

Compiler Construction Assignment 4 – Fall 2011

Robert van Engelen

μ C for the JVM Part 2

In this programming assignment we complete our μ C compiler. First, we extend its syntax with functions and code blocks. We will adopt the C scoping rules for local variables in functions and blocks. To implement scoping we need to implement a symbol table hierarchy as discussed in class. Second, we implement a simple type system for our strongly-typed μ C, for which we will add the `void` and `float` types and type checking of expressions and function arguments. Finally, we implement short-circuit evaluation of Boolean expressions using backpatch lists as discussed in class.

Download

Download the `pr4.zip` file from

<http://www.cs.fsu.edu/~engelen/courses/COP5621/pr4.zip>

Be careful not to overwrite your files of the previous assignment Pr3, because you will need your Pr3 solution to complete Pr4. After unzipping you will get

Makefile	Build dependences
backpatch.c	* Backpatch list operations
bytecode.c	Bytecode emitter
bytecode.h	Bytecode definitions
error.c	Error reporter
global.h	* Global definitions
init.c	Symbol table initialization
javaclass.c	Java class file operations
javaclass.h	Java class file definitions
mycc.l	* Your Lex specification from Pr2/Pr3
mycc.y	* Yacc specification and main program
symbol.c	* Your symbol table from Pr2/Pr3
table.c	Symbol table hierarchy operations
type.c	Type-related operations for the JVM
test#.uc	Test programs
output.txt	example javap bytecode output for tests 1 to 5

You need to complete the files marked * for this assignment.

The μC programming constructs is a subset of the C constructs. To enhance the capabilities of our μC compiler, we will implement the following μC programming constructs in this assignment.

A program *prog* consists of a sequence of *exts* “externals”, consisting of global definitions of functions *func* and variable declarations *decl*:

$$\begin{array}{lcl}
 \textit{prog} & \rightarrow & \textit{exts} \\
 \textit{exts} & \rightarrow & \textit{exts func} \\
 & & | \textit{exts decl} \\
 & & | \varepsilon
 \end{array}$$

Each function is declared with the following syntax:

$$\textit{func} \rightarrow \textit{type ID (args) block}$$

and we have a special function `main` representing the main program (with no arguments and no return value):

$$\textit{func} \rightarrow \text{main () block}$$

A *block* is a block of local variable declarations *decls* followed by statements *stmts* with the following syntax:

$$\textit{block} \rightarrow \{ \textit{decls stmts} \}$$

A comma-separated list of formal arguments *args* of a function has the following syntax:

$$\begin{array}{lcl}
 \textit{args} & \rightarrow & \textit{args , type ID} \\
 & & | \textit{type ID}
 \end{array}$$

The global declarations (externals) and local declarations of variables in a block are defined with the following syntax, which closely resembles the syntax of variable declarations in C limited to `void`, `int`, and `float` types:

$$\begin{array}{lcl}
 \textit{decls} & \rightarrow & \textit{decls decl} \\
 & & | \varepsilon \\
 \textit{decl} & \rightarrow & \textit{list ;} \\
 \textit{list} & \rightarrow & \textit{list , ID} \\
 & & | \textit{type ID} \\
 \textit{type} & \rightarrow & \text{void} \\
 & & | \text{int} \\
 & & | \text{float}
 \end{array}$$

The syntax of statements *stmt* and expressions *expr* introduced in Project 3 remain the same for this Project 4, with the addition of a function call construct:

<i>expr</i>	→	... as in Pr3...
		ID (<i>exprs</i>)
<i>exprs</i>	→	<i>exprs</i> , <i>expr</i>
		<i>expr</i>

After download the mycc compiler can be built by running `make` and used even in its current incomplete state. But only very (very!) simple programs can be compiled.

A simple program that can be compiled and run is this one `test0.uc`:

```
main() { return $0; }
```

Similar to Project 3, the `$0` refers to the command-line argument. The `return` statement used specifically in `main` is unusual in that it does not return control to the caller immediately, but provides the return value as in Project 3. The behavior of `return` will be correct in this project for functions (other than `main`), where the behavior of `return` is as usual and returns control to the caller.

