

Compiler Construction Assignment 3 – Fall 2011

Robert van Engelen

μ C for the JVM

μ C (micro-C) is a small C-inspired programming language. Its programming constructs cover a subset of C. In this assignment we will implement a compiler `mycc` for μ C. The compiler compiles μ C programs to the Java virtual machine.

To implement the compiler we need the Java code-generation components of programming assignment 1, the lexical analyzer of programming assignment 2, and a new parser implementation based on Yacc (or Bison). This new parser utilizes translation schemes in Yacc to emit Java bytecode.

In the next programming assignment (the last assignment following this assignment) we will further extend the capabilities of our `mycc` μ C compiler by adding static semantics such as data types, apply type checking, and implement scoping rules for functions and blocks in μ C.

Download

Download the `pr3.zip` file from

<http://www.cs.fsu.edu/~engelen/courses/COP5621/pr3.zip>

After unzipping you will get the following files

Makefile	A makefile
bytecode.c	The bytecode emitter (same as Pr1)
bytecode.h	The bytecode definitions (same as Pr1)
error.c	Error reporter
global.h	Global definitions
init.c	*) Symbol table initialization
javaclass.c	Java class file operations (same as Pr1)
javaclass.h	Java class file definitions (same as Pr1)
mycc.l	*) Lex specification
mycc.y	*) Yacc specification and main program
symbol.c	*) Symbol table operations
test#.uc	A number of μ C test programs

The files marked *) are incomplete. For this assignment you need to complete these files. You can use the code you wrote for Pr1 and Pr2.

We will implement the following μ C programming constructs in this assignment:

<i>stmts</i>	\rightarrow	<i>stmts stmt</i>	Statement sequencing
		ε	
<i>stmt</i>	\rightarrow	;	Empty statement
		<i>expr</i> ;	Expression statement (assignments, function calls)
		if (<i>expr</i>) <i>stmt</i>	If-then
		if (<i>expr</i>) <i>stmt</i> else <i>stmt</i>	If-then-else (disambiguation: else matches closest if)
		while (<i>expr</i>) <i>stmt</i>	While loop
		do <i>stmt</i> while (<i>expr</i>) ;	Do-while loop
		for (<i>expr</i> ; <i>expr</i> ; <i>expr</i>) <i>stmt</i>	For loop with start, while-condition, and update expr.
		return <i>expr</i> ;	Return from program (return from function in Pr4)
		{ <i>stmts</i> }	Compound statement block

The grammar for these statements and expressions in Yacc notation is (see `mycc.y`):

```

stmts  : stmts stmt
      | /* empty */
      ;

stmt   : ';'
      | expr ';'      { emit(pop); /* does not leave a value on the stack */ }
      | IF '(' expr ')' stmt
                { /* TO BE COMPLETED */ error("if-then not implemented"); }
      | IF '(' expr ')' stmt ELSE stmt
                { /* TO BE COMPLETED */ error("if-then-else not implemented"); }
      | WHILE '(' expr ')' stmt
                { /* TO BE COMPLETED */ error("while-loop not implemented"); }
      | DO stmt WHILE '(' expr ')' ';'
                { /* TO BE COMPLETED */ error("do-while-loop not implemented"); }
      | FOR '(' expr ';' expr ';' expr ')' stmt
                { /* TO BE COMPLETED */ error("for-loop not implemented"); }
      | RETURN expr ';'
                { emit(istore_2); /* return val goes in local var 2 */ }
      | '{' stmts '}'
      | error ';'    { yyerrok; }
      ;

```

Expressions *expr* are subset C expressions composed of identifiers (the variables), integer constants, character constants, the program arguments `$0`, `$1`, ..., and most of the ANSI C operators.

The `return` statement is unusual in that it does not return control to the caller immediately, but provides the return value similar to the `output` statement in programming assignment 1. This behavior will be changed later in Pr4 when we implement functions.

In its current incomplete state the `mycc` μ C compiler can still be built with `make`. However, only the `return` statement and very simple expressions are implemented. For example, the compiler will produce executable code for the program: `return 123`; but not much else that is useful.

You will notice a lot of parse table conflicts generated by Yacc (or Bison). These conflicts (except the one for the if-then-else ambiguity) should be resolved by adding Yacc declarations for associativity and precedence.

