1. fm language compiler

For this assignment (finally, the last part of the compiler!), you will add code to the parser that you wrote in hw6 so that it generates an actual executable program (in the Postscript language).

The implementation plan is that the code generation is included directly in the parser at the appropriate points as the parser discovers the structure of the program. The generated code is fairly straightforward, since it is mostly expression evaluation and function calls.

2. Grading

The parser code that you wrote for hw6 is the basis for hw7. Since it is possible that your parser is not complete, the grading for homework 7 is a little different from previous assignments.

One half of the score for hw7 will be for just the parser, ie, the same assignment as hw6. So if your hw6 parser still has problems, you can try to finish it up and turn it in for hw7 for half credit. The second half of hw7 credit is the new code generation capability. So if you turn in just a parser for hw7, we will grade that and use that score as half of your hw7 score. If you also attempt to add the code generation capabilities, we will also grade that and use that as the remaining half of your hw7 score.

3. Java classes

java class Flip

This is the main program for the compiler. It is very similar to ParseTest.java from hw6. There are several command line switches that can be used to control various aspects of the output, including method entry/exit tracing, symbol definition, and echoing source lines.

java class `Parser`

This is the only class that needs to be modified for this assignment. I have provided the javadoc for my implementation of the entire compiler, including private methods, so you may want to compare them with your own. There is NO requirement that your implementation match mine, but it might be helpful to review the javadoc if you are trying to work out some details in your parsing.

4. Generating the code to be emitted

The methods in this section generate the code and emit it during the parsing process, using the utility methods defined in the last section. This is relatively straightforward but somewhat tedious. We are implementing the "semantic actions" associated with the syntax when we do this.

a. *in* `public boolean parse()` (provided)

Use **`generateCode`** to write out the Postscript identifier string "%!PS-Adobe-3.0".

b. *in* `private void parseProgram()` (provided)

After matching the Token.ID that is the name of the movie, use generateMovieStart(prevToken.getLabel()) to generate the beginning of file information as described above under generateMovieStart.

After matching the Token.EOF that indicates the end of the source file, use generateMovieEnd() to generate the end of file information as described above under generateMovieEnd.

c. in `private void parseMovieBody()` (provided)

Before parsing anything else, generate the beginning of the prolog. If there is a prolog block, parse that. Generate the end of the prolog. Then parse the page blocks.

d. *in* `parseVariableDeclaration()` (provided in this discussion)

Here we actually generate some real code. The productions associated with this method are

11. *variableDeclaration* → *id* **:** *type***();** | *id* **:** *type***(***exprList***);**

So, your method code has several calls to matchToken as you work your way through the productions. You need to add code to generate a symbol declaration in Postscript. Let's assume your code looks something like mine. Here's what the resulting method looks like (next page).

```
private void parseVariableDeclaration() {
  traceEntry();
  try {
    matchToken(Token.ID);
    Symbol var = new Symbol(prevToken.getLabel(),Symbol.VARIABLE);
    symbolTable.putSymbol(SC_VARIABLE,var.getLabel(),var);
    matchToken(Token.COLON);
    generateCode("/"+var.getLabel());              // ps literal symbol
    matchToken(Token.ID);
    String type = prevToken.getLabel();
    var.putAttribute("CLASSNAME",type);
    matchToken(Token.LPAREN);
    if (theToken.getType()!=Token.RPAREN) {
      parseExprList();
    }
    matchToken(Token.RPAREN);
    generateCode(type+"."+type+" def");   // call constructor and store result
    traceSymbol(var);
    matchToken(Token.SEMICOLON);
  }
  catch (SyntaxException e) {
    processSyntaxException(e);
  }
  traceExit();
}
```

Notice that various actions are taken as we know the information needed.

First of all, once we have decoded the name of the variable, we can generate the beginning of the Postscript definition statement. This has the form "/name value **def**". So when we know the name of the variable, we write out "/name".

The Token.ID that follows the colon is the type of the variable, so we parse that and save the result as an attribute of the Symbol. The attribute is called CLASSNAME, and we store that for later use.

Then we parse the expression list if there is one. The Postscript to implement that expression list is emitted during the parsing, so we don't need to worry about it here.

