ICSI 311 Assignment 6 – The Parser Part 4

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

We are almost done! We have to write the rest of the statement types and blocks of statements.

In C-like languages, there are two different ways we can write a "block":

```
if (something)
      a=0; // this is a one line block
```
or

```
if (something) {
      // this is a multi-line block
}
```

This same logic is used in a lot of places – if(), while(), do-while(), the AWK actions, functions, etc.

The if is a little bit of an interesting case, because unlike every other instruction, it can be chained. Consider:

```
if (something)
      // case 1
else if (somethingElse)
      // case 2
else if (aThirdThing)
      // case 3
else
      // case 4
```
We will handle this by making the "IfNode" a simple linked list. We could choose to use Java's built-in LinkedList, but it is easier to make our own.

For is a little bit interesting, too. There are two different for nodes in AWK (much like Java):

```
for (int i=0;i<100;i++)     and    for (a in array)
```

I made a key assumption in this – you cannot embed a for loop in the definition of another for loop. Technically, this is not true, but we will assume that it is. Once you do that, you can think of the problem this way – if the IN token appears between the parenthesis, it must be a for-in, otherwise it is a for(;;). This is a bit of an ugly problem – you have to match the parenthesis and do a little bit of look ahead (peek) but it is necessary. A more parsable language design would be use a different key word (like foreach).

Details

ParseBlock needs to handle either the case of a multi-line block or a single line block. This is pretty easy – we can look for an open curly brace token. For single line, it should call a new Function: ParseStatement() and add the result of that to the BlockNode's statement list. In either case, don't forget to AcceptSeperators().

For multi-line, we need to call ParseStatement() repeatedly until it doesn't return a statement, adding the statements to the ParseBlock as we go. Then we need to check for the close curly brace.

ParseStatement() will try to parse each of the statement types, returning the first one that succeeds. In English, this is something like "are you an IF? No, OK, are you a WHILE? No, OK, are you a …"

You will need to make nodes and parsing functions (parseIf, parseFor, etc.) for each statement type:

Continue: takes no parameters, has no additional information.

Break: takes no parameters, has no additional information.

If: has a condition (ParseOperation()) and a statements (ParseBlock()). Needs to handle else if/else, so each IfNode has a "Next" for the linked list of if's. I made this recursive, but it could be done iteratively.

For: has to decide between the two "for" formats. Make a StatementNode for each (ForNode, ForEachNode).

Delete: takes a parameter which is either just a name (delete the whole array) or array reference with a comma separated list of indices. Example: delete a[1,2,3,4]

While: Has a condition (ParseOperation()) and statements, simpler than if

Do-While: Similar to While

Return: takes a single parameter (ParseOperation()).

Finally, a ParseOperation() can be a statement! Consider an assignment statement. But not all return values from ParseOperation are statements. Only a few:

AssignmentNodes, PostDec, PreDec, PostInc, PreInc, FunctionCallNode (see below)

Examples:
a=5, a--, --a, a++, ++a, someFunction(10)

One final thing – we have to parse a function call. Create a new StatementNode (FunctionCallNode) that takes the name of the function to call and the parameters (LinkedList<Node>). A function call requires a name and parenthesis with an optional comma separated list of ParseOperation() inside.

But we aren't going to call ParseFunctionCall() from ParseStatement() – because an assignment can be a function call. Consider:
a=doSomeFunction(1,2,3)

Instead, at the end of ParseBottomLevel(), you were calling ParseLValue(). Right before that, try a ParseFunctionCall() and return its Node if it succeeds.

At this point, we should be able to test with many real AWK programs, so long as you remember to change the regular expression patterns from /pattern/ to `pattern` and add parenthesis to any functions (getline, print, printf, exit, nextfile, next).

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
Unit Tests	Don't exist (0)	At least one (3)	Missing tests (6)	All functionality tested (10)
ParseBlock	Doesn't exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (6)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (10)
ParseStatement	Doesn't exist (0)	Tries most statement types or doesn't always return the first (5)		Tries each statement type and returns the first to succeed (10)
ParseContinue/Break	Doesn't exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (1)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (5)
ParseIf	Doesn't exist (0)	One of: Node data structure correct, parses correctly,	Two of: Node data structure correct, parses correctly,	All of: Node data structure correct, parses correctly,

		throws exceptions with good error messages (3)	throws exceptions with good error messages (6)	throws exceptions with good error messages (10)
ParseFor	Doesn’t exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (6)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (10)
ParseWhile	Doesn’t exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (1)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (5)
ParseDoWhile	Doesn’t exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (1)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (5)
ParseDelete	Doesn’t exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (6)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (10)
ParseReturn	Doesn’t exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (1)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (5)
ParseFunctionCall	Doesn’t exist (0)	One of: Node data structure correct, parses correctly, throws exceptions with good error messages (3)	Two of: Node data structure correct, parses correctly, throws exceptions with good error messages (6)	All of: Node data structure correct, parses correctly, throws exceptions with good error messages (10)