Compilers Project

M++

By: Omar Wael Bazaraa, Abdelrahman Eid, Ibrahim Radwan, Ahmed Samir Hamed

Introduction

M++ is a simple programming language compiler similar to the C-language but with less functionalities, yet M++ preserves the complex expressions and branch statements of the C-language.

In M++, we used *Flex* (a compiler generating tool similar to *Lex*) to generate the lexer analyzer, and we used *Bison* (a compiler generating tool similar to *Yacc*) to generate the syntax analyzer.

Although M++ is a mini-version of C-language, it provides powerful error handling techniques with meaningful error messages reporting for various syntax and semantic error scenarios.

Overview

In this section, we are going to give a brief descriptions and examples for the syntax and semantics allowed by M++. As we said, it is almost identical to C-language but with less features.

Data Types

In M++, we support the basic data types but unfortunately, we do not support arrays or pointers.
The supported types:

- **`void`**: is only valid as a function return type to tell that it has no value to return.
- **`int`**: is an integer numeric value data type.
- **`float`**: is a real numeric value data type.
- **`char`**: is a character value data type.
- **`bool`**: is a Boolean value data type that accepts either **`true`** or **`false`**.

Variable/Constant Declarations

In M++, we support scoped variables and constants declaration. Each variable or constant has its own scope, and multiple variable/constants can be declared with the same identifier only if they are in different scopes. As in C-language, constants must be initialized while being declared.

e.g.

```
int x;
const float PI = 3.14;
char c = 'c';
bool flag = true;
int a = 0, b, MAX = 100;
```

If-Else Control Statements

We support if-else control statement in almost the exact same way as in C-language. If the if-condition evaluates to a non-zero value, then the if-body will be executed. Otherwise, the else-body will be executed if exists. If-body and else-body can either be one statement, or can be multiple statements enclosed by a block.

e.g.

```
if (x) {
      if (y > 0)
            /* if-body */;
      else if (z & 1)
            /* else-if-body */;
      else
            /* else-body */;
}
```

Switch Statements

Like if-statement, we support switch-statement in almost the exact same way as in C-language. The switch-expression must be of integer value, and the case-expression must be a constant integer value. Also, multiple case-expressions that evaluate to the same value is not allowed. Like C, the code of the matched case will be executed and the execution will continue to the below code of other cases until a break-statement is found.

```
switch (state) {
case 1:
case 2:
      /* do something */
case RUNNING: // RUNNING must be defined as constant
      /* do something */
      break;
default:
      /* default */
}
```

For/While/Do-While Loops

M++ supports loops in almost the exact same way as in C-language. We support for-loops, while-loops, and do-while loops. Break-statements and continue-statements are supported within the scope of a loop, and they function like in C-language, they break or continue the execution of the inner most loop.

Functions

M++ supports functions but with limited functionalities than that of the C-language. We do not support default parameters. We do not support neither function prototyping nor function overloading. Return-statements are allowed within the scope of a function. And functions can only be defined in the global scope.

e.g.

```
int fibonacci(int n) {
	return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Expressions

In M++, we support complex expressions similar to those of C-language. We support almost the entire set of operators supported by C-language with the same precedence and associativity.

