SYSTEMS II — PROJECT 3
Building a compiler's internal representation

1 The goal of this project

Recall, from Project 2, that we created a parser that was really only a *verifier*—it determined whether a given text input conformed to a particular grammar or not. For this project, we are going to expand on that code, creating a true parser that generates an *internal representation* of the text parsed.

2 Some updates to the grammar

The grammar that you implemented had a few deficiencies. Below is a new grammar (the whole thing), but first, here is a list of the key changes to the Project 2 grammar:

- `<statement>`: A reordering of rules that satisfy this one. In particular, `<expression>` now comes last, ensuring that none of the keywords are, by default, taken as identifiers.
- `<begin end>`: This rule was not explicitly named, but rather appeared in `<statement>` directly. It is now separated like the other statement types (e.g., `<if then>`, `<while>`).
- `<alphanumsym>`: This rule replaces `<alphanumsym>` adding to it the `<symbol>` rule (see below) to allow other desirable naming symbols.
- `<alphanumsym list>`: This rule replaces `<alphanumeric list>` in the obvious manner.
- `<symbol>`: The list of non-alphabetic, non-numeric symbols that can be used for naming.
- `<identifier>`: Now allows an identifier to begin with a `<symbol>` as well as an alphabetic character, and to contain those symbols in subsequent character positions as part of `<alphanumsym>`.
- `<hex digit>`: Now includes lower case `a` through `f` as valid digits for hexidecimal.

Here is the complete grammar:

```
<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> |
                        <if then else> |
                        <while> |
                        <begin end> |
                        <expression> ]
<expression list>  -> [ null | <expression> <expression list> ]
<expression>       -> [ <identifier> |
                        <integer> |
                        <procedure call> ]
<procedure call>   -> '(' <identifier> <expression list> ')'
<if then>          -> 'ifthen' '(' <expression> ')' <statement>
<if then else>     -> 'ifthenelse' '(' <expression> ')'
                        <statement> 'otherwise' <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' ]
```

Other updates: There are also some updates to the `CharacterStream` class. In particular, the `skipWhiteSpace()` method now returns a `true` if one or more whitespace characters were skipped, and `false` otherwise, thus allowing the caller to determine whether whitespace occurred where it should not have or vice versa.

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 internally representing the parsed text in a structured manner. First, to obtain the code for these classes, login to the CS systems (`castor` or a workstation in SMudd 007), and do the following:

```
$ cd cs26
$ tar -xzvpf ~sfkaplan/public/cs26/project-3.tar.gz
$ cd project-3
```

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 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.

[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.

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

Part 1: First, modify your methods that implement production rules so that they return object pointers. As described above, many of the methods should return `Character`, `String`, or an object of one of the provided classes. Copy your methods from your old `Parser.java` file into this new one, updating your code to create and return these objects. Particularly, start with the methods that would return one of the objects of classes that I've already defined for you.

Part 2: I haven't written all of those classes. For example, there are no classes whose objects would contain information about parsed `<if then>` or `<while>` statements. You will need to use existing classes (e.g., `<BeginEnd>`) as examples, and create new classes to complete the collection.[2]

Ultimately, the top-level `<program>` production rule should return a `List<Declaration>` object.[3] 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.

4 How to submit your work

Use the `cs26-submit` command to turn in your programs. From your `project-3` directory, do this to submit **all** of the classes, including ones written by both you and me:

```
cs26-submit project-3 *.java
```

This assignment is due at **11:59 pm** on **Friday, March 13.**

[2]Again, if you are unsure about inheritance, I can help you with writing these classes.

[3]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.