Finally we match the right paren, and generate the call to the appropriate constructor followed by the Postscript operator **def**. The name of the constructor is the name of the type, followed by a dot, followed by the name of the type again, eg. Integer.Integer. For example, the first variable declaration in StickBoy.fm is:

outline : Box(540,720);

and this generates the Postscript code:

```
/outline
540
720
Box.Box def
```

e. *in* `parsePageBlock()`

The production associated with pageBlock is

13. *pageBlock* → **show (** *integer* **) {** *pageStatements* **}**

The show statement executes all the pageStatements for however many pages are specified. One feature of conventionally structured Postscript files is that each page of the document must stand alone. Consequently, you can't carry a variable value over from page to page. The only thing that you can assume is that the prolog has been executed.

The way I chose for us to implement this is to parse all the pageStatements and emit the code to the buffer, then copy that code over and over for however many pages were specified, with appropriate header and footer code to mark each page.

The number of pages to be generated is given by the integer in parens. So after you parse this integer token, you need to extract the value and store it in a local variable that you can use for loop control later. Something like "int nPages = prevToken.getIntValue();" will do the trick.

After matching the Token.LCURLY, and before parsing the pageStatements, you need to call openBuffer so that the pageStatement code is emitted to the buffer. After parsing the pageStatements with parsePageStatements, and matching the Token.RCURLY, you call closeBuffer.

At this point, you have parsed and compiled all the code in the pageBlock that describes the pages, but you have not written it out to the Postscript output file.

We keep track of the current page number using a Symbol defined and updated by the Parser. This Symbol is named pageNumber, and can be referenced by the fm code just as though it were a user declared variable. The initial value is 0, set in the Parser constructor, but it is updated in our SymbolTable and in the generated Postscript code before each page is written to the output file.

You can get the current value of pageNumber from the SymbolTable like this:

```
Symbol pc = symbolTable.getSymbol(SC_VARIABLE,"pageNumber");
int count = ((Integer)pc.getAttribute("value")).intValue();
```

So we now have all the information needed to write the code for each page of the pageBlock. If showCode is enabled, the method should loop nPages times and generate the following code:

1. The beginning of a page. This code is written with one or more calls to generateCode. The code comprises a DSC comment identifying the page, the Postscript save operator, and a definition of the pageNumber variable. For example, the first page of StickBoy.ps starts with the following code (next page).

```
%%Page: x.1 1
save
/pageNumber 1 def
```

The variable parts are the page number (x.1 and 1) in the %%Page line, and the page number (1) in the pageNumber definition line.

2. Whatever code is in the buffer. This code is copied to output with a call to emitBuffer.
3. The end of a page. This code is written with one or more calls to generateCode. The code comprises the Postscript restore and showpage operators. For example, the first page of StickBoy.ps ends with the following code:

```
restore
showpage
```

The page number that is written out in each case is count+k, where count is the number of pages previously written and k is the page within the current page block. After writing out all the pages in this page block, you need to update the value of pageNumber in the SymbolTable so that it is correct if another pageBlock is compiled.

```
pc.putAttribute("value",new Integer(count+nPages));
```

f. *in* **`parsePageStatement()`**

Here we generate some more executable code. The productions associated with pageStatement are:

15. *pageStatement* →
    { *pageStatements* **}**
    | *methodCall;*
    | *id* **=** *expr***;**
    | **if (***boolExpr***)** *pageStatement*
    | **if (***boolExpr***)** *pageStatement* **else** *pageStatement*

Nothing special needs to be generated for the pageStatements non-terminal.

For the two "if" productions, we need to generate a Postscript "if" or "ifelse" statement. The format of such a statement is "boolean procedure **if**" or "boolean procedure1 procedure2 **ifelse**". Procedures in Postscript are surrounded by curly brackets "{" and "}".

The process for an "if" statement is as follows. After matching Token.KW_IF and Token.LPAREN, we parse the boolExpr. This will emit the code to generate a boolean value. We then use generateCode to write out a left curly "{". Following this, we parse the pageStatement that is executed if the boolean is true using parsePageStatement, and then use generateCode to write out a right curly "}".

If the next token is Token.KW_ELSE, we need to generate procedure2 for the false condition. To do this, we use generateCode to write out a left curly "{", parse the pageStatement that is executed if the boolean is false, and then use generateCode to write out a right curly "}".

Following this the parser uses generateCode to write out the correct operator, either **if** or **ifelse**.

