## CMPE 152 Programming Project (Part II: Parser)

## Project Introduction

In this project, you will write a parser, based on the lexical analyzer you wrote, for a programming language specified by a given grammar. Your parser should be able to parse the input using the top-down parsing approach and perform basic sematic actions such as enforcing scoping rules, type checking, tree generation, etc.

## Due date

The assignment is due 11:59 pm on Nov 6.

## Language Definition

Our target language (let’s call it mini-lang) to be parsed is defined by the below BNF grammar (terminals or tokens are in **bold**):

*program -> block*

*block ->* ***{*** *decls stmts* ***}***

*decls ->* ***ε*** *| decls decl*

*decl -> type* ***id*** *;*

*type ->* ***int*** *|* ***float*** *|* ***bool***

*stmts ->* ***ε*** *| stmts stmt*

*assign ->* ***id******=*** *allexpr* ***;***

*stmt -> assign |* ***if******(****allexpr****)*** *stmt |* ***if******(****allexpr****)*** *stmt* ***else*** *stmt |* ***while******(****allexpr****)*** *stmt*

*|* ***do*** *stmt* ***while******(****allexpr****)******;*** *|* ***for******(****assign allexpr* ***;*** *incdecexpr****)*** *stmt |* ***break******;*** *| block*

*allexpr -> allexpr || andexpr | andexpr*

*andexpr -> andexpr* ***&&*** *equal | equal*

*equal -> equal* ***==*** *rel | equal* ***!=*** *rel | rel*

*rel -> expr* ***<*** *expr | expr* ***<=*** *expr | expr* ***>*** *expr | expr* ***>=*** *expr | expr*

*expr -> expr* ***+*** *term | expr* ***–*** *term | term*

*term -> term* ***\**** *factor | term* ***/*** *factor | factor*

*incdecexpr ->* ***id++*** *|* ***id--***

*factor ->* ***(****allexpr****)*** *| incdecexpr |* ***id*** *|* ***num*** *|* ***real*** *|* ***true*** *|* ***false***

At the top level, mini-lang starts with a program (non-terminal *program*), which is a block that consists of declarations (non-terminal *decls*) followed by statements (non-terminal *stmts*) enclosed in a pair of curly brackets. Declarations can be empty, or have one or more “*type* ***id****;*” patterns, where type can be one of the three types (*int* for integers, *float* for real numbers, and *bool* for boolean expressions). Supported statement forms include assignment, *if* branch, *do while* loop, *while* loop, *for* loop, and *break*. Statement can also be an entire block.

Expression in mini-lang features a multi-level hierarchical structure. The “bottom level” expression construct is called *factor* (it depends on no other non-terminals, thus has the highest precedence). With *factor* we can construct *term* with \* and / operators. One or more *term* can form *expr* with + and – operators. One or more *expr*, connected with comparison operators (< > <= >=), can form relation expression *rel*, which is further consumed by *equal*’s production rule. From *equal* we can get *andexpr*, and finally the top level, the generic expression *allexpr. allexpr* is used in the statements discussed above.

The terminal **id** in the mini-lang grammar is the string pattern recognized as ID token in your lexical analyzer project. Similarly, **num** terminal corresponds to the NUM token, and **real** terminal corresponds to the REAL token. Reserved keywords such as *if, while, do, for, true, false* should also be recognizable by your lexer that you have already built.

## Scope and Symbol Table

In mini-lang, identifiers have to be declared before usage. The visibility of a particular declaration in mini-lang is determined by scopes (as done by most popular languages), which are associated with blocks. A new scope (i.e., current scope) starts at the beginning of a block, and terminate at the end of the block. Scopes can be nested, as their corresponding blocks. Identifiers declared within the current scope/block override the identifiers with the same name (token value) declared in outer scope(s). In general, upon a usage, an identifier is searched from the current scope first all the way to the outermost scope in order. For example, in the below mini-lang program, the first a = 1.6 is a valid assignment as a uses the float a declaration. T second a = 1.6 is invalid as a uses the bool a declaration declared in the outer scope.

{

bool a;

{

float a;

a = 1.6; // This is fine

}

{

a = 1.6; // This should result in a type error

}

}

You should choose an appropriate data structure to keep track of nested symbol tables. Your symbol table structure can be used by your parser to enforce scope semantic rules. The sample code provides an example symbol table implementation (the Env class in the symbols package).

## Parser Implementation

Based on the non-terminals and the production rules in the grammar, it is often convenient to implement a set of classes that can be used to construct a syntax tree for an input program. In general, we should have a class for each form of production body that represents a critical programming construct (e.g., those generate intermediate code). However, there is not a strict rule or clear one-to-one mapping, i.e., you may have one class to abstract several production bodies (or programming constructs) with similar characteristics, or have more than one class to help constructing one structure. For example, *rel* has multiple production bodies, but the only difference of those bodies are operators, then you may use one class with an operator field to represent all of them. On the other hand, *stmt* has quite different forms of production bodies and thus it is more straightforward to implement each form using a dedicated class (e.g., If, Else, Break, Set, While, Do, Block classes in the inter package of the sample code). Note that you are responsible for adding necessary new classes for the programming constructs in the grammar but not implemented (e.g., For, incdecexpr) in the given sample code. Some constructs such as *decls*, are not even needed to be included in the syntax tree as the declaration information is recorded in the symbol table. All those classes can extend a base class (e.g., Node), which provides common functions such as tracking source code line information, printing errors, etc. You may also add functions into the Node class to make tree traversal and information retrieval easier (e.g., type of node, list of children, etc.).

