ICSI 311 Assignment 3 – The Parser Part 1

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.

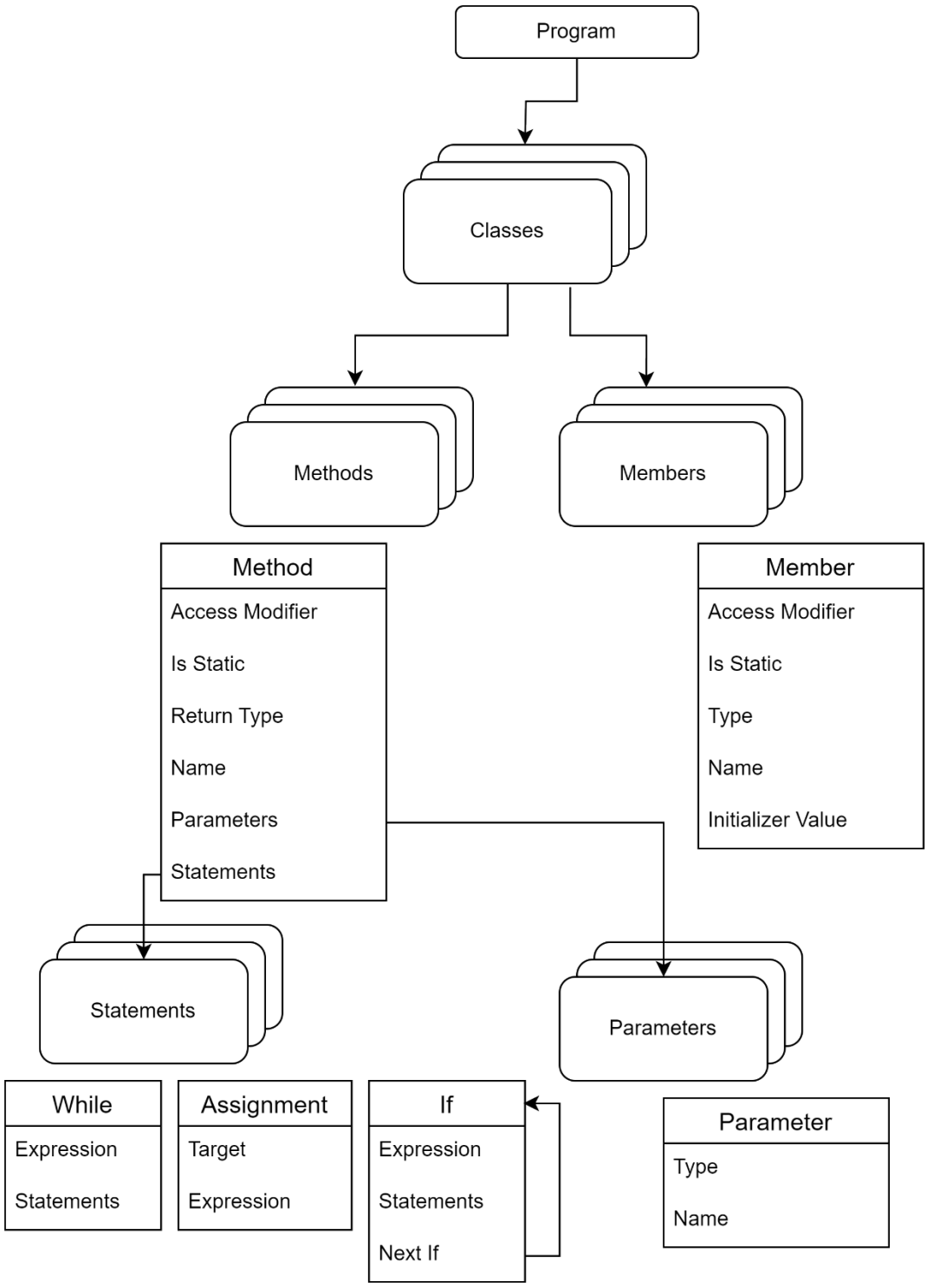
You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

