Mini-C to JVM

CS 452

Due May 4, 2007

1 Introduction

For this programming project you will construct a compiler for a simple C-like language we will call *Mini-C*. Your compiler will generate assembler source code that ultimately targets the *Java Virtual Machine* (JVM). You will first convert the grammar described in Section 2.2 into a form usable by YACC. Next you will create the lexical analysis function `yylex()` by either implementing it directly or using LEX. Once you are confident your lexical analyzer and parser are working correctly, you will augment this parser so that it generates a *syntax tree* for each top-level statement and function declaration in the input source program. The lexical analyzer should be retaining line number information, so that syntax errors and static semantic errors are reported (via `yyerror()`) with line numbers indicating the proximity of the offending code. Finally, once you have created the appropriate syntax trees, generate assembler source code as described in Section 3.

2 The Language

2.1 Lexical Elements

Table 2.1 lists the lexical elements of Mini-C. Whitespace is ignored, anything following `\\` to the end of the line is treated as a comment, and keywords and identifiers are case sensitive. If you use the `-d` option, YACC will generate a header file `y.tab.h` containing the appropriate constants for multi-character tokens:

```
yacc -d minic.y
```

This header file will also define `YYSTYPE` which is used to define semantic stack attributes for the various terminals and non-terminals in the language.

2.2 The Grammar

The grammar for Min-C is listed in Figure 1. After either massaging the grammar or using the appropriate YACC directives to enforce the proper operator precedence and associativity, there should only be one shift/reduce error resulting from the dangling-else ambiguity; YACC will shift by default which will match an `else` with the closest `if`. Figure 2 lists a simple Mini-C program that reads an integer from the user and prints out its truncated square root.

3 Target Assembly

There are three directives that specify the Java class name, static fields, and static methods respectively. Figure 3 illustrates each directive. The assembler source code begins with a *class directive*:

```
.class <classname>
```

Static fields are specified with a *field directive*:

```
      program → decl_list
    decl_list → decl_list decl | decl
         decl → var_decl | fun_decl
     var_decl → type_spec IDENT ; | type_spec IDENT [ ] ;
    type_spec → VOID | BOOL | INT | FLOAT
     fun_decl → type_spec IDENT ( params ) compound_stmt
       params → param_list | VOID
   param_list → param_list , param | param
        param → type_spec IDENT | type_spec IDENT [ ]
compound_stmt → { local_decls stmt_list }
  local_decls → local_decls local_decl | ε
   local_decl → type_spec IDENT ; | type_spec IDENT [ ] ;
    stmt_list → stmt_list stmt | ε
         stmt → expr_stmt | compound_stmt | if_stmt | while_stmt |
                return_stmt | break_stmt
    expr_stmt → expr ; | ;
   while_stmt → WHILE ( expr ) stmt
      if_stmt → IF ( expr ) stmt | IF ( expr ) stmt ELSE stmt
  return_stmt → RETURN ; | RETURN expr ;

increasing precedence

         expr → IDENT = expr | IDENT [ expr ] = expr
              → expr OR expr
              → expr AND expr
              → expr EQ expr | expr NE expr
              → expr LE expr | expr < expr | expr GE expr | expr > expr
              → expr + expr | expr – expr
              → expr * expr | expr / expr | expr % expr
              → ! expr | - expr | + expr
              → ( expr )
              → IDENT | IDENT [ expr ] | IDENT ( args ) | IDENT . size
              → BOOL_LIT | INT_LIT | FLOAT_LIT | NEW type_spec [ expr ]
     arg_list → arg_list , expr | expr
         args → arg_list | ε
```

Figure 1: The Mini-C grammar with the dangling-else ambiguity and the usual operator ambiguities. The operators are listed in increasing order of precedence; They are left-associative except for the assignment operator and the unary operators (which are right-associative) and the comparison operators (which are non-associative).

lexeme	*tokens*
(){};,+-*%<>=![]	ASCII value
<= >= == != || &&	LE GE EQ NE OR AND
if else while return break	IF ELSE WHILE RETURN BREAK
new size	NEW SIZE
void bool int float	VOID BOOL INT FLOAT
true false	TRUE FALSE
[0-9]+	INT_LIT
[0-9]+.[0-9]+	FLOAT_LIT
[a-zA-Z_][a-zA-Z_0-9]*	IDENT
EOF	0

