SYSTEMS II — PROJECT 1
Revision 2 [2010-Feb-18]
Compiler parsing and internal representation

1 The goal of this project

You will take possession of a the skeleton of a compiler. Your first task will be to *parse* program text and, whenever doing so successfully, building an *internal representation*. Your skeleton contains all the code and structure needed except for the key methods for parsing and for creating the chain of objects that represent the program parsed. Adding that code will be your task.

2 The grammar

Here is the complete grammar for our langauge. We can make changes, but starting with something complete will help us get started.

```
<program>          -> <declaration list>
<declaration list> -> [ null | <declaration> <declaration list> ]
<declaration>      -> [ <variable> | <procedure> ]
<variable list>    -> [ null | <variable> <variable list> ]
<variable>         -> 'var' <integer> <identifier>
<procedure>        -> 'procedure' <integer> <identifier>
                                  '(' <variable list> ')'
                                  '[' <variable list> ']'
                                  <statement>
<statement list>   -> [ null | <statement> <statement list> ]
<statement>        -> [ 'return' <expression> |
                        <if then else> |
                        <if then> |
                        <while> |
                        <begin end> |
                        <expression> ]
<expression list>  -> [ null | <expression> <expression list> ]
<expression>       -> [ <reference> |
                        <dereference> |
                        <identifier> |
                        <integer> |
                        <procedure call> ]
<reference>        -> '&' <identifier>
<dereference>      -> '*' <expression>
<procedure call>   -> '(' <identifier> <expression list> ')'
<if then>          -> 'if' '(' <expression> ')' <statement>
```

```
<if then else>      -> 'if' '(' <expression> ')' <statement>
                       'else' <statement>
<while>             -> 'while' '(' <expression> ')' <statement>
<begin end>         -> '{' <statement list> '}'
<identifier>        -> [ <alphabetic> | <symbol> ] <alphanumsym list>
<alphabetic>        -> [ 'a' | 'b' | 'c' | ... | 'z' |
                         'A' | 'B' | 'C' | ... | 'Z' ]
<alphanumsym list> -> [ null | <alphanumsym> <alphanumsym list> ]
<alphanumsym>       -> [ <alphabetic> | <dec digit> | <symbol> ]
<symbol>            -> [ '!' | '@' | '#' | '$' | '%' | '^' |
                         '_' | '-' | '+' | '=' | '|' | '\' |
                         ':' | '<' | '>' | '?' | '/' ]
<integer>           -> [ <dec int> | <hex int> | <bin int> ]
<dec int>           -> [ <dec digit> <dec digit list> |
                         '-' <dec digit> <dec digit list> ]
<dec digit list>    -> [ null | <dec digit> <dec digit list> ]
<dec digit>         -> [ '0' | '1' | '2' | ... | '9' ]
<hex int>           -> '0x' <hex digit> <hex digit list>
<hex digit list>    -> [ null | <hex digit> <hex digit list> ]
<hex digit>         -> [ <dec digit> |
                         'A' | 'B' | ... | 'F' |
                         'a' | 'b' | ... | 'f' ]
<bin int>           -> '0b' <bin digit> <bin digit list>
<bin digit list>    -> [ null | <bin digit> <bin digit list> ]
<bin digit>         -> [ '0' | '1' ]
```

3 Internal representation

3.1 Getting the code

I have written a set of Java classes to help you get started with the problem of parsing and internally representing the parsed text in a structured manner. First, to obtain the code for these classes, login to the CS systems and do the following:

```
$ cd cs26
$ tar -xzvpf ~sfkaplan/public/cs26/project-1.tar.gz
$ cd project-1
```

3.2 Understanding this code

Notice that the code you just obtained is collection of classes, many of which are related to one another via *inheritance*.[1] Each of these classes is named and designed such that it corresponds to

[1] If you are not familiar with inheritance, please see me, and I can bring you up-to-date with the aspects of this language feature that you need to know here.

some production rule.

Your primary goal in this assignment is to have each production rule **return an object that contains the information parsed by that rule**, or, if the parsing failed, **return** `null` **to indicate failure**. Thus, these methods that implement production rules will no longer return `true` or `false`. Specifically:

- The lowest-level production rules that parse individual characters (e.g., `<decimal digit>`) should return a pointer to a `Character` object. Note that a `Character` is an object that contains a `char`, but since it's an object, we have the additional capability of returning `null` when none of the desired characters are found during parsing.
- Production rules that return sequences of characters (e.g., `<bin digit list>`) should return a pointer to a `String` object. After all, strings are character sequences.
- All higher-level rules should return a pointer to a **specially designed object** whose purpose is to store the information read by a particular parsing rule. For example, the code that I just provided contains a `Variable` class, where each `Variable` object contains both the name of declared variable and its size (that is, the number of bytes associated with that named space).

If you look in the provided `Parser.java` file, your will see methods for parsing decimal integers. Specifically, notice that the methods follow the pattern described above. Similarly, consider writing methods to parse a variable declaration:

1. Check for the `var` keyword.
2. Call on the method to parse an integer, getting back a pointer to an `Integer` object.
3. Call on the method to parse an identifier, getting back a pointer to an `Identifier` object.
4. Create and return a new `Variable` object, passing its constructor the `Integer` and `Identifier` objects from the previous two steps.

3.3 What you must do

For each production rule, write a method that implements that rule. If the rule succeeds, it should return an object of the proper type. Some of the methods should return standard `Character` or `String` objects; others should return one of the specialized objects provided (e.g., a `ReturnStatement` object).

Ultimately, the top-level `<program>` production rule should return a `List<Declaration>` object.[2] If you try to print this object, it should recursively call the `<toString()>` method in each of the parsed objects, producing a (poorly formatted) representation of what was just parsed. Later, we will add to these classes so that they output appropriate assembly code—for now, we only want to see that we've parsed and internally represented everything correctly.

[2] And if you don't know about Java lists, linked lists, or array lists, let me know so that I can show what these are and how to use them. It's not too complex, I promise.

4 How to submit your work

Use the `cs26-submit` command to turn in your programs. From your `project-1` directory, do this to submit **all** of the classes, including ones written by both you and me:

```
cs26-submit project-1 *.java
```

This assignment is due at **11:59 pm** on **Friday, February 26.**