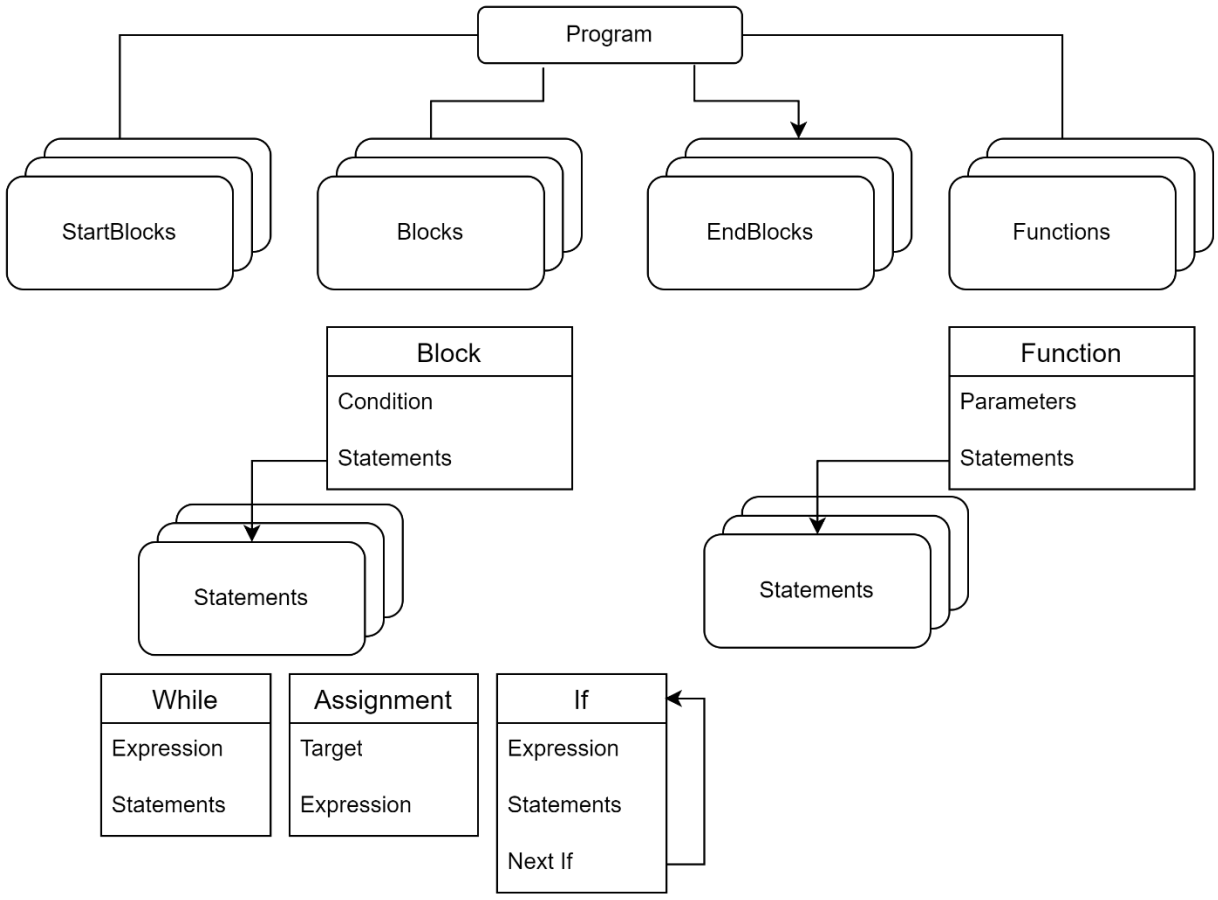
Introduction

In the last two assignments, we transformed a text file into a linked list of tokens. Over the next several assignments, we will transform that list of tokens into a tree. This tree will be a little different, perhaps, than other trees you have seen. It is an n-ary tree with heterogeneous nodes. Consider Java for a moment – a Java program has a root node(Program). A program has one or more classes. A class has zero or more members and zero or more methods. A method has 1 or more statements. See the diagram below.



AWK has a little bit different tree but many similar concepts. AWK is not object oriented, so there are no classes. It was designed for simple scripting so not all code even needs to be in a function! Instead, there are “Blocks”.

An AWK syntax tree has a ProgramNode as the top member. A program consists of a LinkedList of FunctionDefinitionNode and three linked lists of BlockNodes – StartBlocks, EndBlocks and Blocks (neither start nor end).



A FunctionDefinitionNode contains a name, a linked list of parameters and a linked list of StatementNode. Since AWK is dynamically typed and not object oriented, parameters are just names and we don't need to store a return type. For example:

```
function factorial(f) {
       if (f<=1) return 1;
       return f*factorial(f-1);
}
```

This is a good point to mention – the symbol tree is literally the encoding of all the (important) information in a program. If you are not sure what to put in a node, look at what can go into the part of the program.

A BlockNode has two significant components – one is a linked list of StatementNode and the other is a condition – when to run this block. We don't know exactly what type of condition will be in the block (if any). Let's talk about our class hierarchy.

We want an abstract base class (Node). All our nodes will derive from it. Every node **must** have a ToString() method. I made mine formatted to try to look close to what the program itself will look like. That means that when I call ToString() on my ProgramNode, I get output that looks kind of like my input program. This is a very powerful way to visually check your tree.

We will be making a **lot** of classes that derive from Node. Most of them are not very long, though. They have no methods except ToString(), one or more constructors and accessors. These classes are all read-only and should not have mutators, typically.

Finally, we will be using Optional<> a great deal in the parser. This class "wraps" another data type to indicate that there may or may not be one. You can read more about it here:

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Optional.html

Details

Much like we had a StringManager in the Lexer, we want a "TokenManager". This class manages the token "stream" in the same way that the StringManager managed the character "stream". Create this class with a single private member – a linked list of tokens. Make a constructor that accepts that linked list and sets the private member. Then make the following methods:

Optional<Token> Peek(int j) – peek "j" tokens ahead and return the token if we aren't past the end of the token list.

boolean MoreTokens – returns true if the token list is not empty

Optional<Token> MatchAndRemove(TokenType t) – looks at the head of the list. If the token type of the head is the same as what was passed in, remove that token from the list and return it. In all other cases, returns Optional.Empty(). You will use this **extensively.**

Note that there is no accessor for the token list – you must use MatchAndRemove.