Table 1: Lexemes and tokens for Mini-C. The first row lists all of the single character lexemes – their ASCII value is used to encode their token value. The second row lists the two character lexemes and their corresponding tokens (each token in rows 1 through 8 represent some integer constant greater than 256). Following this are keywords for control structures, array operations, primitive types, and boolean literals. Regular expressions are used to describe lexemes for integer literals, floating point literals, and identifiers. "End of file" is encoded with a token value of 0.

descriptor	*type*
V	void (for method return types only)
Z	boolean
I	integer
F	floating point
[Z	array of booleans
[I	array of integer
[F	array of floats

Table 2: Type descriptors.

```
.field <fieldname> <descriptor>
```

The descriptor denotes the field's type as described in Table 2. Static methods are specified with the *method directive*:

```
.method <methodname> (<param-descriptors>)<ret-desciptor> <num-locals>
     ...instruction mnemonics...
.end
```

The parameter descriptors are listed sequentially inside parentheses (e.g., `(IZ[F)` would denote three parameters: an integer, a boolean, and an array of floats). These parameters are indexed on the stack frame beginning with 0. Then number of local variables on the stack frame are then specified; If there are n parameters, then the first local variable is indexed by n. Each line then contains an optional label and on optional instruction mnemonic. *Labels* are alphanumerics strings that represent the address of the current instruction – they begin are column one and end with a colon. The instruction mnemonics are listed in Table 3. Everything after a semicolon on any line denotes a *comments* and is ignored.

3.1 I/O

There are four built-in methods designed for reading input from `stdin` and writing output to `stdout`.

```
int iread(void);
int isqrt(int a, int guess) {  // tail recursive integer square root
  int x;
  if (guess == (x = (guess + a/guess)/2))  // guess via Newton's method
    return guess;
  return isqrt(a, x); // tail recurse
}

int num;

void main(void) {
  num = iread();
  iprint(isqrt(num, num/2));
}
```

Figure 2: Example Mini-C program that reads an integer from `stdin`, stores it in the variable `num`, and then outputs $\lfloor\sqrt{\texttt{num}}\rfloor$.

```
float fread(void);
void iprint(int i);
void fprint(float f);
```

You compiler should enter these method signatures into the symbol table as described in Section **??**.

4 Syntax Trees

The parser generates a *syntax tree* for each function declaration and static variable declaration. Figure 5 shows possible class hierarchies for syntax trees representing statements (which includes static variable declarations) and expressions. The abstract base classes `Stmt` and `Expr` should each specify a virtual method for generating code that is implemented by each specialized subclass. We also need to record each `Expr`'s type so we can perform *static type checking* and coerce integers into floats when necessary.

4.1 Statement and Expression Classes

Create these bases classes is quite natural to do in a language like C++:

```
class Stmt { // abstract base class for statements
public:
  virtual void gencode(FILE *f) const = 0;  // pure virtual method
};

class Expr { // abstract base class for expressions
protected:
  Type_ type_;  // VOID, BOOL, INT, FLOAT, INT_ARRAY, etc...
public:
  Expr(Type_ t) : type_(t) {}
  Type_ getType() const {return type_;}
  virtual void gencode(FILE *f) const = 0;
};
```

A concrete subclass for if-statements might then look like the following:

```
.class Sqrt   ; Java class name

;; method name = "fsqrt"
;; args: float a (local index 0)
;; locals: float xprev (local index 1)
;;         float x (local index 2)
;; returns float sqrt(a)
.method fsqrt (F)F  2   ; name, descriptor, number of local vars
        fload 0
        fstore 1        ; xprev = a
        fconst 0.5
        fload 0
        fmul
        fstore 2        ; x = 0.5*a
top:    fload 1
        fload 2
        fcmp
        ifcmpeq bot     ; if x == xprev we are done
        fload 2
        fstore 1        ; xprev = x
        fload 0
        fload 2
        fdiv
        fload 2
        fadd
        fconst 0.5
        fmul
        fstore 2        ; next guess: x = (x + a/x)*0.5
        goto top
bot:    fload 2
        freturn         ; return x
.end

.field input F    ; static float field, name = "input"

;; "main" method: no args, returns void
.method main ()V
        invokestatic fread  ; read float from stdin
        putstatic input     ; store float in input
        getstatic input
        invokestatic fsqrt
        invokestatic fprint ; print sqrt(num)
        return
.end
```

Figure 3: Hand crafted assembler program that reads a floating point number from `stdin`, and writes its square to `stdout`.

