Overview

Your goal is to complete a very simple compiler outputting JVM bytecode. This assignment is intended as a refresher for recursive-descent parsing, the correspondence between “high-level” and “low-level code,” and basic Haskell skills.

Repository Setup

All provided files can be retrieved from the subversion repository, and your work will be submitted by committing it into the repository. Set up your working copy of the repository as follows:

1. You may work on this assignment alone. You may also work another student as long as you strictly follow the CS 70 pair-programming rules. If you are working with a partner, if you work as a pair, then figure out both your usernames and put them in alphabetical order. In one of your two accounts, go to the cs132s11 directory and (careful with the spelling!) run

   `svn mkdir <first-username>-<second username>`

   (e.g., `svn mkdir asampson-kmarsh`, not `kmarsh-asampson`, even though Marsh comes alphabetically before Sampson).

2. In your directory (or your pair’s directory),

   `svn copy ../src/a1 .`

   to get a copy of the initial files you’ll need.

3. Use `svn commit` in your cs132s11 directory to make these setup steps permanent.

   At this point you should be able to start programming, and committing your changes.
The Microtree Input Language

The “Microtree” language you must compile is described in Figure 1. It’s a very simple imperative language with variables, integers, addition, subtraction, assignments, comparisons, and printing of integers, and conditional statements (if-then-else). For conditionals and loops, 0 is considered false and non-zero numbers are true. (Notice that the loop is a do loop, not a while loop!)

Because the provided lexer is much simpler (hence stupider) than the ones described in class, tokens must be separated by whitespace. In particular, one must write \((3+2) + -1\) rather than \(((3+2) + -1)\). Tokens are represented as strings, so this expression corresponds to the list

\[\text{"(", \\"("}, \text{"3"}, \text{"+"}, \text{"2"}, \text{"\}"}, \text{"+"}, \text{"-"}, \text{"1"}, \text{"\"}}\]

Output Format

Java class files are actually in an unpleasant binary format. What we’d like is an assembler for JVM, but Sun doesn’t provide one. Fortunately, we can use an open-source tool like Jasmin (http://jasmin.sourceforge.net.)

---

1Work only on the assignment when you’re physically together; be logged in only to a single computer, with one keyboard and screen for the two of you; at every point one student is primarily typing and the other is watching and making suggestions; the two roles must be regularly exchanged and shared equally during the assignment.
• The Jasmin distribution can be found in /cs/cs132/jasmin-1.1/ on Knuth, but a copy of the jar file is provided in the repository. To run Jasmin on a file myfile.j then, the command is

```
java -jar jasmin.jar myfile.j
```

This translates your file into a class file that can be executed (via a command like java Main).

• Your program should be translated into a class named Main, containing a single main function. That is, your Jasmin output should be a file with the following form:

```
.class public Main
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit locals Integer number of local variables required
.limit stack Maximum stack depth required
Your code
return
.end method
```

For simplicity, we’ll just assume the stack never grows beyond 99 values.

• The JVM centers on a single stack where most of the operations occur (as is true of PostScript and of RPN calculators). Each routine can have local variables, but they are specified by number (starting at 0), rather than by name.

The following instructions may be useful in your code:

1. iload local-variable-number
   Pushes an integer from the specified variable onto the stack.

2. istore local-variable-number
   Pops an integer from the stack and stores it into the specified variable.

3. ldc integer-constant
   Pushes the given constant onto the stack.

4. dup
   Push a copy of the top stack element onto the stack.

5. swap
   Exchange the top two stack items

6. iadd
   isub
   imul
   idiv
   irem
   Replace the top two values of the stack (which must be integers) by their sum, difference, product, quotient, or remainder (respectively).
7. **label:**
   Target for a jump. The label is an arbitrary identifier and is followed by a colon.

8. **goto label**
   The obvious

9. **ifeq label**
   **ifne label**
   **iflt label**
   **ifgt label**
   **ifle label**
   **ifge label**
   **ifgt label**
   Compares the **top of the stack to zero** and jumps to the given label if the comparison holds. (Consumes that value on the top of the stack in the process.)