In addition to the new language constructs described above, we will implement scoping for global and local variables, short-circuit evaluation, functions, and type checking.

1 Task 1: Implementing Local Variables

Use parts of your Pr3 Yacc specification `mycc.y` to implement the code generation for arithmetic operations in this new `mycc.y` project file. In this first step you should only consider modifying the semantic actions for productions for *expr* and adding semantic actions to emit code for statements *stmt* as you did for Pr3. Also add the appropriate directives to define the associativity of the arithmetic operators as in Pr3.

Now your compiler implementation matches the power of your Pr3 version of your compiler, except that local variables need to be handled differently compared to Pr3. The use of `localvar` of the `Symbol` struct is obsolete, because we no longer assume that all variables are global and can be stored in a single JVM frame. Instead, we use a symbol table hierarchy to store information on global variables and functions and local variables, as explained in class. The table structure `Table` is declared in `global.h` and operations on the tables are defined in `table.c`.

Note that at the very begin of the Yacc rules nonterminals `prog` and marker nonterminal `Mprog` in `mycc.y` create and push a new table on the table pointer stack `tblptr` and initializes the offset stack `offset`, and pops the table pointer and offset stacks, respectively:

```
prog      : Mprog exts  { addwidth(top_tblptr, top_offset);
                        pop_tblptr;
                        pop_offset;
                        }
;
Mprog    :              { push_tblptr(mktable(NULL));
                        push_offset(0);
                        }
;
```

The push and pop operations are implemented as macros in `mycc.y`.

Upon entering a local scope for function `main`, we need to create a new table with `mktable`, link it to the previous one, and push it on the table pointer stack `tblptr`. When exiting the local scope, we pop the `tblptr` and `offset` and use `enterproc` to store the function in the global table. The `tblptr` stack is used to refer to the table of the innermost scope using `top_tblptr`. The `offset` stack is used to compute the “place” of a variable. For local variables, the place information is the position at which the local variable will be stored in the JVM frame’s local variable area (similar to the `localvar` value used in Pr3). To compute the place, we use the `offset` stack to keep track of the next available place number for a local variable. For global variables introduced later, we use the constant pool index as the place information for a global variable.

Before you start, take a look at the table management and code emitted for the `main` function shown below. Note that marker nonterminal `Mmain` is responsible for the familiar initial JVM-related work before the body of the function is parsed, while the semantic actions for `func` emit the code after parsing the body:

```
func : MAIN '(' ')' Mmain block
{ // need a temporary table pointer
  Table *table;
  // the type of main is a JVM type descriptor
  Type type = mkfun("[Ljava/lang/String;", "V");
  // emit the epilogue part of main()
  emit3(getstatic, constant_pool_add_Fieldref(&cf, "java/lang/System", "out", "Ljava/io/PrintStream;"));
  emit(iloop_2);
  emit3(invokevirtual, constant_pool_add_Methodref(&cf, "java/io/PrintStream", "println", "(I)V"));
  emit(return_);
  // method has public access and is static
  cf.methods[cf.method_count].access = ACC_PUBLIC | ACC_STATIC;
  // method name is "main"
  cf.methods[cf.method_count].name = "main";
  // method descriptor of "void main(String[] arg)"
  cf.methods[cf.method_count].descriptor = type;
  // local variables
  cf.methods[cf.method_count].max_locals = top_offset;
  // max operand stack size of this method
  cf.methods[cf.method_count].max_stack = 100;
  // length of bytecode is in the emitter's pc variable
  cf.methods[cf.method_count].code_length = pc;
  // must copy code to make it persistent
  cf.methods[cf.method_count].code = copy_code();
  if (!cf.methods[cf.method_count].code)
    error("Out of memory");
  // advance to next method to store in method array
  cf.method_count++;
  if (cf.method_count > MAXFUN)
    error("Max number of functions exceeded");
  // add width information to table
  addwidth(top_tblptr, top_offset);
  // need this table of locals for enterproc
  table = top_tblptr;
  // exit the local scope by popping
  pop_tblptr;
  pop_offset;
  // enter the function in the global table
  enterproc(top_tblptr, $1, type, table);
}
```

