SYSTEMS II — PROJECT 2
A recursive descent parser

1 The complete grammar

We have, during class, devised something of a language for our own use on our snazzy, new, somewhat buggy AMHCS ISA. Here is the complete, snazzy, new, likely-to-be-somewhat-buggy grammar for our language:

```
<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>          -> [ <expression> |
                          'return' <expression> |
                          <if then> |
                          <if then else> |
                          <while> |
                          '{' <statement list> '}' ]
<expression list>    -> [ null | <expression> <expression list> ]
<expression>         -> [ <identifier> |
                          <integer> |
                          '(' <identifier> <expression list> ')' ]
<if then>            -> 'ifthen' '(' <expression> ')' <statement>
<if then else>       -> 'ifthenelse' '(' <expression> ')'
                          <statement> 'otherwise' <statement>
<while>              -> 'while' '(' <expression> ')' <statement>
<identifier>         -> <alphabetic> <alphanum list>
<alphabetic>         -> [ 'a' | 'b' | 'c' | ... | 'z' |
                          'A' | 'B' | 'C' | ... | 'Z' ]
<alphanum list>      -> [ null | <alphanum> <alphanum list> ]
<alphanum>           -> [ <alphabetic> | <dec digit> ]
<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' ]
<bin int>           -> '0b' <bin digit> <bin digit list>
<bin digit list>    -> [ null | <bin digit> <bin digit list> ]
<bin digit>         -> [ '0' | '1' ]
```

2 Your assignment

Your mission, should you choose to accept it (and you better), is to write a *recursive descent parser*, as discussed in class. That is, this parser should have one method per production rule, evaluating whether the stream of text being processed can be applied to that rule. Recall that the goal of this parser is **only** to determine whether a stream of text correctly corresponds to the grammar; the problem of generating assembly code will come later.

I have begun the parser, which is capable, at the moment, only of parsing `<integer>` and `<identifier>` rules (and those on which they depend). It should be a sufficient example to see how a method can implement one of production rules.

To obtain this sample parser code and get started, login to the CS systems (`castor` or a workstation in SMudd 007), and do the following:

```
$ cd cs26
$ mkdir project-2
$ cd project-2
$ tar -xzvpf ~sfkaplan/public/cs26/project-2-parser.tar.gz
```

The `CharacterStream` class provides a simple interface to the input text, making it easy to read characters from the input and (critically) to reset the input to any point in the stream. This feature is important when one possible production rule is unsuccessfully applied, but another might be the correct one, and needs to re-read the same sequence of characters.

It is the `Parser` class that will require your attention. The `main` method reads the input and attempts to apply the existing rules to it. When your parser is complete, `main` should read the input and apply it only to the top-level production rule, `<program>`.

Note that you must write your own test inputs to determine whether your parser is working. I strongly recommend that you progress iteratively, writing one or two more methods, and then crafting a test input to determine whether it's correct.

3 How to submit your work

Use the `cs26-submit` command to turn in your programs, like this:

```
cs26-submit project-2 Parser.java
```

This assignment is due at **11:59 pm** on **Monday, February 16.**