The `yacc` and `bison` option `-v` produces `y.output` with LALR(1) parse table with the conflicts. You should inspect the `y.output` to view the LALR(1) states with shift/reduce (and possibly reduce/reduce) conflicts.

First complete the `symbol.c` insert/lookup operations, e.g. using the code you wrote for the previous project(s). Now you can build the `mycc` compiler and run it on a very simple program such as `test0.uc`. Then execute the resulting `Code` class as follows:

```
$ ./mycc test0.uc
Compilation successful: saving Code.class
$ java Code
123
```

If you get a "java.lang.NoClassDefFoundError", set the `CLASSPATH` to include ".". The next test requires one command-line argument:

```
$ ./mycc test1.uc
Compilation successful: saving Code.class
$ java Code 102109
102109
```

The next program doesn't compile, because the operators and their associativity and precedence levels have not been defined yet:

```
$ ./mycc test2.uc
+ operator not implemented
```

Lex Specification

Our `mycc` μ C compiler accepts a subset of the ANSI C grammar. It relies on the lexical analyzer `mycc.l` to provide a token stream that is compliant with ANSI C (although some simplifications are applicable). Thus, the lexical analyzer must be able to recognize the full set of ANSI C keywords, operators, identifiers, and literal constants. You can reuse the Lex specification of assignment 2 to extend `mycc.l`. The Lex actions should provide tokens and token values (`yy1val`) to the Yacc-based parser.

Yacc Specification

The grammar and main program are defined in `mycc.y`. The grammar should be completed with semantic actions to emit the correct Java bytecode. To implement semantic actions you should use marker nonterminals. The actions for three marker nonterminals `L`, `N`, and `M` are already defined in `mycc`. These three marker nonterminals have actions that are used to direct the control flow for conditional programming constructs and loops with conditional and unconditional jumps. As usual, backpatching should be used to set the target location of a forward jump.

The current grammar for expressions is ambiguous and the appropriate declarations for operator associativity and precedence should be added to disambiguate the grammar, except for the ambiguous if-then-else. Thus, only one shift-reduce conflict should remain.

Arithmetic expressions are only performed on integers. So you can ignore floats and strings, even though we define the content of the Yacc stack's attributes to include symbols (for identifiers), numbers (for integers), floats, strings, and locations (for backpatching). The Yacc specification of the alternate data types of its synthesized attributes are defined in a Yacc `%union`:

```
%union
{ Symbol *sym; /* token value yylval.sym is the symbol table entry of an ID */
  unsigned num; /* token value yylval.num is the value of an int constant */
  float flt; /* token value yylval.flt is the value of a float constant */
  char *str; /* token value yylval.str is the value of a string constant */
  unsigned loc; /* location of instruction to backpatch */
}
```

Each grammar symbol can have one of these types for its synthesized attribute. For example:

```
%token <sym> ID
```

defines `ID` to be a token with attribute type `sym`. Thus, `yylval.sym` is the attribute value of token `ID` in the Lex specification and the attribute value in the Yacc specification is referenced with `$i`, as in:

```
| ID          { emit2(iloop, $1->localvar); }
```

Note that a global counter `localvar` in `mycc.1` is used to assign JVM local variable indexes to variables in the source code, so all variables in the μC source code are mapped to JVM local variables.

The objective of this assignment is to implement all semantic actions required to support the procedural programming constructs and integer arithmetic of μC .

A Note on Short-Circuit Operators

For this assignment you **should not implement short-circuit code** for the operators `||`, `&&`, and `!` (textbook section 6.6.2 uses short-circuit for conditional operators). But rather these should be implemented as logical operators taking values 0 (false) or 1 (true) by utilizing the following mapping:

```
a||b = a|b
a&& b = a&b
!a    = 1-a
```

Bonus for Extra Credit

You can earn 1% extra credit on the total final grade of this course by implementing a `break` statement that terminates the (closest) loop construct (`do`, `while` and `for` loop). Note that loops may be nested, and multiple `break` statements may appear at the same or at different loop levels. Implementing this is not trivial, but the challenge is rewarding. Evaluate your approach first before implementing it.