DUE: Thursday, December 9, 2004, 11:59 p.m.
fm language compiler

For this assignment (finally, the last part of the compiler!), you will add code to the parser that you wrote in hw7 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.

We have provided you with a solution Parser.java for HW7, so you may use that or your own code as a starting point. However, we have also provided some additional new code in Parser.java that you will find helpful – these are the parts marked (PROVIDED) below. So you may want to either use our Parser.java to start, or else copy the relevant parts to your own code.

java class Flip

This is the main program for the compiler. It is very similar to ParseTest.java from hw7. It expects to be able to call the method setShowCode(boolean b) in the Parser, so you will have to implement at least that one method before the program will compile.

java class `Parser`

This is the only class that needs to be modified for this assignment.

(PROVIDED) You need to add a method to the Parser class named **public void setShowCode(boolean b)** to set a flag that tells the Parser whether or not to generate code. This method works exactly the same way as **setShowMethods(boolean b)** implemented in hw7. It stores the provided boolean value in an instance variable (a variable **showCode** is already defined for you in Parser.java), and then the code generation methods check to see if they should emit code when called.

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

(PROVIDED) `private void generateCode(String code)`

If showCode is enabled, this method calls the 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 what actually needs to be generated.

(PROVIDED) `private void generateMovieStart(String title)`

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

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 (described later). The second line is the echoed source code from the file, written out when the source line is read in. The third, fourth, and fifth lines 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.

(PROVIDED) `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, under parsePageBlock().

`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.

`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.

(PROVIDED) `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;
}
```

Generating the code to be emitted

The only remaining task is to actually generate the code and emit it during the parsing process. This is relatively straightforward but somewhat tedious. We are implementing the "semantic actions" associated with the syntax when we do this.

in `public boolean parse()`

Use generateCode to write out the Postscript identifier string "%!PS-Adobe-3.0" before you do anything else.

in `private void parseProgram()`

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.

in `private void parseProgram()`

You should call generatePrologStart() at the beginning of this method. Then if there is a prolog in the FM you are already calling parsePrologBlock(). After this, call generatePrologEnd(), independent of whether there was a prolog or not, before calling parsePageBlocks().

(PROVIDED) *in* **`parseVariableDeclaration()`**

(This is provided but read carefully as an example)
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. This is an explicit example of the implementation to give you an idea of how this works.

```
// Match name of variable and add to symbol table
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

// Get type of variable and add this as an attribute of the variable
matchToken(Token.ID);
String type = prevToken.getLabel();
var.putAttribute("CLASSNAME",type);

// Get arguments, if any
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);
```

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
```

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 CompilerIO.openBuffer so that the pageStatement code is emitted to the buffer. After finishing with parsePageStatements, 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.

   ```
   %%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. Exception: when writing the first page number in the "%%Page" line (the one following "x."), you should use only the relative page number 'k', 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. The final pageNumber is also the value that is used to calculate the total number of pages in generateMovieEnd().

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

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);
   }
   break;
}
```

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.

in **parseExprTail()**

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

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

If you have a OP_ADD or OP_SUB token here, then you need to *first* output the code for second operand of the sum/difference (by calling parseTerm()), *then* output the appropriate Postscript function call (this operator must be last, since Postscript is a postfix language). You can generate this postscript by calling `generateCode(operator(opToken))` at the appropriate time. Notice that `operator()` requires a Token argument, so you'll need to remember the appropriate Token to use.

in **parseTermTail()**

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

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

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. (the Java class you want is Double)

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:

```
default:   // (this is for a 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;
}
```

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()). You don't have to check that this is a legal postscript method, just pass along whatever was in the input file. 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:

```
default:   // 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
    break;
```

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

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

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.

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, again with then generateCode(operator(opToken))

Debugging

We have provided a number of test files in the fm directory. It's not required that your output exactly match our sample output, but they should be semantically equivalent – e.g. give the same postscript commands in the same order, and your output should be a legal postscript file. You can test this in two ways:

1. Compare against our sample output. You may want to use some program to help the comparison. For instance, download CompareIt from http://www.grigsoft.com/files.htm
2. Using GSView to accept view the output Post script and see if it has the same behavior as the provided files.

If you use the provided starter code the output postscript should match pretty closely. When you compare, remember that comments which are the result of echoing the input files can be ignored.

Turn-in

Just Parser.java.