Compiler Construction Assignment 1 – Spring 2018

Robert van Engelen

In this assignment we construct a compiler in C that translates source program code into target JVM bytecode for execution by the JVM. The programming language is given by the following grammar:

stmt	→	'{' *opt_stmts* '}'
	|	`ID` = *expr* **;**
	|	**if** '(' *expr* ')' *stmt* **else** *stmt*
	|	**while** '(' *expr* ')' *stmt*
	|	**return** *expr* **;**
opt_stmts	→	*stmt opt_stmts*
	|	ε
expr	→	*term moreterms*
moreterms	→	+ *term moreterms*
	|	− *term moreterms*
	|	ε
term	→	*factor morefactors*
morefactors	→	* *factor morefactors*
	|	/ *factor morefactors*
	|	% *factor morefactors*
	|	ε
factor	→	'(' *expr* ')'
	|	- *factor*
	|	`NUM`
	|	`ID`
	|	**arg** '[' `NUM` ']'

where `NUM` is an integer constant **arg[ ]** is a input argument indexed from 0 to 127, and `ID` is an identifier composed of alphanumeric characters and underscores.

1 Scanner

We should first write a "lexer", (lexical analyzer a.k.a. scanner) in C. The code is similar to the `lexan()` function in `lexer.c` shown here:

```
/****   lexer.c   ****/

#include "global.h"
```

```
char lexbuf[BSIZE];
int lineno = 1;
int tokenval = NONE;

int lexan()
{
  int t;
  while (1)
  { t = getchar();
    if (t== ' ' || t == '\t')
      ;
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit(t))
    { ungetc(t, stdin);
      scanf("%d", &tokenval);
      return NUM;
    }
    else if (isalpha(t) || t == '_')
    { int p, b = 0;
      while (isalnum(t) || t == '_')
      { lexbuf[b] = t;
        t = getchar();
        b = b + 1;
        if (b >= BSIZE)
          error("compiler error");
      }
      lexbuf[b] = EOS;
      if (t != EOF)
        ungetc(t, stdin);
      p = lookup(lexbuf);
      if (p == 0)
      { printf("calling insert\n");
        p = insert(lexbuf, ID);
      }
      tokenval = p;
      return symtable[p].token;
    }
    else if (t == EOF)
      return DONE;
    else
    { tokenval = NONE;
      return t;
    }
  }
}

/****   error.c   ****/

#include "global.h"

error(char *m)
{
  fprintf(stderr, "line %d: %s\n", lineno, m);
  exit(1);
}
```

terminal	*token*	*tokenval*
+	'+'	–
=	'='	–
⋮	⋮	⋮
if	IF (256)	–
⋮	⋮	⋮
eof	DONE (300)	–
ID	ID (301)	JVM frame local variable index ≥ 3
INT8	INT8 (302)	8-bit non-negative integer value (0...127))
INT16	INT16 (303)	16-bit non-negative integer value (0...32767)
INT32	INT32 (304)	32-bit non-negative integer value (0...2147483647)

Table 1: Terminals, tokens, and token attributes

The lexical analyzer is a routine `lexan()` that is called by the parser to find tokens. The routine reads the input one character at a time and returns to the parser the token it found. The value of the attribute associated with the token is assigned to a global variable `tokenval`. The lexical analyzer uses the symbol-table routine `lookup` to determine whether an identifier lexeme has been previously seen and the routine `insert` to store a new lexeme into the symbol table. It also increments a global variable `lineno` every time it sees a newline character.

The scanner that we need to implement should recognize operators, keywords, non-negative integers, and identifiers as tokens. To do so, we use ASCII codes for single-character tokens such as `+`, and token codes above 255 for keywords. For example as shown in Table 1: The first column in the table lists the terminals used by the grammar (the grammar will be defined in the next section). We replace token `NUM` with three tokens, namely `INT8`, `INT16`, and `INT32` to represent 8-, 16-, and 32-bit integers whose values are stored in the `int tokenval` variable. The token `ID` is an identifier starting with a letter or underscore followed by letters, digits, and/or underscores. The second column lists the operator tokens you need for your program.