10. **return**
    Returns from the function or method.

An integer on top of the stack can be printed using the following sequence of instructions (verbatim):

```
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(I)V
```

### Your Assignment

1. In the file `Parse.hs`, complete the recursive-descent parser. (You should only need to modify `parseE` and `parseS`, and add any relevant helper functions. Your functions do not need to match the structure of the provided grammar exactly, but there should be a close correspondence.

   Note that, because Haskell function have no side-effects, all the parsing functions take in a list of tokens, and return both abstract syntax and a list of leftover tokens. The hardest part of this assignment is getting the plumbing right; feeding the leftover tokens from one subexpression into the parser for the next subexpression.

   You can test your code by using the `ParseTest` program; to compile, run:

   ```
   ghc --make ParseTest
   ```

   and then say `./ParseTest test-input1` or whatever other file you would like to parse.

   You should see the tokens in the given file, and the generated Microtree abstract syntax.
2. In the file ToJVM.hs, complete the translation from Microtree abstract syntax to JVM bytecode instructions. Again, Haskell forces us to thread a “state” value through the translation functions, so be careful with this plumbing as well.

After you compile the code with ghc -o ToJVM you can run the translator and test the results by running

```
./ToJVM test-input1
java -jar jasmin test-input1.j
java Main
```

3. Modify the parser to make parentheses in expressions optional, i.e., allow

```
3 * 2 * ( 1 + 1 )
```

in addition to

```
( 3 * 2 ) * ( 1 + 1 )
```

Multiplication should have the higher precedence than addition, and both operators should be left-associative.

(You should still require parentheses around the expressions if if and do statements, however.)

4. Add a break statement that lets you jump out of the (innermost) loop. This will likely require

- Extending the Microtree abstract syntax;
- Extending the parser to handle break;
- Extending the JVM translation to handle the abstract syntax for a break statement.

(This in turn is likely to require an extra argument to transS, so that you can keep track of the label following the current loop.)

5. Come up with a more interesting test input, and name it interesting-test. (Don’t forget to svn add it to the repository!)

Hints

1. Should you happen to get the error hClose: illegal operation (handle is finalized), this probably signals an infinite loop, e.g., forgetting to remove tokens from the list as you parse.

2. If you’re curious what sort of code javac creates, you can see the bytecodes in any class file (with a somewhat by running

```
/cs/cs132/bin/jad -dis MyClass.class
```
(or whatever the class file is called) on Knuth or Wilkes.

3. The overloaded function `read` can be used to convert strings to integers. You can convert an integer to a string with `show`. You can can test whether a character is a digit with `Char.isDigit`.

4. Do *not* do any arithmetic during translation. Just emit the Java bytecodes to do the arithmetic.

5. Concentrate on correctness over efficiency. In particular, don’t worry about the time needed to append lists (the `++` operator).

6. If you naively make parentheses optional, you get a wildly non-LL(1) grammar. Fortunately, the grammar for parenthesis-optional expressions can be re-written as:

\[
\begin{align*}
\text{<factor>} & ::= \text{<nonnegative-integer-constant>} \\
& \quad \mid \text{<identifier>} \\
& \quad \mid ( \text{<expr>} ) \\
& \quad \mid - \text{<factor>}
\end{align*}
\]

\[
\begin{align*}
\text{<term>} & ::= \text{<factor>} \{ \ast \text{term} \}
\end{align*}
\]

\[
\begin{align*}
\text{<expr>} & ::= \text{<term>} \{ \ast \text{<term>} \}
\end{align*}
\]

where here the curly braces are not concrete syntax, but “extended BNF notation” denoting 0 or more copies of the included syntax. That is, an expr-consuming function would be a loop looking for 1 or more factors separated by "+" tokens, etc. This gives a “flat” representations of sums, which then needs to be converted into a properly-nested tree structure.