instruction	*operand stack*	*description*
aload <index>	⇒ arrayref	Load local array reference
areturn	arrayref ⇒	Return array reference
arraylength	arrayref ⇒ length	Get length of array
astore <index>	arrayref ⇒	Store local array reference
dup	val ⇒ val, val	Duplicate
dup_x2	a, b, c ⇒ c, a, b, c	Duplicate 2-deep
fadd	a, b ⇒ a + b	Float add
faload	arrayref, index ⇒ value	Load float from array
fastore	arrayref, index, val ⇒	Store into float array
fcmp	a, b ⇒ −1 | 0 | 1	Compare floats $(a < b \mid a = b \mid a > b)$
fconst <float>	⇒ float	Push float constant
fdiv	a, b ⇒ a/b	Float divide
fload <index>	⇒ val	Load local float
fmul	a, b ⇒ a * b	Float multiply
fneg	a ⇒ −a	Float negate
freturn	val ⇒	Return float
fstore <index>	val ⇒	Store local float
fsub	a, b ⇒ a − b	Float subtract
getstatic <var>	⇒ val	Get static var
goto <label>		Branch.
i2f	int ⇒ float	Convert int to float
iadd	a, b ⇒ a + b	Integer add
iand	a, b ⇒ a&b	Bitwise and
iaload	arrayref, index ⇒ value	Load integer from array
iastore	arrayref, index, val ⇒	Store into integer array.
iconst <int>	⇒ int	Push integer constant
idiv	a, b ⇒ a/b	Integer divide
if_cmp<op> <label>	a, b ⇒	Compare and branch if equal, not equal, ...
if<op> <label>	int ⇒	Conditional branch <op>={eq|ne|lt|le|gt|ge}
iload <index>	⇒ val	Load local integer
imul	a, b ⇒ a * b	Integer multiply
ineg	a ⇒ −a	Integer negate
invokestatic <func>	[arg1, [arg2..]] ⇒ [result]	Call function
ior	a, b ⇒ a|b	Bitwise or
irem	a, b ⇒ a%b	Remainder
ireturn	val ⇒	Return integer
istore <index>	val ⇒	Store local integer
isub	a, b ⇒ a − b	Integer subtract
ixor	a, b ⇒ a|b	Bitwise exclusive-or
newarray {int|float}	count ⇒ arrayref	New array of int's or float's
nop		No operation
pop	val ⇒	Pop
pop2	val, val ⇒	Pop twice
putstatic <var>	val ⇒	Put static var
return		Return void.
swap	a, b ⇒ b, a	Swap operand stack values

Table 3: Instruction set. The first column lists the instruction mnemonics in alphabetical order. The second column shows the instruction's effect on the operand stack.

```
.class Sqrt

.method isqrt (II)I 2
        iload 1
        iload 1
        iload 0
        iload 1
        idiv
        iadd
        iconst 2
        idiv
        dup
        istore 2
        isub
        ifeq 001
        iconst 0
        goto 002
001:
        iconst 1
002:
        ifeq 000
        iload 1
        ireturn
000:
        iload 0
        iload 2
        invokestatic isqrt
        ireturn
.end

.field num I

.method main ()V 0
        invokestatic iread
        dup
        putstatic num
        pop
        getstatic num
        getstatic num
        iconst 2
        idiv
        invokestatic isqrt
        invokestatic iprint
.end
```

Figure 4: Output assembly code from my compiler for the listing in Figure 2.

```
class IfStmt : public Stmt { // concrete subclass for if-statements
protected:
  Expr *cond;
  Stmt *thenStmt;
  Stmt *elseStmt;
public:
  IfStmt(Expr *cond_, Stmt *then_, Stmt *else_) :
    cond(cond_), thenStmt(then_), elseStmt(else_) {
    if (cond->getType() != BOOL)
      yyerror("If-else-stmt condition must be of type bool!");
  }
  IfStmt(Expr *cond_, Stmt *then_) :
    cond(cond_), thenStmt(then_), elseStmt(0) {
    if (cond->getType() != BOOL)
      yyerror("If-stmt condition must be of type bool!");
  }
  virtual void gencode(FILE *f) const;
};

void IfStmt::gencode(FILE *f) const {
  Label bot();            // new label to mark bottom of stmt
  cond->gencode(f);       // generate code for condition expr
  if (elseStmt) {         // if there is an "else" part...
    Label mid();          //  ...new label for else code
    emit(f,"ifEq",mid);   //  ...emit branch when bool expr is 0
    thenStmt->gencode(f); //  ...emit code for "then" stmt
    emit(f,"goto",bot);   //  ...emit branch to bottom
    emit(mid);            //  ...emit label for "else" part
    elseStmt->gencode(f); //  ...emit code for "else" stmt
  } else {                // otherwise...
    emit(f,"ifEq",bot);   //  ...emit branch when bool expr is 0
    thenStmt->gencode(f); //  ...emit code for "then" stmt
  }
  emit(bot);              // emit label for bottom
}
```