Create a `global.h` file with your global declarations of the tokens:

```
/****   global.h   ****/

#include <stdio.h>
#include <ctype.h>

#define BSIZE 128
#define NONE  -1
#define EOS   '\0'

#define IF     256
... /* to be completed */
#define DONE   300
#define ID     301
#define INT8   302
#define INT16  303
#define INT32  304

int tokenval;
int lineno;
```

```
struct entry
{   char *lexptr;
    int token;
};

struct entry symtable[];
```

The keywords are to be stored in the symbol table at initialization of the table (see `symbol.c` and `init.c`:

```
/****   symbol.c   ****/

#include "global.h"

#define STRMAX 999
#define SYMMAX 100

char lexemes[STRMAX];
int lastchar = -1;
struct entry symtable[SYMMAX];
int lastentry = 0;

int lookup(char s[])
{ int p;
  for (p = lastentry; p > 0; p = p - 1)
    if (strcmp(symtable[p].lexptr, s) == 0)
      return p;
  return 0;
}

int insert(char s[], int tok)
{ int len;
  len = strlen(s);
  if (lastentry + 1 >= SYMMAX)
    error("symbol table full");
  if (lastchar + len + 1 >= STRMAX)
    error("lexemes array full");
  lastentry = lastentry + 1;
  symtable[lastentry].token = tok;
  symtable[lastentry].lexptr = &lexemes[lastchar + 1];
  lastchar = lastchar + len + 1;
  strcpy(symtable[lastentry].lexptr, s);
  return lastentry;
}

/****   init.c   ****/

#include "global.h"

struct entry keywords[] =
{ "if",     IF,
  ... /* to be completed */
  0, 0
};

init()
```

```
{ struct entry *p;
  for (p = keywords; p->token; p++)
    insert(p->lexptr, p->token);
}
```

The `tokenval` of an identifier `ID` refers to the frame slot where the value of the `ID` will be held at run time, which will have to start at 3. So the first identifier's `tokenval` is 3, the next identifier's `tokenval` is 4, and so on. The end-of-file token is `DONE`.

Make sure to test your scanner on standard input. To do so, write a `main()` function in a new file `lextest.c` that uses your `lexan()` scanner to print the <*token*,*tokenval*> pairs scanned from standard input. For example:

```
./lextest < mytest
```

with `mytest` a file containing:

```
{ n = 2; m = 1000*n+arg[0]; }
```

The program should print:

```
<{,-1> <ID,3> <=,-1> <INT8, 2> <;,-1> <ID,4> <=,-1> <INT16,1000> <*,-1> <ID,3> <+,-1>
<ARG,-1> <[,-1> <INT8,0> <],-1> <;,-1> <},-1>
```

Write a `Makefile` to build `lextest`. If you are not familiar with `make` or your experience with `make` is rusty, visit

`http://www.gnu.org/software/make/manual/html_chapter/make_toc.html`

Remember that tabs are significant in a Makefile (so copy-pasting the contents of Makefiles is problematic). An example `Makefile` is included in the software that you will have to download for the second part of the assignment in the next section.

To recap, for this first part of the assignment, you should implement the source code files `global.h`, `lexer.c`, `init.c`, `error.c`, `symbol.c`, `lextest.c`, and a `Makefile` and test your scanner. Before you proceed with the next section, make sure your scanner works fine. You don't need to submit the `lextest` code for grading. The lexical analyzer will be tested and graded together with the rest of the project code after you completed the next part.

2 Translation Schemes

Download the `Pr1.zip` file from

`http://www.cs.fsu.edu/~engelen/courses/COP5621/Pr1.zip`

Note: when unzipping be careful not to overwrite your Makefile. Unzip the archive in a safe place. The archive contains

`Makefile`	a makefile to build the example program
`bytecode.h`	JVM opcodes
`bytecode.c`	a bytecode emitter
`javaclass.h`	Java class file structures
`javaclass.c`	Java class file structure operations
`calcclass.c`	example program to test the bytecode emitter

The `calcclass.c` source shows how the structure of the Java class file is set up and how the bytecode emitter is invoked.

Execute the command `make calcclass` on a Unix/Linux machine to build `calcclass`. When you manually run the `calcclass` program (or when you type `make Calc`), it saves a Java class file `Calc.class` in the current directory. This Java program is a simple test program that adds the two values provided at the command line when you run it:

```
java Calc 2 3
```

This should print `5`. If you get a `java.lang.NoClassDefFoundError` then set the `CLASSPATH` environment variable with `setenv` to point to the directory with the class file, e.g. `setenv CLASSPATH .` to load the class from the current directory.