```
(((++x) = y++) = (8 * 7 - MAX) ^ (1 << i)) = (z = 3);
```

Comments

M++ supports the same comment styles as in C-language. The comments can either be:

- Line comment
  ```
  // This is a line comment
  ```
- Block comment (multi-line comment)
  ```
  /**
   * This is a block comment
   * that can span
   * multiple lines
   */
  ```

Errors Detected and Reported

In this section, we are going to list some of the syntax and semantic errors that M++ can detect and report.

Syntax Errors

Scope-related errors:

1. Code blocks or statements (other than variable, constants, and function declaration/definition) in the global scope.
2. Continue-statement outside for, while, or do-while scopes.
3. Break-statement outside for, while, do-while, or switch scopes.
4. Return-statement outside function scope.
5. Case and default labels outside switch scope.

Other syntax errors:

1. Variable or constant declared with type void.
2. Constant declaration without initialization.
3. Any other invalid syntax

Semantic Errors

Identifier/Expression-related errors

1. Identifier re-declaration in the same scope.
2. Undeclared identifier access.
3. Constant assignment after declaration.
4. Invalid operand types. (i.e. operands of type void or pointer to function).
5. Float operand to modulus operator.
6. Float operand to bitwise operators.
7. Use of uninitialized variable.
8. Increment and decrement operators with r-value operand.

Switch-statement-related errors

1. Switch and case statements with non-integer expression.
2. Case-statement with non-constant expression.
3. Multiple default-labels in switch scope.
4. Multiple case-labels with the same constant expression in switch scope.
5. Cross variables initialization in switch-statement.

Function-related errors

1. Value returned in void function.
2. Void returned in value-typed function.
3. Variable or constant call as a function.
4. Function call with more/less arguments than its parameters.
5. Function call with invalid argument type (i.e. argument of type void or pointer to function).
6. Function parameter with default value.

Tokens

We used *Flex* to write the regular expression of the allowed tokens by M++.

The following table contains a list of these regular expressions.

Token Regex	Description
`[0-9]+`	Positive integer number
`(([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*))`	Positive float number
`([eE][-+]?{INTEGER})`	Exponentiation
`({INTEGER}{EXP}|{FLOAT}{EXP}?)`	Positive real number
`[_a-zA-Z]([_a-zA-Z]|[0-9])*`	Identifier
`(\'.\')`	Character value
`"true", "false"`	Boolean values
`"void", "bool", "char", "int", "float"`	Data types
`"if", "else", "switch", "case", "default", "for", "do", "while", "break", "continue", "return", "const"`	Reserved keywords
`"=", "+", "-", "*", "/", "%", "&", "|", "^", "~", ">>", "<<", ">", ">=", "<", "<=", "==", "!=", "&&", "||", "!", "++", "--",`	Operators similar to *C*-language
`"(", ")", "{", "}", ",", ":", ";"`	Other allowed tokens
`"//"(.)*`	Line comment
`[ \t\r\n]+`	Whitespaces

Production Rules

We used *Bison* to write the production rules defining M++ syntax.

We used *Bison* precedence and associativity features to resolve the following ambiguity:

1. The precedence and associativity of the mathematical operators.
2. The dangling else problem.

The following table contains a list of the production rules used.

We use the following naming convention for the below symbols:

- Upper case words for terminal symbols
- Lower case words for non-terminal symbols

Syntax Production Rule
Main Rules
$program \rightarrow \varepsilon$ $program \rightarrow stmt\_list$
$stmt\_list \rightarrow stmt$ $stmt\_list \rightarrow stmt\_list\ stmt$ $stmt\_list \rightarrow stmt\_block$ $stmt\_list \rightarrow stmt\_list\ stmt\_block$
$stmt\_block \rightarrow \text{'\{'}\ \text{'\}'}$ $stmt\_block \rightarrow \text{'\{'}\ stmt\_list\ \text{'\}'}$
$stmt \rightarrow BREAK\ \text{';'} \mid CONTINUE\ \text{';'} \mid \text{';'}$ $stmt \rightarrow expression\ \text{';'}$ $stmt \rightarrow var\_decl\ \text{';'}$ $stmt \rightarrow if\_stmt \mid switch\_stmt \mid case\_stmt$ $stmt \rightarrow for\_stmt \mid while\_stmt \mid do\_while\_stmt\ \text{';'}$ $stmt \rightarrow function \mid return\_stmt\ \text{';'}$
Variable/Constant Declaration
$var\_decl \rightarrow type\ IDENTIFIER$ $var\_decl \rightarrow CONST\ type\ IDENTIFIER$ $var\_decl \rightarrow type\ IDENTIFIER\ \text{'='}\ expression$ $var\_decl \rightarrow CONST\ type\ IDENTIFIER\ \text{'='}\ expression$