Create a Parser class. Much like the Lexer, it has a constructor that accepts a LinkedList of Token and creates a TokenManager that is a private member.

The next thing that we will build is a helper method – boolean AcceptSeperators(). One thing that is always tricky in parsing languages is that people can put empty lines anywhere they want in their code. Since the parser expects specific tokens in specific places, it happens frequently that we want to say, "there HAS to be a ";" or a new line, but there can be more than one". That's what this function does – it accepts any number of separators (newline or semi-colon) and returns true if it finds at least one.

Create a Parse method that returns a ProgramNode. While there are more tokens in the TokenManager, it should loop calling two other methods – ParseFunction() and ParseAction(). If neither one is true, it should throw an exception.

bool ParseFunction(ProgramNode)

bool ParseAction(ProgramNode)

A ProgramNode is the top-most node in our tree. It should have 4 lists: BEGIN BlockNodes, END BlockNodes, other BlockNodes and FunctionDefinitionNodes. A BlockNode is a class that derives from Node. It contains a LinkedList of StatementNode and an Optional<Node> Condition.

In AWK a function call is:

function NAME (parameterList) {

The parameter list could be empty or it could have one or more comma separated names. Remember that AWK is dynamically typed, so there are no types needed in the parameter list.

But there is a tricky bit here – AWK allows newlines after any comma and/or after the close parenthesis. This is why AcceptSeperators() is so convenient. If written correctly (with MatchAndRemove), it will accept the separators without disturbing any other tokens.

Create a new class FunctionDefinitionNode derived from Node. It should hold the function name, the collection of parameter name and a LinkedList of StatementNode. StatementNode is a new abstract class that also derives from Node – it is empty but serves as the base class for all of the statements that we will add later. Don't forget to add a constructor and a ToString to FunctionDefinitionNode.

ParseFunction should return false if this is not a function. If it is a function, it should create the FunctionDefinitionNode, populate it with name and parameters and add it to the ProgramNode's function list. Call ParseBlock() (see below) and add the statements from the BlockNode to the FunctionDefinitionNode's linked list of StatementNode.

Remember that there are 4 different kinds of action:

```
BEGIN { /* do something at the beginning of the program*/}
(a==5) { /* do something for every input line if a = 5 */}
{ /* do something for every input line */}
END { /* do something  at the end of the program*/}
```

Note that the only real difference is the conditions. Parsing conditions is a little tricky – we will do it later. Likewise, dealing with statements is something that we will deal with later. For now, look for BEGIN; if you find it, call ParseBlock() (just returns a new BlockNode, for now). If the block isn't BEGIN, look for END. If it's not one of those, call ParseOperation (returns an empty Optional for now) then calls ParseBlock(). Put the result BlockNode into the appropriate place in the ProgramNode.

```
BlockNode ParseBlock()
Optional<Node> ParseOperation()
```

Testing right now is unusual. We have "stubbed out" (i.e. made functions that just return null/empty) some code. But what can we test? We can certainly write unit tests on our TokenManager. If that doesn't work correctly, tracking down bugs in the Parser will be really hard. We can test AcceptSeperators(), ParseFunction() and ParseAction() to make sure that they follow the patterns they should and insert their nodes into the ProgramNode.

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 (6)	Missing tests (12)	All functionality tested (20)
Token Handler	Doesn't exist (0)	Exists and holds tokens (3)	Exists, member correct, some methods correct, constructor correct (12)	All methods, members, constructor correct (20)
AcceptSeperators	Don't exist (0)	Attempted (3)		Accepts one or more, returns true if any (5)
Parse	Don't exist (0)	Attempted (3)		Creates ProgramNode, loops over Parse calls correctly, throws exception if unknown item found (5)
ParseFunction	Don't exist (0)	One of: Uses MatchAndRemove(), handles any number of parameters, fills in ASTNode correctly (5)	Two of: Uses MatchAndRemove(), handles any number of parameters, fills in ASTNode correctly (10)	Uses MatchAndRemove(), handles any number of parameters, fills in ASTNode correctly (15)
ParseAction	Don't exist (0)	One of: Uses MatchAndRemove(), handles any number of parameters, fills in ASTNode correctly (3)	Two of: Uses MatchAndRemove(), handles any number of parameters, fills in ASTNode correctly (6)	Uses MatchAndRemove(), handles any number of parameters, fills in ASTNode correctly (10)
ProgramNode	Don't exist (0)	Attempted (3)		Members exist, correctly initialized, reasonable ToString() (5)
FunctionDefinitionNode	Don't exist (0)	Attempted (3)		Members exist, correctly initialized, reasonable ToString() (5)

BlockNode	Don't exist (0)	Attempted (3)		Members exist, correctly initialized, reasonable ToString() (5)