Pascal Parser

In this lab, we'll implement a simple parser, which will execute Pascal-like statements as it parses them. In the next lab we'll separate executing from parsing.

Before beginning this lab, make sure that your scanner will correctly scan the test file. Note that strings such as "<=" and ":=" should scan as a single token. Also note that the end of a program is denoted by the symbol '.' and you should return the token "EOF" for a '.' and if an end of file is reached. Also, at the end of input ('.') the scanner `hasNext()` method should return false.

Background: Context-Free Grammars

Before we explore the language we'll be parsing, let's look at a program in another simple language. It appears to count down to zero.

```
loop:
if count = 0 goto end
count := count - 1
goto loop
end:
```

The following is *not* a program in that language, even though it is comprised of the same tokens.

```
:goto 1 0 loop count
if end = - :=
```

To describe a programming language, we need to know more than just what tokens it allows. We need to know its *grammar*—the legal arrangements of those tokens. Specifically, we'll use something called a *context-free grammar* to specify the grammars for our programming languages.

> *program* → *inst*
> *program* → *program inst*
> *inst* → **`id :`**
> *inst* → **`id :=`** *expr op expr*
> *inst* → **`goto id`**
> *inst* → **`if`** *expr* **`=`** *expr* **`goto id`**
> *expr* → **`num`**
> *expr* → **`id`**
> *op* → **`+`**
> *op* → **`-`**

In this notation, tokens like **`:=`**, **`goto`**, and **`if`** are called *terminals*, as are **`num`** (any number token) and **`id`** (any identifier token). These are the symbols that appear in the language. On the other hand, *program*, *inst* ("instruction"), *expr* ("expression"), and *op* ("operation") are called *nonterminals*. They don't actually appear in the program, but instead they name meaningful groups of symbols, much like "prepositional phrase" or "sentence". A context-free grammar consists of a set of rules called *productions*, along with a start symbol. Here, *program* is the start symbol.

Notice that this grammar tells us there are two kinds of operations in the language, four kinds of instructions, etc. Notice also that this grammar is recursive. A *program* might consist of a single instruction, or it might be a whole program followed by one more instruction. In other words, a *program* is a list of one or more instruction.

Notice too that a grammar doesn't tell us how to execute the code. It only tells us what a program looks like—not what it does.

It is often helpful to specify a grammar more compactly as follows, where | means "or".

> *program* → *inst* | *program inst*
> *inst* → **`id :`** | **`id :=`** *expr op expr* | **`goto id`** | **`if`** *expr* **`=`** *expr* **`goto id`**
> *expr* → **`num`** | **`id`**
> *op* → **`+`** | **`-`**

Exercise 1: The Parser Class

We're ready to begin writing our top-down recursive descent parser. Just as our scanner looked ahead one character, our parser will look ahead one token.

Create a package called `parser`. Create a class called `Parser` in the `parser` package. Its constructor should take in a `Scanner` and store it to an instance variable. The constructor should also call the `Scanner`'s `next` method and store the returned `String` in another instance variable which will represent the current token.

Write a private method called `eat`, which should take in a `String` representing the expected token. If the expected token matches the current token, then `eat` should ask the `Scanner` for the next token and store this as the current token. Otherwise, it should throw an `IllegalArgumentException` reporting what token was expected and what token was found. The constructor and `eat` method are the only two places that we will ever directly modify the current token.

Background: Recursive Descent Parsing

Throughout this lab we'll be writing many `parseXXXXX` methods, where `XXXXX` denotes a particular kind of statement, expression, etc., being parsed. For example, consider the `parseIf` method. `parseIf` is called when the current token marks the beginning of an `IF` statement, as shown here.

```
IF n < 0 THEN n := -n; abs := n;
↑
```

When `parseIf` returns, all of the tokens that comprise the `IF` statement have been "eaten", and the current token is now the first one after the `IF` statement.

```
IF n < 0 THEN n := -n; abs := n;
                       ↑
```

We might therefore declare `parseIf` as follows.

```
/**
 * precondition:  current token begins an IF
 * statement postcondition: all tokens in
 * statement have been eaten;
 * current token is first one after the IF statement
 */

private void parseIf()
{
  eat("IF");
  ...
}
```

Similar precondition/postconditions will apply to *all* of the `parseXXXXX` we'll be writing. Notice that we're calling `eat("IF")` to advance to the second token, rather than directly calling the `Scanner`'s `next` method (which we will *never* do in these methods). This way, if the first token turns out not to be `IF`, the parser will halt with a suitable error message.