Although C does not have much support for OOP techniques we can do something similar. Here are analogous base “classes” for statements and expressions:

```
typedef struct Stmt {  /* base class for statements */
  void (*gencode) (struct Stmt *this, FILE *f); /* ptr to code generation method */
} Stmt;

typedef struct Expr {  /* base class for expressions */
  TYPE_ type_;  /* VOID, BOOL, INT, FLOAT, INT_ARRAY, etc... */
  void (*gencode) (struct Expr *this, FILE *f); /* ptr to code generation method */
} Expr;
```

The gencode field is a pointer to the appropriate function that generates code for the instance of a specific subclass. Now we can create a concrete subclass IfStmt by creating a structure whose first field is exactly the same as Stmt’s first field – extended fields follow this:

```
typedef struct {
  void (*gencode) (Stmt *this, FILE *f); /* must be first field! */
  Expr *cond;
  Stmt *thenStmt;
  Stmt *elseStmt; /* may be NULL */
} IfStmt;
```

Now we build a "constructor" for `IfStmt`'s by first defining a specific code generation routine:

```
static void ifStmt_gencode(Stmt *this, FILE *f) {
  IfStmt *s = (IfStmt *) this;  /* cast generic Stmt* to IfStmt* */
  char *bot = newLabel();
  s->cond->gencode(s->cond, f); /* invoke gencode method on s->code */
  if (s->elseStmt != NULL) {
    char *mid = newLabel();
    fprintf(f,"\tifeq %s\n", mid);
    s->thenStmt->gencode(s->thenStmt, f);
    fprintf(f, "\tgoto %s\n", bot);
    fprintf(f, "%s:\n", mid);
    s->elseStmt->gencode(s->elseStmt, f);
  } else {
    fprintf(f,"\tifeq %s\n", bot);
    s->thenStmt->gencode(s->thenStmt, f);
  }
  fprintf(f, "%s:\n", bot);
}
```

Note how we "manually" pass the reference to the specific instance as the first argument to the `gencode` method. Our "constructor" code is as follows:

```
Stmt *createIfStmt(Expr *cond, Stmt *thenStmt, Stmt *elseStmt) {
  IfStmt *s = (IfStmt *) myalloc(sizeof(IfStmt));
  if (cond->type_ != BOOL_)
    yyerror("Condition for if statement must be of type bool!");
  s->gencode = ifStmt_gencode;
  s->cond = cond;
  s->thenStmt = thenStmt;
  s->elseStmt = elseStmt; /* might be NULL if no else-part */
  return (Stmt *) s;  /* cast to base class type */
}
```

The central idea is that we can treat an "object" of type `(IfStmt *)` as an object of type `(Stmt *)`.

5 Syntax Directed Translation

The parser first constructs syntax trees for function definitions and static variable declarations, and then (if all goes well) invokes their respective code generation routines. We can either generate code for each function definition and variable declaration as we encounter them, or place them all in a list and generate code only when we have parsed the entire input file.

As YACC performs its *bottom-up* parse we can associate *attribute information* with each grammar symbol stored on YACC's parser stack. These attributes are created when YACC performs a reduction using some production's right-hand-side (RHS) to match the *handle* on top of the stack. When a reduction is performed, YACC first invokes the production's associated *semantic action* routine.