The two remaining productions both start with a Token.ID. This ambiguity could be fixed by a left-factor rewrite of the grammar, but I chose to just do a little ad-hoc fix in the code.

```
case Token.ID: {  // Either a method call or an assignment
   matchToken(Token.ID);
   Token t = prevToken;       // remember the id at the beginning
   if (isFirst(theToken,"callEnd")) {     // start of a method call or ...
      parseCallEnd(t);
      matchToken(Token.SEMICOLON);
   } else {                               // ... an assignment statement
      matchToken(Token.OP_ASSIGN);
      generateCode("/"+t.getLabel());
      parseExpr();
      generateCode("def");
      matchToken(Token.SEMICOLON);
   }
}
```

Notice that the Token describing the id at the beginning of the statement is passed down to parseCallEnd so that the procedure call can be formatted correctly.

The assignment statement is simply another **def** expression starts with /name, followed by the right hand side expression followed by the Postscript **def** operator.

g. *in* `parseExprTail()`

This is where we generate the additive expression operators. The relevant production is

16.2 *exprTail* → + *term exprTail* | - *term exprTail* | ε

The associated code looks something like this:

```
try {
   if (isFirst(theToken,"exprTail")) {
      matchTokenArray((int[])firstSets.get("exprTail"));
      Token opToken = prevToken;
      parseTerm();
      generateCode(operator(opToken));
      parseExprTail();
   }
}
```

Notice that we match the operator token (Token.OP_ADD or Token.OP_SUB) but don't emit the code right away. Postscript is a postfix language, so we parse and emit the term, then emit the

proper operator. The "operator(opToken)" method just returns the correct Postscript operator for the given token, eg, for Token.OP_ADD it returns "add".

h. *in* `parseTermTail()`

This method is very similar to parseExprTail. The relevant production is

17.2 *termTail* → * *factor termTail* | / *factor termTail* | ε

i. *in* `parseFactor()`

Now we are actually emitting the values that form the expressions. The relevant production is

18. *factor* → *integer* | *real* | **(** *expr* **)** | *id* | *methodCall*

For a Token.INTEGER, we just emit the integer value that was parsed. So we have something like:

```
case Token.INTEGER: {
      matchToken(Token.INTEGER);
      generateCode(Integer.toString(prevToken.getIntValue()));
      break;
}
```

Token.REAL is very similar.

For Token.ID, we have to distinguish between a method call and a reference to a variable by itself, just as we did for the assignment statement. Again, I chose to distinguish them in the code, rather than a grammar rewrite. The result can look something like this:

```
case Token.ID: {
    matchToken(Token.ID);
    Token t = prevToken;                    // remember the id at the beginning
    if (isFirst(theToken,"callEnd")) {      // start of a method call or ...
        parseCallEnd(t);
    } else {                                // ... a variable by itself
        Symbol var = symbolTable.getSymbol(SC_VARIABLE,t.getLabel());
        if (var == null) {
            throw new SyntaxException("Variable must be declared before use: "+t.getLabel());
        }
        generateCode(t.getLabel());
    }
    break;
}
```

j. *in* **parseCallEnd(Token t)**

There are two kinds of method calls defined in fm, those that are standard Postscript functions (ie, they do not refer to a particular variable) and those that are instance methods (ie, they refer to a particular variable). The relevant production is

19. *methodCall* → *id*() | *id*(*exprList*) | *id***.***id* **( )** | *id***.***id***(***exprList***)**
19.1 *methodCall* → *id callEnd*

19.2 *callEnd* → () | (*exprList*) | .id() | .id(*exprList*)

Since the initial Token.ID has already been matched before we get to parseCallEnd, that token is passed into the method.

The standard Postscript functions do not use the dot notation and have no notion of an associated variable. Thus the implementation is relatively simple. Match the Token.LPAREN, parse the exprList if any (which will emit the appropriate code for the method arguments), match the Token.RPAREN, and then emit the name of the method to call using generateCode(t.getLabel()). For example, the expression abs(3) is compiled to

```
3
abs
```

The instance methods are a little more complicated because we need to know the variable, the method name, and the variable type in order to generate the call correctly. My implementation of a call to an instance method looks like this:

```
} else if (theToken.getType()==Token.DOT) {  // id.id(), id.id(exprList) or ...
    matchToken(Token.DOT);
    matchToken(Token.ID);
    Token t2 = prevToken;
    matchToken(Token.LPAREN);
    if (theToken.getType()!=Token.RPAREN) {
        parseExprList();
    }
    matchToken(Token.RPAREN);
    Symbol var = symbolTable.getSymbol(SC_VARIABLE,t.getLabel());
    if (var == null) {
        throw new SyntaxException("Variable must be declared before use: "+t.getLabel());
    }
    String type = (String)var.getAttribute("CLASSNAME");
    if (type == null) {
        throw new SyntaxException("Object reference must have defined class type: "+t.getLabel());
    }
    generateCode(t.getLabel());              // the variable (ie, "this")
    generateCode(type+"."+t2.getLabel());    // the class method
} else {                                 // an error
```