Exercise 2: Numbers

Write the `private` method `parseNumber` described below.

```
/**
  * Your comments documenting this method according to the
  * style guide
  * precondition:  current token is an integer
  * postcondition: number token has been eaten
  * @return the value of the parsed integer
  */
  private int parseNumber()
```

The following built-in Java method will be helpful in writing `parseNumber`:

```
int num = Integer.parseInt("3");
//num is 3
```

Make sure that `parseNumber` calls the `eat` method appropriately.

Exercise 3: The `WRITELN` Statement

Write the `public` method `parseStatement`. For now, we will have only simple `WRITELN` statements, formatted according to the following production.

stmt → **WRITELN (num) ;**

In addition to eating all tokens associated with the `WRITELN` statement, `parseStatement` should also use `System.out.println` to print the value of the number being printed. It should not return a value. Be sure to make appropriate use of your `parseNumber` method.

Now test your code by parsing the following text, including the period at the end. Your parser should print out the number 3.

```
WRITELN(3);
.
```

Exercise 4: Factors

In this exercise, we'll modify the behavior of the `WRITELN` statement to print out a *factor*, intead of a number. We'll use the term *factor* to refer to any expression we might multiply or divide (although we won't be multiplying or dividing just yet), such as `9`, `-2`, `(2)`, `-(-3)`, etc. Here is our new grammar.

stmt → **`WRITELN`** **`(`** *factor* **`)`** **`;`**
factor → **`(`** *factor* **`)`**
factor → **`-`** *factor*
factor → **`num`**

Write the recursive method `parseFactor`, which should both parse the current factor and return its value. In the following example, `parseFactor` should return positive 3.

```
WRITELN(-(-3));.        WRITELN(-(-3));.
        ↑                             ↑
```

before `parseFactor`   *after* `parseFactor`

Test that your parser can print various factors correctly before moving on.

Exercise 5: Terms

We'll now modify our parser so that the `WRITELN` statement can print out the value of a *term*. A *term* is any expression that might be added or subtracted, such as `16 / (2 * -4)`, as summarized in the following grammar. (`/` performs an integer division, so `7 / 2` should evaluate to 3.) Notice that a *factor* might now be a *term* in parentheses.

stmt → **`WRITELN`** **`(`** *term* **`)`** **`;`**
term → *term* **`*`** *factor* | *term* **`/`** *factor* | *factor*
factor → **`(`** *term* **`)`** | **`-`** *factor* | **`num`**

Although we can understand what this grammar specifies (and that it will follow the correct order of operations), it isn't so easy to parse.

Suppose we naively begin writing `parseTerm` as follows.

```
private int parseTerm()
{
   if ( ??? )
   {
      //term -> term * factor | term / factor
      int val = parseTerm();
      if (token.equals("*"))
      {
         eat("*");
         return val * parseFactor();
      }
      ...
   }
   else
      //term -> factor
      return parseFactor();
}
```

The handling of the rules *term* → *term* ***** *factor* and *term* → *factor* seem reasonable, but how can we determine which of these rules should apply? Look back at the grammar you parsed in the previous exercise. There, we could determine which kind of *factor* we had just by looking at the current token, since each rule began with a different token. In general, if our recursive descent parser need to distinguish between *n* different rules, *n*–1 of them must begin with a different terminal. With *term*, we've got three different rules, and none of them begins with a terminal, so we can't tell from the current token which rule we should apply. Luckily, we can rewrite our grammar in a way that we *can* parse, as shown here. (This is called *left-factoring*. Technically, it's a different grammar describing the same language.)

> *stmt* → **WRITELN (** *term* **) ;**
> *term* → *factor whileterm*
> *whileterm* → ***** *factor whileterm* | **/** *factor whileterm* | ε
> *factor* → **(** *term* **)** | **-** *factor* | **num**

This grammar tells us that every *term* begin with a *factor*, which may be followed by an arbitrary number of multiplications and divisions. Although the new nonterminal *whileterm* is defined recursively, it is best parsed using a while-loop. In other words, parse the first *factor*. Then, while the next token is a `*` or `/`, parse the next *factor*, multiplying and dividing as you go.