To view the Java bytecode, disassemble the class file with:

```
javap -c Calc
```

The code that you will see corresponds to the following class definition:

```
import java.lang.*;
public class example
{  public static void main(String[] arg)
        {       int[] val = new int[arg.length];
                for (int i = 0; i < arg.length; i++)
                        val[i] = Integer.parseInt(arg[i]);
                int result;
                result = val[0] + val[1];
                System.out.println(result);
        }
}
```

Notice that the command-line arguments are converted to ints and stored in array `val`. The `result` variable is used to store the result of the computation and print it to standard out.

In this programming assignment we will generate bytecode from a source program that replaces the part in the code above `result = val[0] + val[1];` with the bytecode generated by our compiler from our own programs. So the code shown above just functions a a template to handle the integer-valued inputs (stores these in `val`$[i]$) and prints the output (stored in `result`).

In the generated code for expressions, command-line arguments that are provided to the compiled program are pushed on the operand stack with the following instructions

```
aload_1
```

```
bipush nnn
iaload
```

where `nnn` is the index. Local temporary variables are always available in the frame (as long as `max_local` is large enough). A local variable's value (r-value) is pushed on the stack with `iload nnn` where `nnn` is the index of the variable (note that variables 0 to 2 are already occupied). A local variable is assigned with the popped value on the stack using `istore nnn` where `nnn` is the index of the variable. Constant integers are pushed with `bipush`, `sipush`, and `ldc`, where the last operation requires storing the 32-bit integer value in the constant pool.

This is the partially completed translation scheme:

stmt	→	'{' *opt_stmts* '}'
	|	`ID` { *var* = tokenval } = *expr* { emit2(`istore`, *var*); } `;`
	|	**if** '(' *expr* ')' { emit(`iconst_0`); *loc* := pc; emit3(`if_icmpeq`, 0); }
		stmt { backpatch(*loc*, pc−*loc*); } **else** { *...goto next statement ...* } *stmt*
	|	**while** '(' { *test* := pc } *expr* ')' { emit(`iconst_0`); *loc* := pc; emit3(`if_icmpeq`, 0); }
		stmt { emit3(`goto_`, *test*-pc); backpatch(*loc*, pc−*loc*); }
	|	**return** *expr* { emit(`istore_2`); *...goto end of code ...* } `;`
opt_stmts	→	*stmt opt_stmts*
	|	ε
expr	→	*term moreterms*
moreterms	→	+ *term* { *...?...* } *moreterms*
	|	− *term* { *...?...* } *moreterms*
	|	ε
term	→	*factor morefactors*
morefactors	→	* *factor* { *...?...* } *morefactors*
	|	/ *factor* { *...?...* } *morefactors*
	|	% *factor* { *...?...* } *morefactors*
	|	ε
factor	→	'(' *expr* ')'
	|	- *factor* { *...?...* }
	|	`INT8` { emit2(`bipush`, tokenval); }
	|	`INT16` { emit3(`sipush`, tokenval); }
	|	`INT32` { emit2(`ldc`, constant_pool_add_Integer(&cf, tokenval)); }
	|	`ID` { *...emit?...* }
	|	**arg** '[' `INT8` ']' { emit(`aload_1`); emit2(`bipush`, tokenval); emit(`iaload`); }

where the start symbol of the grammar is *stmt*.

Complete the parts indicated by *...?...*.

Example programs can be found in the Appendix of this document. The usual C-like (!) semantics of the constructs are applicable. Thus, assignments are expected to overwrite the value of the identifier assigned, the if-then construct evaluates a condition (zero is false, nonzero is true), and the while loop executes the statement (or block statement) as long as the condition is true (nonzero). The **output** statement is special. It specifies the output value the program needs to return, without actually returning to the flow of control to the caller. Thus, multiple **output** statements can be executed, but only the last value will be returned

to the caller when the execution of the statement (block) ends.

Note that `arg[0]` is the first program argument, `arg[1]` is the second, and so on (these values are stored in the `val` array of the template code listed above). Identifiers refer to local temporary variables. These local variables are allocated in the *frame* of the method by the JVM, so the code simply refers to the appropriate *frame slots* starting with 3 (the first 3 slots are used by the method argument array and local variables used to convert command-line arguments to an integer array).