The marker nonterminal `Mmain` does all the initial work before the body of `main` is parsed:

```
Mmain :
{ int label1, label2;
  Table *table;
  // create new table for local scope of main()
  table = mktable(top_tblptr);
  // push it to create new scope
  push_tblptr(t);
  // for main(), we must start with offset 3 in the local variables of the frame
  push_offset(3);
  // init code block to store stmts
  init_code();
  // emit the prologue part of main()
  emit(aload_0);
  emit(arraylength);
  emit2(newarray, T_INT);
  emit(astore_1);
  emit(iconst_0);
  emit(istore_2);
  label1 = pc;
  emit(iloop_2);
  emit(aload_0);
  emit(arraylength);
  label2 = pc;
  emit3(if_icmpge, PAD);
  emit(aload_1);
  emit(iloop_2);
  emit(aload_0);
  emit(iloop_2);
  emit(aaload);
  emit3(invokestatic, constant_pool_add_Methodref(&cf, "java/lang/Integer", "parseInt", "(Ljava/lang/String;)I"));
  emit(iastore);
  emit32(iinc, 2, 1);
  emit3(goto_, label1 - pc);
  backpatch(label2, pc - label2);
  // global flag to indicate we're in main()
  is_in_main = 1;
}
```

The code above is implemented for you, but you should study the approach to understand the table hierarchy. The hierarchy consist of a table for global declarations: functions, `main`, and variables (see later), and a table for local variables. Note that we use the `offset` value to index local variable starting at 3 in case of function `main`, because local frame variables 0 to 2 are used in the prologue and epilogue code.

Your task is to add semantic actions to parse local variable declarations and to generate code for expressions and statements on local variables. To do so, you need to add actions to `list` to populate the table of the local scope while parsing the body of the function:

```
%type <typ> type list
list : list ',' ID
    { // add code to enter variable ID in table with type and place
      // for local variables, use offset as place and increment offset
      $$ = $1;
    }
    | type ID
    { // add code to enter variable ID in table with type and place
      // for local variables, use offset as place and increment offset
      $$ = $1;
    }
    ;
```

Note that `type` and `list` have a synthesized type attribute that is passed along the list of identifiers.

To parse a variable in an expression, you need to add code implement:

```
expr : ...
| ID '=' expr
{ int place;
  // check ID is in table for local variables (level=1)
  // add code to get place information for ID
  // if type of ID is integer:
  emit2(istore, place);
  // else if type of ID is float:
  emit2(fstore, place);
}
...
| ID
{ int place;
  // check ID is in table for local variables (level=1)
  // add code to get place information for ID
  // if type of ID is integer:
  emit2(iloop, place);
  // else if type of ID is float:
  emit2(fload, place);
}
...
```

Note that table operations are defined in `table.c` and type checking operations are defined in `type.c`

After completing the changes, your compiler should be able to generate code for programs with a single function `main` and local variables, such as:

```
main()
{ int n, fac;
  n = $0;
  fac = 1;
  while (n > 0)
    fac *= n--;
  return fac;
}
```

However, type checking is not performed yet so we assume that programs with integer variables will work and those with float operations might not.