Go ahead and write `parseTerm` in this fashion. Be sure to test that each of the following evaluates correctly:

```
WRITELN(6 * 2 / 3);.
WRITELN(6 / 2 * 3);.
WRITELN(6 / (2 * 3));.
```

Exercise 6: Expressions

We're now ready to add addition and subtraction expressions to our grammar as follows.

stmt → **`WRITELN (`** *expr* **`) ;`**
expr → *expr* **`+`** *term* | *expr* **`-`** *term* | *term*
term → *term* **`*`** *factor* | *term* **`/`** *factor* | *factor*
factor → **`(`** *expr* **`)`** | **`-`** *factor* | **`num`**

Go ahead and modify your parser so that it correctly parses and evaluates expressions. As before, you may find it helpful to use left-factoring to rewrite the grammar. Be sure to test that each of the following evaluates correctly:

```
WRITELN(2 + 3 * 4);.
WRITELN(2 * 3 + 4);.
WRITELN((2 + 3) * 4);.
```

Exercise 7: Blocks

Consider the following program.

```
BEGIN
   WRITELN(1);
   WRITELN(2);
   BEGIN
      WRITELN(3);
   END;
END;
.
```

The output of this program is shown below.

```
1
2
3
```

To support programs like this one, we will expand our grammar to include block statements as follows.

> *stmt* → **`WRITELN (`** *expr* **`) ;`** | **`BEGIN`** *stmts* **`END ;`**
> *stmts* → *stmts stmt* | ε
> *expr* → *expr* **`+`** *term* | *expr* **`-`** *term* | *term*
> *term* → *term* **`*`** *factor* | *term* **`/`** *factor* | *factor*
> *factor* → **`(`** *expr* **`)`** | **`-`** *factor* | **`num`**

As before, since these new productions do not lend themselves to recursive descent parsing, it will help to think of them left-factored as follows.

> *stmt* → **`WRITELN (`** *expr* **`) ;`** | **`BEGIN`** *whilebegin*
> *whilebegin* → **`END ;`** | *stmt whilebegin*

Go ahead and modify `parseStatement` so that it parses and executes both `WRITELN` statements and `BEGIN`/`END` block statements.

Exercise 8: Variables

The following program incorporates variables. When run, it prints the number 15.

```
BEGIN
   x := 2;
   y := x + 1;
   x := x + y;
   WRITELN(x * y);
END;
.
```

We only need to make two simple changes to our grammar for it to accomodate the use of variables. We need assignment statements, and we need to allow a variable to be used as a *factor*, as shown below.

> *stmt* → **`WRITELN (`** *expr* **`) ;`** | **`BEGIN`** *stmts* **`END ;`** | **`id :=`** *expr* **`;`**
> *stmts* → *stmts stmt* | ε
> *expr* → *expr* **`+`** *term* | *expr* **`-`** *term* | *term*
> *term* → *term* **`*`** *factor* | *term* **`/`** *factor* | *factor*
> *factor* → **`(`** *expr* **`)`** | **`-`** *factor* | **`num`** | **`id`**

Of course, your program will need to remember which variables have been assigned which values. For this, you'll need to create a new instance variable of type `Map`, whose keys are `String` representing variable names and whose values are `Integers`. Go ahead and make these changes so that you can execute programs with variables.

If You Finish Early

Try adding support for any of the following features. Do *not* try to implement control structures like ifs, while loops, procedures, etc.

- Extend your `Scanner` so that it ignores comments, using the following syntax.

```
WRITELN(1); (*
WRITELN(2);    should only print 1 and 4
WRITELN(3); *)
WRITELN(4);
```

- Add support for performing the mod operation, using the following syntax.

```
WRITELN(9 mod 4); (* prints 1, the remainder when
                     9 is divided by 4 *)
```

- Add support for reading user input, using the following syntax.

```
READLN(x); (* waits for the user to type a number,
              and stores it in the variable x *)
```

- Add support for booleans, using the following syntax.

```
x := TRUE;    (* x is TRUE *)
y := 2 >= 3; (* y is FALSE *)
x := 1 = 2;   (* x is FALSE *)
y := 1 <> 2; (* y is TRUE, like 1 != 2 in Java *)
x := NOT x;
x := x AND (FALSE OR y);
WRITELN(1 = 2); (* prints "FALSE" *)
```

- Add support for strings, using the following syntax.

```
x := 'hi';
y := x, 'ppopotamuses';
WRITELN(3, ' ', y); (* prints "3 hippopotamuses" *)
```

- Add support for arrays, using the following syntax.

```
a := array[1..5];
a[5] := 3;
WRITELN(a[5]); (* prints 3 *)
```