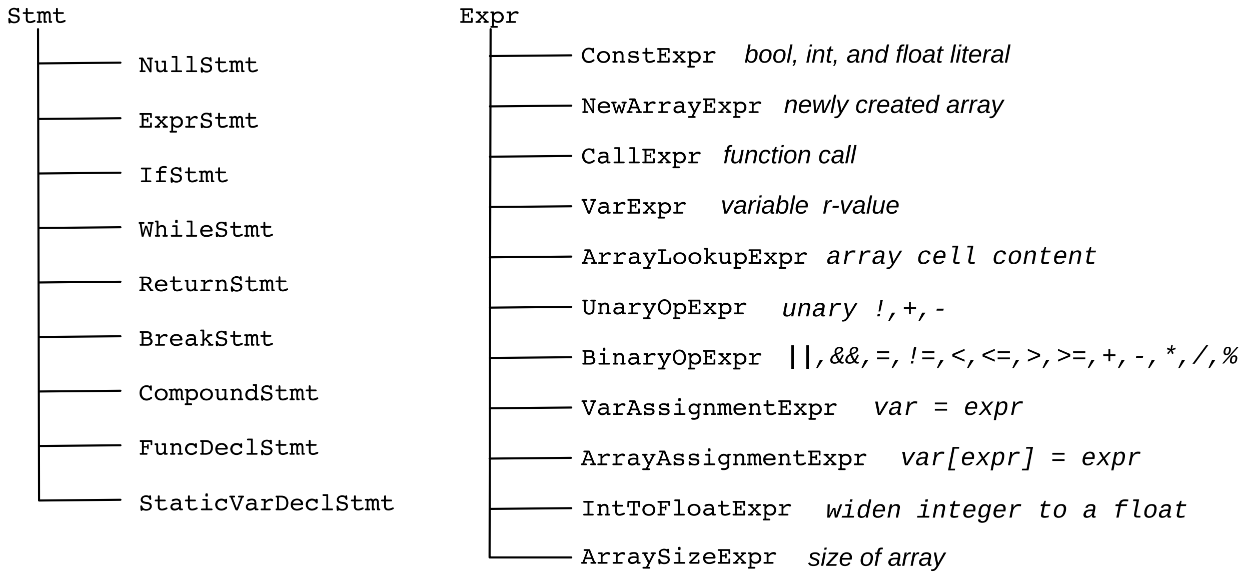
Stmt
NullStmt
ExprStmt
IfStmt
WhileStmt
ReturnStmt
BreakStmt
CompoundStmt
FuncDeclStmt
StaticVarDeclStmt
Expr
ConstExpr bool, int, and float literal
NewArrayExpr newly created array
CallExpr function call
VarExpr variable r-value
ArrayLookupExpr array cell content
UnaryOpExpr unary !,+,-
BinaryOpExpr ||,&&,=,!=,<,<=,>,>=,+,-,*,/,%
VarAssignmentExpr var = expr
ArrayAssignmentExpr var[expr] = expr
IntToFloatExpr widen integer to a float
ArraySizeExpr size of array

Figure 5: Statement and expression syntax tree class hierarchies.

5.1 Attribute types stored on YACC's stack

Most of our semantic actions create a new attribute and store it on the stack. In the following simple example, we store a constant from an enumerated type on the stack which represents one of our primitive types:

```
type_spec     : VOID       {$$ = VOID_;}
              | BOOL       {$$ = BOOL_;}
              | INT        {$$ = INT_;}
              | FLOAT      {$$ = FLOAT_;}
              ;
```

We can then use this and the semantic string attribute associated with the terminal `IDENT` (this string attribute was created by the lexical analyzer) to create variable and array declaration statements

```
var_decl      : type_spec IDENT ';'
                       {$$ = createStaticVarDeclStmt($1, $2);}
              | type_spec IDENT '[' ']' ';'
                       {$$ = createStaticArrayDeclStmt($1, $2);}
              ;
```

Here `$1` and `$2` represent the attributes associated with the first and second symbols (`type_spec` and `IDENT`) on each production's right hand side. We call these *synthesized attributes* since they were "synthesized" when either YACC reduced to `type_spec` or when LEX generated the `IDENT` token. In YACC's preamble, I define the following union type to hold any of the various attributes I will want to store on the stack:

```
%union {
  char *s;
  int i;
  float f;
  Symbol *sym;
  TYPE_ type_;
  Expr *expr;
  List *list;
  Stmt *stmt;
}
```

I then tell YACC which field to associate with each grammar symbol that has an associated attribute. For example, the `s` field points to `IDENT`'s string and the `type_` field holds `type_spec`'s constant:

```
%token <s> IDENT
%type <type_> type_spec
```

5.2 Semantic Actions

Most of the work involves creating the appropriate syntax trees which is triggered by the semantic actions we define. It is usually best to perform *static semantic checks* as soon as possible so that you can report line number information retained by the lexical analyzer. If you wait until all the trees are built to perform these checks, you will not be able to tell the programmer the location of the offending code. [1]