2 Task 2: Implementing Short-Circuit Evaluation

To implement short-circuit evaluation of boolean expressions, we need to add `truelist` and `falselist` synthesized attributes to `expr`. Therefore, we define in `global.h` a new struct `Expr` for the synthesized attributes of `expr`:

```
struct Expr
{ Backpatchlist *truelist;
  Backpatchlist *falselist;
};
typedef struct Expr Expr;
```

and add it to the Yacc %union:

```
%union
{ ...
  Expr exp;
};
%type <exp> expr
```

You need to implement the `Backpatchlist` type and `backpatch` operations on lists as described in class:

```
Backpatchlist *makelist(int location);
void backpatchlist(Backpatchlist *list, int location);
Backpatchlist *mergelist(Backpatchlist *list1, Backpatchlist *list2);
```

To simplify the process of generating code, we will assume that relational operators and short-circuit operators do not return values 0 and 1, but are Boolean typed. This means that we no longer have to translate relations and the logical operations to integer values. Similarly, we can assume that the `if`, `while`, and `for` constructs only use the backpatch lists to target specific parts of the code and no longer have to check the value on the top of the operand stack.

The `if` construct is implemented as

```
stmt : ...
  | IF '(' expr ')' L stmt
  { // add code to check if $3.truelist and $3.falselist are non-NULL, ...
    // ... otherwise invalid (expr is not short-circuit)
    // when condition is true, jump to stmt:
    backpatchlist($3.truelist, $5);
    // when condition is false, jump to instructions after stmt:
    backpatchlist($3.falselist, pc);
  }
L : { $$ = pc; }
;
```

Implement the backpatch operations in a new file `backpatch.c` and add the function declarations to `global.h`. Also implement the short-circuit code for `&&`, `||`, `!` (not), `if`, `while`, and `for`. For relational operators you should emit `if_icmprel` and `goto` bytecode instructions as explained in class. Also add code to set up the backpatch lists for the relational operations.

Note: we don't use `nextlist` of the textbook's solution.

3 Task 3: Implementing Functions

Take a closer look at the implementation of `main` to familiarize yourself with the code to setup a local scope. We need this approach to implement regular functions by adding actions to:

```
func : ...
  | type ID '(' Margs args ')' block
  { // add code to create function's type with mkfun()
    // add new static method to Class file method array (see below)
    // invoke addwidth, pop tblptr and offset, enter procedure in global table
  }
}
```

```

;
...
Margs :
{ // add code to create new table and push on tblptr and push offset 0
  init_code();
  is_in_main = 0;
}
;

```

Global functions are added as static methods to the Code Class file:

```

cf.methods[cf.method_count].access = ACC_PUBLIC | ACC_STATIC;
cf.methods[cf.method_count].name = ... // name of the function;
cf.methods[cf.method_count].descriptor = ... // type of the function;
cf.methods[cf.method_count].code_length = pc; // the code size
cf.methods[cf.method_count].code = copy_code();
if (!cf.methods[cf.method_count].code)
  error("Out of memory");
cf.methods[cf.method_count].max_stack = 100;
cf.methods[cf.method_count].max_locals = top_offset;
cf.method_count++;
if (cf.method_count > MAXFUN)
  error("Max number of functions exceeded");

```

After adding the code to declare functions, we need to add code to implement function calls:

```

expr : ...
| ID '(' args ')'
{ emit3(invokestatic, constant_pool_add_Methodref(&cf, cf.name, $1->lexptr, gettype(top_tblptr, $1)));
}

```

We also need to make sure return works properly:

```

stmt : ...
| RETURN expr
{ if (is_in_main)
  emit(istore_2);
  else // for now, assume return type is int:
  emit(ireturn);
}

```

At this point, programs with functions should work. For example:

```
int inc(int val)
{ return val + 1;
}
main()
{ int n;
  n = $0;
  return inc(n);
}
```

