Compiler Design Assignment 3 - Spring 2021

Assignment designed by Robert van Engelen

μc for the JVM

μc (micro-C) is a small C-inspired programming language. In this assignment we will implement a compiler in C++ for μc. The compiler compiles μc programs to java class files for execution with the Java virtual machine.

To implement the compiler, we can reuse the same concepts in the code-generation parts that were done in programming assignment 1 and reuse parts of the lexical analyzer you implemented in programming assignment 2. We will implement a new parser based on Yacc/Bison. This new parser utilizes translation schemes defined in Yacc grammars to emit Java bytecode.

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

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 μc test programs

The files marked $*$) are incomplete. For this assignment you are required to complete these files. You can reuse parts of the code you wrote for Pr1 and Pr2.

Download RE/flex from `https://sourceforge.net/projects/re-flex` and build with `./build.sh`. The Flex documentation is at `http://dinosaur.compilertools.net/flex`. The RE/flex documentation is at `http://www.cs.fsu.edu/~engelen/doc/reflex/html` (RE/flex has additional features compared to Flex). The Makefile assumes that reflex is located in a local directory in the current project directory, but you are free to adapt this to reflect your installation of RE/flex.

We will use the following μc programming constructs in this assignment to implement a parser:

stmts	→	*stmts stmt*	Statement sequencing
	|	ε	
stmt	→	`;`	Empty statement
	|	*expr* `;`	Expression statement (assignments, function calls)
	|	`if` `(` *expr* `)` *stmt*	If-then
	|	`if` `(` *expr* `)` *stmt* `else` *stmt*	If-then-else (disambiguation: else matches closest if)
	|	`while` `(` *expr* `)` *stmt*	While loop
	|	`do` *stmt* `while` `(` *expr* `)` `;`	Do-while loop
	|	`for` `(` *expr* `;` *expr* `;` *expr* `)` *stmt*	For loop with start, while-condition, and update expr.
	|	`return` *expr* `;`	Return from program (return from function in Pr4)
	|	{ *stmts* }	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); /* do 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; }
        ;
expr    : ID   '=' expr { emit(dup); emit2(istore, $1->localvar); }
        ...
```

Expressions *expr* are a subset ANSI C expressions and composed of identifiers (variables), integer constants, character constants, a special form to refer to the program's command-line arguments denoted by `$0`, `$1`, `$2`, etc. and most of the ANSI C operators defined for expressions (see the `mycc.y` file).

The `return` statement returns from the program with a return value (this behavior will be extended to implement function returns later in Pr4 when we implement functions).

In its current incomplete state the μc compiler `mycc` can be built with `make`. However, it won't run on any input since the minimum requirement for keyword lookup and parsing is not complete yet (in `symbol.c`).

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 shell variable 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 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 (`yylval`) 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 through 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(iload, $1->localvar); }
```

Note that a global counter `localvar` in `mycc.l` 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.

- *End*