Write a predictive parser for the grammar (in a new source file `compile.c`). The semantic actions in the grammar should be part of the predictive parser. The semantic actions emit the bytecode translation of the program being compiled.

To complete the compiler, you also have to write the necessary code to set up the internal Java class file structure. Use the `javaclass.h` API functions developed for this assignment to do so. See `calcclass` for an example. Name the class "`Code`", add a `static public void main` method with `String[]` argument, generate the method's code using the bytecode emitter, and save the Java class file. Consult the `calcclass.c` source for examples.

The initial part of the bytecode of the method is always the same. This code converts the command-line arguments into an integer array that is accessed with the `arg` references in expressions. The initial part of the code before the generated part is:

```
emit(aload_0);
emit(arraylength);              // arg.length
emit2(newarray, T_INT);
emit(astore_1);                 // val = new int[arg.length]
emit(iconst_0);
emit(istore_2);                 // i = 0
label1 = pc;                    // label1:
emit(iload_2);
emit(aload_0);
emit(arraylength);
label2 = pc;
emit3(if_icmpge, PAD);          // if i >= arg.length then goto label2
emit(aload_1);
emit(iload_2);
emit(aload_0);
emit(iload_2);
emit(aaload);                   // push arg[i] parameter for parseInt
index1 = constant_pool_add_Methodref(&cf, "java/lang/Integer", "parseInt",
    "(Ljava/lang/String;)I");
emit3(invokestatic, index1);    // invoke Integer.parseInt(arg[i])
emit(iastore);                  // val[i] = Integer.parseInt(arg[i])
emit32(iinc, 2, 1);             // i++
emit3(goto_, label1 - pc);      // goto label1
backpatch(label2, pc - label2); // label2:
```

The above code is followed by the emitted code from the semantic actions in the grammar. The finalize the code, the following code is to be added:

```
... backpatch operation(s) to complete the goto_ ops for each 'return' ...
index2 = constant_pool_add_Fieldref(&cf, "java/lang/System", "out",
    "Ljava/io/PrintStream;");
```

```
emit3(getstatic, index2);       // get static field System.out of type PrintStream
emit(iload_2);                  // push parameter for println()
index3 = constant_pool_add_Methodref(&cf, "java/io/PrintStream", "println", "(I)V");
emit3(invokevirtual, index3);   // invoke System.out.println(result)
emit(return_);                  // return
```

Note that the backpatch operations are needed when the source program uses 'return', since each 'return' will need to jump with a `goto_` to the end of the code, which is the statement after these backpatch operations to print out the returned value.

You may assume that the maximum number of local variables `max_local` is 127. Similarly you may assume that the maximum operand stack size `max_stack` is 127.

For this task you will need to use the `bytecode.h`, `bytecode.c`, `javaclass.h`, and `javaclass.c` sources, use your `global.h`, `lexer.c`, `init.c`, `error.c`, and `symbol.c` from the previous section, write the `compile.c` source, and update the `Makefile` to generate `compile`. Submit all of these and the `lextest.c` source for evaluation and grading.

A Test Programs

A.1 Program 1

```
return arg[0] + arg[1];
```

Compile: `compile < prog1`

Example input: `java Code 3 5`

Example output: `8`

A.2 Program 2

```
{
  if ( arg[0] )
    if ( arg[1] )
      return 1;
    else
      return 2;
  else
    return 3;
}
```

Compile: `compile < prog2`

Example input: `java Code 2 1`

Example output: `1`

A.3 Program 3

```
{
  n = arg[0];
  fac = 1;
  while ( n )
  {
    fac = fac * n;
    n = n - 1;
  }
  return fac;
}
```

Compile: `compile < prog3`

Example input: `java Code 3`

Example output: `6`

A.4 Program 4

```
{
  pow = 1;
  n = arg[0];
  m = arg[1];
  while ( m )
  {
    if ( m % 2 )
    {
      pow = pow * n;
      m = m - 1;
    }
    else { }
    if ( (m - 1) % 2)
    {
      n = n * n;
      m = m / 2;
    }
    else { }
  }
  return pow;
}
```

Compile: `compile < prog4`

Example input: `java Code 3 5`

Example output: `243`