Most semantic actions are quite straight forward. For example, the following actions generate `IfStmt` trees:

```
if_stmt       : IF '(' expr ')' stmt           {$$ = createIfStmt($3,$5,NULL);}
              | IF '(' expr ')' stmt ELSE stmt {$$ = createIfStmt($3,$5,$7);}
              ;
```

Some situations are more subtle. For example, a `break` statement is used to exit the innermost while loop. In order to do this, we need to know the address of the instruction following the while loop. Unfortunately, the action associated with a `break` statement can not obtain this address via synthesized attributes only. To remedy this situation we embed a semantic action in the body of the while-loop that is invoked before the reductions for the trailing non-terminals occur (`expr` and `stmt` in the following):

```
while_stmt    : WHILE {wtop();} '(' expr ')' stmt
                  {$$ = createWhileStmt($4, $6, wbot_label); wbot();}
              ;
```

The function `wtop()` creates a new label that will be used to mark the instruction following the body of the loop. Since while loops can be nested, we need to push and pop this label on and off a stack:

```
void wtop(void) {
  wbot_label = newLabel(); /* create label marking the loop bottom */
  wpush(wbot_label);       /* preserve label on stack */
}
```

Note that `wbot_label` is a global variable that is used by the while-loop constructor. After we are done building the syntax tree, we restore `wbot_label` to its previous state.

```
void wbot(void) {
  wbot_label = wpop();  /* restore wbot_label */
}
```

When we construct a break-statement tree, we first check to see if the stack is empty – if so, then we have an error since the break statement is not nested inside a while loop. Otherwise, we build a tree that will eventually generate code to branch to the address corresponding to `wbot_label`.

[1] Your lexical analyzer could store all of the lexemes and their corresponding line numbers in the symbol table for later error reporting.

6 The Symbol Table

The *symbol table* will be used to store information associated with variable and function names. For variables, we will want to know if the symbol is a global variable (*i.e.,* a static field), a formal parameter, or a local variable. For parameters and local variables we want to know their index or *offset* in the local stack frame. For each function, we want to know the return type, the number of parameters and each of their types, and the number of local variables it used.

The symbol table must also be designed to handle the *static scoping* rules of the language. Each function and each compound statement introduce a new scope wherein new local variables can be declared. This is typically handled by nesting symbol tables and searching for symbols starting with the innermost table. For example, we can define a semantic action that is invoked just as we ender a new scope:

```
compound_stmt : '{' {newScope();} local_decls stmt_list '}'
                    {$$ = createCompoundStmt(symtab, $3, $4); endScope();}
              ;
```

We use a global variable `offset` to index the location of the next parameter or local variable in the stack frame. The `newScope()` function preserves this offset on a stack and creates a new nested symbol table. Another global variable `symtab` is used to reference the current (*i.e.,* innermost) symbol table.

```
void newScope() {
  opush(offset);  /* preserve local variable offset */
  symtab = createSymbolTable(symtab); /* create new nested symbol table */
}
```

When we leave the current scope we restore `offset` and `symtab`.

```
void endScope() {
  symtab = symtab->parent;  /* restore to parent symbol table */
  offset = opop();          /* restore old variable ofset */
}
```

There are three semantic actions associated with the following function declaration:

```
fun_decl : type_spec IDENT '(' {newScope();} params ')'
           {returnType=$1; funcProto($2, $1, $5, symtab->parent);} compound_stmt
           {endScope(); $$ = createFuncDeclStmt($2, $8, symtab);}
         ;
```

We first enter a new scope before the formal parameters are processed (we will also enter a new scope when the function body, which is a `compound_stmt`, is translated). The second set of semantic actions first sets the global variable `returnType` so that we can perform type-checking (and possible integer to float coercion) for return-statements in the function body; Second, the routine `funcProto` enters function prototype information into the root symbol table so that the function can be called recursively in its body. The third and last semantic action ends the parameter scope and builds the appropriate syntax tree for a function declaration. We could avoid "doubly nesting" symbol tables by not reusing `compound_stmt` and creating a similar new non-terminal `body` that is structurally equivalent to `compound_stmt` but did not create a new nested symbol table.

