators are supported: &&, ||

The following arithmetic operators are supported: +, -, \*, /

The assignment operator is supported: =

**Comments**

All comments must be enclosed in /\* and \*/. Multi-line comments are supported.

**List of all reserved keywords:**

void if cin

int else cout

char while endl

main break

return continue

**All other supported tokens and regular expressions for their lexemes**

Token Lexeme

-------- ----------------

identifier [a-zA-Z] ([a-zA-Z] | [0-9])\*

numeric literal [0-9]+

string literal “ [any char except “ or eoln]\* ”

comment / ’\*’ [any char except sequence of ‘\*/’ ]\* ‘\*’/

relational operators == | < | >

logical operators && | ||

addition operators + | -

multiplication operators \* | /

assignment operator =

input op. >>

output op. <<

left paren. (

right paren. )

left brace {

right brace }

left bracket [

right bracket ]

semicolon ;

**C-- Grammar in EBNF**

<program> ::= <function\_definition> {<function\_definition>}

<function\_definition> ::= <type> ID\_TOK ( [<parameter\_list>] ) <compound\_stmt>

<type> ::= int | char | void

<parameter\_list> ::= <type> <lvalue> { , <type> <lvalue> }

<compound\_stmt> ::= ‘{‘ {<statement>} ‘}’

<statement> ::= <var\_decl>

| <assignment>

| <if\_stmt>

| <while\_loop>

| <cin\_stmt>

| <cout\_stmt>

| <jump\_stmt>

| <compound\_stmt>

<var\_decl> ::= <type> <lvalue> ;

<assignment> ::= <lvalue> = <expression>

<expression> ::= <logical\_or\_expression>

<logical\_or\_expression> ::= <logical\_and\_expression>

| <logical\_or\_expression> || <logical\_and\_expression>

<logical\_and\_expression> ::= <equality\_expression>

| <logical\_and\_expression> && <equality\_expression>

<equality\_expression> ::= <relational\_expression>

| <equality\_expression> == <relational\_expression>

<relational\_expression> ::= <additive\_expression>

| <relational\_expression> < <additive\_expression>

| <relational\_expression> > <additive\_expression>

<additive\_expression> ::= <mul\_expression>

| <additive\_expression> + <mul\_expression>

| <additive\_expression> - <mul\_expression>

<mul\_expression> ::= <primary\_expression>

| <mul\_expression> \* <primary\_expression>

| <mul\_expression> / <primary\_expression>

<primary\_expression> ::= ( <expression> )

| <val\_token>

<val\_token> ::= <lvalue>

| LITERAL\_NUM\_TOK

| STRING\_TOK

<if\_stmt> ::= if ( <expression> ) <statement> [ else <statement> ]

<while\_loop> ::= while ( <expression> ) <statement>

<cin\_stmt> ::= cin >> <lvalue> { >> <lvalue> } ;

<cout\_stmt> ::= cout <cout\_mid> ;

<cout\_mid> ::= << <val\_token> <cout\_mid> { << <val\_token> <cout\_mid> }

| << endl <cout\_mid> { << endl <cout\_mid> }

<jump\_stmt> ::= continue ;

| break ;

| return [ <expression> ] ;

| ID\_TOK ( [<argument\_list>] ) ;

<lvalue> ::= ID\_TOK [ ‘[‘ LITERAL\_NUM\_TOK ‘]’ ]

<argument\_list> ::= <val\_token> { , <val\_token> }

**Reference:**

Kernighan and Richie, The C Programming Language, 2nd ed.

**CMM (C-- compiler)**

The source code for CMM is broken down into the following main components:

1 – Lexical Analyzer (scanner) – developed using Lex and consists of a specification file (lex\_spec.l) that contains the regular expressions for the tokens in the language. The file is fed to Flex (GNU’s version of Lex) and lex.yy.c is generated which contains the C code for the scanner.

2 – Syntax Analyzer (parser) – developed using Bison (GNU’s version of Yacc). Bison is given a file (parser.y) that includes the grammar describing the syntax of C--. Bison generates 2 files:

* parser.tab.h: contains the tokens declared in parser.y; this file is included in the scanner for accessing the token declarations
* parser.tab.c: contains the C code for the parser

In addition, the parser also includes parserUtils.cpp which contains helper functions for communicating with the Symbol Table and the Code Generator.

3 – Symbol Table – C++ class used to allow inserting of new symbols in each block and looking up symbol in active blocks. Implemented using linked-list of hashtables.

4 – Code Generator – C++ class used to write out MIPS code to the output file. Most of the code generation tasks themselves are done in parserUtils.cpp.

5 – Testcases – the testcases directory contains various small applications that show the syntax of C--. It also includes various negative testcases.

**Building CMM**

The BUILD script in the top directory is all that’s required to build the compiler. Simply run it and it should produce and executable called cmm.

**CMM Limitations**

* Missing support for functions
* Missing support for the following statements: break, continue, return
* Supports only 2 data types: int, char
* Doesn’t support arrays of more than one dimension