To implement the main parser logic, predictive parsing is recommended. For each non-terminal in the grammar, write one primary function (with optional helper functions if you would like to) to process the parsing for that non-terminal. The function for a non-terminal shown as a production head needs to invoke functions for the non-terminals in the body of that production rule. If a production head has multiple production bodies to choose from, the look-ahead token can be examined to determine which production body will be used (predictive parsing relies on information about the first symbols/tokens that can be generated by a production body). For a piece of pseudocode for the predictive parsing algorithm, please refer to the lecture slides.

Note that the grammar definition has several productions with left recursion. You may use the left recursion elimination technique to rewrite some of the production rules to facilitate your implementation. Or you can directly implement (without actually rewriting the production rules or introducing new non-terminals) if you know what the resulting string pattern looks like for a production rule with left recursion. For example, the declarations production rule with left recursive, if handled, would result in infinite recursive call with the predictive parsing approach:

*decls -> ε | decls decl*

*decl -> type* ***id*** *;*

Intuitively, we know the above production rules simply result in empty or repeated “*type* ***id***;” patterns. As a result, we can use a loop to retrieve look-ahead tokens and keep invoking the *decl* production as long as the look-ahead tokens match the *decl* pattern.

Based on the above information, implement a Parser class with member functions to handle each and every non-terminals. You may also include helper functions in the parser to process input, print errors, amongst other things you would like to take care of.

You may take the sample source code from Canvas as a reference for your implementation.

## Types in Mini-lang

In mini-lang, any expression has a type (e.g, **id** is one possible form of expression, thus also have a type determined by the preceding type keyword in the declaration). Therefore, your class (and subclasses) that represent any form of expression should keep a field indicating the type for the expression. You can implement this field using a Type class so that you could also store other useful information for the type such as the length in bytes the type takes, the string literal (the reserved keywords) for the type, etc. Alternatively, you can use a string field in your expression class to indicate the type of the expression. In such a case, you may use another table to track metadata for each type. It is up to you which approach you would like to use.

The mini-lang has three basic types: int, float, and bool. Our mini-lang does not do type conversion, i.e, **multiple variables or expressions in any operation have to be of the same type**. Note that bool type variables cannot be used with arithmetic operators such as +, -, \*, and /.

## Error Handling

Your parser should be able to handle two types of errors: type error and grammar error. Below are directions on where and how each type of error can be detected:

* Type Error: When dealing with assignments (*assign*) and expressions (*allexpr, andexpr, … etc.*), your parser should check the type of each subcomponent to determine whether there is a type error. If there is an error, output the relevant error information and line number. For expressions, if your type checking passes, you should determine a type for the resulting expression. For example, when handling E1 = E2 + E3, you find that E2.type and E3.type are valid types to be connected with the + operator, then you need to set E1.type accordingly.
* Grammar Error: During parsing, if the parser is at a point where there is no way to match any production rules, it should output an error message indicating a grammar error.

## Testing & Output

To test your parser, you need to write a driver class with a main function to invoke your parser. Your main function should look like this:

public static void main(String[] args) throws IOException {  
 Lexer lex = new Lexer();  
 Parser parse = new Parser(lex);  
 Prog tree = parse.program();  
 System.out.printf("\nSyntax tree:\n");  
 String treeStr = printTree(tree);  
 System.out.printf(treeStr);  
}

Test your parser with the below input:

{

int r; int dd; int a; int d;

r = a; dd = d;

while( dd <= r ) dd = 2\*dd;

}

The output should be something similar to this:

Syntax tree:

+--PROGRAM

| +--BLOCK

| | +--STMTS

| | | +--ASSIGN

| | | | +--Token r

| | | | +--Token a

| | | +--STMTS

| | | | +--ASSIGN

| | | | | +--Token dd

| | | | | +--Token d

| | | | +--STMTS

| | | | | +--WHILE

| | | | | | +--Token <=

| | | | | | | +--Token dd

| | | | | | | +--Token r

| | | | | | +--ASSIGN

| | | | | | | +--Token dd

| | | | | | | +--Token \*

| | | | | | | | +--Token 2

| | | | | | | | +--Token dd

| | | | | +--

For an invalid input, your program should output error message indicating the type of the error with the line information.

## Usage of Sample Code:

You may refer to and implement your parser based on the sample parser code, which provides examples on how type checking, scoping/symbol table, and parsing can be implemented for a similar grammar. Note that the sample code differs from the requirement of this project in at least the following ways:

1. Certain token definitions (e.g., ID and FOR) are different
2. Types checking requirement is different
3. The self- increment and decrement expressions
4. **For** loop (sample does not implement a **For** tree node or the corresponding parsing logic)

If you use or refer to the sample code, please make necessary changes so that the behavior of your parser conforms to the project requirement.

## Project Submission:

Submit the below files before the project is due:

1. All of your source files
2. Sample test files and its corresponding outputs
3. A readme file that describes:
   1. how to build and run your parser;
   2. features you implemented or modified based on the sample code (symbol table structure, syntax tree, type checking, etc.);
   3. any cool stuff you added or ideas you came up with during the implementation

Organize all of the above files in a folder hierarchy and zip the top level folder as one single zip file named as such: project2-team-name.zip

Submit into Canvas: Project #2. Top-down Parser

## Project Grading:

90% of your score is determined by:

* Whether your parser can correctly detect program with valid vs invalid syntax (30%)
* Type and scope handling (25%)
* Whether your parser has basic error handling (15%)
* Whether your parser can output correct syntax tree (20%)

The rest 10% depends on the impression on the code quality and documentation (e.g., how to run your code, comments and readme).