$multi\_var\_decl \rightarrow var\_decl\ ','\ ident$ $multi\_var\_decl \rightarrow var\_decl\ ','\ ident\ '='\ expression$ $multi\_var\_decl \rightarrow multi\_var\_decl\ ','\ ident$ $multi\_var\_decl \rightarrow multi\_var\_decl\ ','\ ident\ '='\ expression$
Expressions Rules
$expression \rightarrow expr_1 \mid expr_2 \mid expr_3$
$expr_1 \rightarrow expression\ OPR_{binary}\ expression$ $expr_1 \rightarrow OPR_{unary}\ expression$ **Note:** OPR_{binary} is being substituted by all binary operators and OPR_{unary} is being substituted by all unary operators except increment and decrement operators
$expr_2 \rightarrow INC\ expression \mid DEC\ expression$ $expr_2 \rightarrow expression\ INC \mid expression\ DEC$
$expr_3 \rightarrow\ '('\ expression\ ')'$ $expr_3 \rightarrow value \mid IDENTIFIER \mid function\_call$
Branch Rules
$branch\_body \rightarrow stmt \mid stmt\_block$
$if\_stmt \rightarrow IF\ '('\ expression\ ')'\ branch\_body$ $if\_stmt \rightarrow IF\ '('\ expression\ ')'\ branch\_body\ ELSE\ branch\_body$
$switch\_stmt \rightarrow SWITCH\ '('\ expression\ ')'\ branch\_body$
$case\_stmt \rightarrow CASE\ expression\ ':'\ stmt$ $case\_stmt \rightarrow DEFAULT\ ':'\ stmt$
$while\_stmt \rightarrow WHILE\ '('\ expression\ ')'\ branch\_body$
$do\_while\_stmt \rightarrow DO\ branch\_body\ WHILE\ '('\ expression\ ')'$
$for\_stmt \rightarrow FOR\ '('\ for\_stmt\ ';'\ for\_expr\ ';'\ for\_expr\ ')'\ branch\_body$
$for\_stmt \rightarrow \varepsilon \mid var\_decl \mid expression$ $for\_expr \rightarrow \varepsilon \mid expression$

Function Rules
$function \rightarrow type\ IDENTIFIER$ '(' $param\_list$ ')' $stmt\_block$
$param\_list \rightarrow\ \varepsilon \mid var\_decl \mid param\_list\_ext$ ',' var_decl $param\_list\_ext \rightarrow var\_decl \mid param\_list\_ext$ ',' var_decl
$function\_call \rightarrow IDENTIFIER$ '(' arg_list ')'
$arg\_list \rightarrow\ \varepsilon \mid expression \mid arg\_list\_ext$ ',' $expression$ $arg\_list\_ext \rightarrow expression \mid arg\_list\_ext$ ',' $expression$
$return\_stmt \rightarrow RETURN\ expression \mid RETURN$
Other Rules
$type \rightarrow TYPE\_VOID \mid TYPE\_BOOL \mid TYPE\_CHAR \mid TYPE\_INT \mid TYPE\_FLOAT$
$value \rightarrow BOOL \mid CHAR \mid INTEGER \mid FLOAT$

Quadruples

The following table contains a list of the quadruples generated by M++.

Almost every quadruple is associated with a type. Types define the data type and the size (number of bytes) of the operands and the result of the quadruples, unless stated otherwise.

The quadruples work in the following manner:

1. Pops the operands (if any) from the top of the stack depending on their type.
2. Applies the operation defined by the quadruple.
3. Inserts the result (if available) back to the top of the stack.

In the following table, we denote the top (the last element) of the stack as $S1$, and the 2nd last element of the stack as $S2$.

Quadruple	Description
Operations Quadruples	
`ADD_<type>`	$S1 \leftarrow (S2 + S1)$
`SUB_<type>`	$S1 \leftarrow (S2 - S1)$
`MUL_<type>`	$S1 \leftarrow (S2 * S1)$
`DIV_<type>`	$S1 \leftarrow (S2/S1)$
`MOD_<type>`	$S1 \leftarrow (S2\ MOD\ S1)$
`NEG_<type>`	$S1 \leftarrow (-S1)$
`AND_<type>`	$S1 \leftarrow (S2\ AND\ S1)$
`OR_<type>`	$S1 \leftarrow (S2\ OR\ S1)$
`XOR_<type>`	$S1 \leftarrow (S2\ XOR\ S1)$
`NOT_<type>`	$S1 \leftarrow NOT(S1)$
`SHL_<type>`	$S1 \leftarrow (S2 \ll S1)$
`SHR_<type>`	$S1 \leftarrow (S2 \gg S1)$
`GT_<type>`	$S1 \leftarrow (S2 > S1)$, the result is Boolean.

`GTE_<type>`	$S1 \leftarrow (S2 \geq S1)$, the result is Boolean.
`LT_<type>`	$S1 \leftarrow (S2 < S1)$, the result is Boolean.
`LTE_<type>`	$S1 \leftarrow (S2 \leq S1)$, the result is Boolean.
`EQ_<type>`	$S1 \leftarrow (S2 = S1)$, the result is Boolean.
`NEQ_<type>`	$S1 \leftarrow (S2 \neq S1)$, the result is Boolean.
Type Conversion Quadruples	
`<type>_TO_<type>`	Converts $S1$ from the LHS type to the RHS type.
Stack Quadruples	
`PUSH_<type> <value>`	Pushes <value> to the top of the stack.
`POP_<type> <dst>`	Pops $S1$ and saves it into <dst>.
Jump Quadruples	
`JMP <label>`	Unconditional jump to the given label.
`JNZ_<type> <label>`	Jump to the given label if $S1$ is not equals to zero.
`JZ_<type> <label>`	Jump to the given label if $S1$ is equals to zero.
Function-related Quadruples	
`PROC <ident>`	Defines a new procedure.
`CALL <ident>`	Calls an already defined procedure.
`RET`	Return from a procedure.