Each function parameter and local variable declaration is processed in the context of the current, most deeply nested, symbol table. For example, here is how I translate local variable declarations:

```
local_decl    : type_spec IDENT ';'         {$$ = localDecl($1,$2);}
              | type_spec IDENT '[' ']' ';' {$$ = localArrayDecl($1,$2);}
              ;
```

The function `localDecl()` makes sure we do not declare multiple variables with the same name, warns the user about "shadowed variables," inserts the appropriate symbol into `symtab` using the current `offset` (which is incremented) and returns this newly inserted symbol.

6.1 I/O Function Symbols

Before parsing starts, we need to enter prototypes for the I/O functions from Section 3.1 into the top-level symbol table so that we can call these functions without the parser complaining.

7 Code Generation Issues

The JVM uses an *operand stack* to hold temporary values and instruction operands. This stack is separate from the stack frame used to hold local variables. Table 3 lists the instructions we will use along with their use of the operand stack. The various "load" instructions push local variables onto the operand stack, while the "store" instructions pop the values into a local variable. Note that `istore` and `fstore` do the exact same thing, but allow the JVM to *verify* that the instructions are "type safe." `putstatic` and `popstatic` move data between the operand stack and "static field" variables.

7.1 Boolean expressions

Besides the type information that is stored as descriptors in the class file, boolean values are simply represented as integers (0 for `false` and 1 for `true`) in the JVM. Since the JVM does not have any comparison routines that push "false" (0) or "true" (1) on the stack, we need generate several lines of code to do this ourselves. For example, the function

```
bool less(int a, int b) {
  return a < b;
}
```

could be translated as follows

```
.method less (II)Z 2
        iload 0
        iload 1
        isub        ; push a - b
        iflt 000    ; if result negative (i.e., a < b) then branch
        iconst 0    ; else push "false"
        goto 001
000:
        iconst 1    ; push "true"
001:
        ireturn
.end
```

Here we used integer subtraction to push either a negative value, zero, or a positive value on the stack. We then branch to the code that pushes either a 0 or a 1 on the stack. A floating point comparison would look the similar, except that we would use `fcmp` in place of `isub` (and the other `i`'s would become `f`'s).

7.2 Assignment as an expression

Assignment statements are really expressions which means we need to leave the result of the assignment on the operand stack. Use the `dup` instruction to make a duplicate of the r-value before storing it. When an expression is treated as a statement, we can discard the value using the `pop` instruction. Consider the following source code which uses a chain of assignments: [2]

[2] In languages like C, `=` is right associative to support assignment chaining, but this is not critical in Mini-C.

```
int x;
void assign1(int y) {
  int z;
  z = x = y = 123;
}
```

Below is the corresponding generated code that use `dup` for each assignment expression; `pop` is used to toss the result when the expression is finally treated as a statement (via `expr_stmt`).

```
.field x I
.method assign1 (I)V 1
        iconst 123      ; r-value = 123
        dup             ; duplicate r-value
        istore 0        ; pop duplicate 123 into y (local 0)
        dup             ; duplicate 123 again
        putstatic x     ; pop duplicate into x (field)
        dup             ; duplicate 123 again
        istore 1        ; pop duplicate into z (local 1)
        pop             ; expression statement removes extra duplicate
.end
```

We use assignment expressions for the side effect of storing a result in a variable. Other expression statements have no net effect as in the following example:

```
void blah(void) {
  3*5+6;
}
```

Here is the resulting generated code:

```
.method blah ()V 0
        iconst 3
        iconst 5
        imul
        iconst 6
        iadd
        pop
.end
```

When assigning values to array elements, we can use the `dup_x2` instruction to place the duplicate below the three operands of the `iastore` instruction:

```
void assign2(int y, int z[]) {
  y = z[5] = 123;
}
```

```
.method assign2 (I[I)V 2
        aload 1    ; push array z reference
        iconst 5   ; push index 5
        iconst 123 ; push r-value 123
        dup_x2     ; dup 123 but place it before the iastore operands
        iastore    ; store 123 at z[5]
        dup        ; dup 123 again
        istore 0   ; store 123 in y (local 0)
        pop        ; remove extra 123 duplicate
.end
```

Of course our code could be optimized to avoid the extra `dup` and `pop` instructions.

8 What to submit

The minimum functioning compiler must be able to handle `bool` and `int` data types (arrays not mandatory).

You will electronically submit an archive file containing the following:

- A `README` File that contains the author's contact information, a brief overview of the project, a description of how to build and run your program, and a list of files contained in the archive.
- All source code.
- Useful test programs.

Your project is due at midnight on the due date.