4 Task 4: Implementing Global Variables

Similar to global functions which are stored as static methods of the class, global variables are static fields of the class. Therefore, to implement access to global variables, we need to access these fields. We use the constant pool index as the place information for a global variable. For declarations, we proceed as follows:

```
list : list ', ' ID
{ // check global level:
  if (top_tblptr->level == 0)
  { cf.fields[cf.field_count].access = ACC_STATIC;
    cf.fields[cf.field_count].name = $3->lexptr;
    cf.fields[cf.field_count].descriptor = $1;
    cf.field_count++;
    enter(top_tblptr, $3, $1, constant_pool_add_Fieldref(&cf, cf.name, $3->lexptr, $1));
  }
  else // local variable declaration
    ...
  $$ = $1;
}
| type ID
  if (top_tblptr->level == 0)
  { cf.fields[cf.field_count].access = ACC_STATIC;
    cf.fields[cf.field_count].name = $2->lexptr;
    cf.fields[cf.field_count].descriptor = $1;
    cf.field_count++;
    enter(top_tblptr, $2, $1, constant_pool_add_Fieldref(&cf, cf.name, $2->lexptr, $1));
  }
  else // local variable declaration
    ...
  $$ = $1;
}
```

Note that we store the constant pool index of the field associated with a global variable as place information in the table.

To implement access to global variables in expressions, we modify `expr` as follows

```
expr : ...
| ID
  { if (getlevel(top_tblptr, $1) == 0)
    { emit3(getstatic, getplace(top_tblptr, $1));
    }
    else // local variable
      ...
  }
```

Add a similar emit statement for `putstatic` in assignments to ID.

At this point the compiler should be able to generate code for programs with global variables, such as:

```
int counter; // initialized to 0 by the JVM
int add(int val)
{ return counter += val;
}
main()
{ add(2);
  return add(-1);
}
```

5 Task 5: Implementing Type Checking

So far we assumed that all variables and function arguments were integer, otherwise our programs won't work because we cannot mix integers and floats.

For the final development phase, you need to add a rudimentary form of type checking. You may assume that 2-ary operations have operands of the same type to avoid type coercions. So operations can only be applied to integer operands or float operands. You don't need to type check parameters to functions, but you need to check the return type of a function.

To add type checking, you need to declare a new synthesized attribute `type` for `expr`:

```
struct Expr
{ Backpatchlist *truelist;
  Backpatchlist *falselist;
  Type type;
};
```

And use this in the `%union` as `Expr expr` and also set `%type <expr> expr`. This attribute is used in checking the types of operands, as in:

```
expr : expr '+' expr
{ if (iseq($1.type, $3.type)
  { if (isint($1.type))
    emit(iadd);
    else
    emit(fadd);
  }
  else
  error("Type error");
  $$ .type = $1.type;
}
```

The only place where we want to coerce a type is in an assignment:

```
expr : ID '=' expr
{ int place = ...
  if (isint(gettype(top_tblptr, $1)))
  { if (isfloat($3.type)
    emit(f2i);
```

```

        if (getlevel(top_tblptr, $1) == 0)
            emit3(putstatic, place);
        else
            emit2(istore, place);
    }
    else if (isfloat(gettype(top_tblptr, $1)))
    { if (isint($3.type)
        emit(i2f);
        if (getlevel(top_tblptr, $1) == 0)
            emit3(putstatic, place);
        else
            emit2(fstore, place);
    }
    else
        error("Type error");
    $$ .type = $3.type;
}

```

This allows us to mix integers and floats in assignments.

A global variable in `mycc.y` is necessary to set the return type of the current function we're generating code for (this could have been done with an inherited attribute if Yacc allowed us to do so) and by introducing a new nonterminal `ftype` for a function's return type:

```

%{
...
static Type return_type;
...
%}
func : ...
    | ftype Margs args ')' block
    { Type type = mkfun($3, return_type);
      ...
    }
...
ftype : type ID '('
    { return_type = $1; //
      $$ = $2;
    }
...
stmt : ...
    | RETURN expr
    { if (is_in_main)
        emit(istore_2);
      else if (isint(return_type) && isint($2.type))
        emit(ireturn);
      else if (isfloat(return_type) && isfloat($2.type))
        emit(freturn);
      else
        error("Type error");
    }
}

```

The type of an expression that is a function call is the return type of the function:

```

expr : ...
    | ID '(' exprs ')'
    { ..
      $$ .type = mkret(gettype(top_tblptr, $1));
    }
}

```

- End