For example, the expression outline.draw(30,50); in StickBoy.fm is compiled to the following:

```
30
50
outline
Box.draw
```

k. *in* **`parseBoolExpr()`**

This is pretty simple, since the only actual code here is the **not** operator. If the expression starts with !, then we match Token.OP_NOT, remember the operator, match Token.LPAREN, parse the relExpression (which will emit the code needed to evaluate the relative expression), match Token.RPAREN, and then generateCode(operator(opToken)) which will emit the **not** operator.

If the boolExpr does not start with !, then we just parseRelExpr and let it do the work.

I. *in* `parseRelExpr()`

This is also pretty simple. We just need to move the operator to the end to conform to postfix notation and generate the appropriate Postscript operator.

```
try {
        parseExpr();
        matchTokenArray(new int[] {Token.OP_EQ,Token.OP_GT,Token.OP_LT});
        Token opToken = prevToken;
        parseExpr();
        generateCode(operator(opToken));
}
```

5. Utility methods in Parser.java (all these methods are provided)

Postscript files are organized according to the Document Structuring Conventions (DSC). A simple document starts with several lines of comments. For example, StickBoy.ps starts with:

```
%!PS-Adobe-3.0
% movie StickBoy {
```

The first line identifies the file as Postscript. It is the same for every Postscript file and is written out with a call to **`generateCode`** in the **`parse`** method. The second line is the echoed source code from the file, written out when the source line is read in (assuming that **`CompilerIO setEchoing(true)`** has been called).

The following additions to the Parser class all generate Postscript code one way or another.

a. `private void generateCode(String code)`

If **`setShowCode(true)`** has been called, this method calls the **`CompilerIO emit`** method with the code string it is provided. It is up to the calling routine to define the proper string. There is a discussion later on in this writeup about how to actually create the Postscript code.

b. `private void generateMovieStart(String title)`

```
%%Title: StickBoy
%%Pages: (atend)
%%EndComments
```

These are the third, fourth, and fifth lines at the start of the StickBoy.ps file. They are generated by the **`generateMovieStart`** method and written out with the **`CompilerIO emit`** method (if showCode is enabled). The title string, in this case "StickBoy", is written as part of the title line. The other two lines are the same for every file. There is a discussion later of exactly how to call this method.

c. `private void generateMovieEnd()`

The Postscript file ends with a few DSC comments that wrap it up. For example, StickBoy.ps ends with

```
%%Trailer
%%Pages: 40
%%EOF
```

These lines are generated by **`generateMovieEnd`**. The only variable portion is the number of pages which is tracked throughout the Parser. The technique for doing this is described later.

d. `private void generatePrologStart()`

Every Postscript file that our programs generate will have a "prolog", a section that applies to all pages of the document. The beginning of this section is the same for every document, so we can put the code in a file and copy it to the output object file. If showCode is enabled, then method **`generatePrologStart`** copies the contents of FlipPrologStart.ps to the output file using emitFile.

e. `private void generatePrologEnd()`

Similarly, the end of every prolog is the same for every document, so we put the code in a file and copy it using emitFile if showCode is enabled. The file to copy is FlipPrologEnd.ps.

f. `private String operator(Token t)`

```
/**
* Convert an operator token into the appropriate postscript name
* @param t the Token containing the operator
* @return the equivalent postscript operator
*/
private String operator(Token t) {
      String s = "UNKNOWNOPERATOR";
      if (t == null) return s;
      int op = t.getType();
      if (op == Token.OP_NOT) s = "not";
      else if (op == Token.OP_EQ) s = "eq";
      else if (op == Token.OP_LT) s = "lt";
      else if (op == Token.OP_GT) s = "gt";
      else if (op == Token.OP_ADD) s = "add";
      else if (op == Token.OP_SUB) s = "sub";
      else if (op == Token.OP_MUL) s = "mul";
      else if (op == Token.OP_DIV) s = "div";
      return s;
}
```