2018

Compiler Theory and Practice Course Assignment 2017/2018

CPS2000

LUKE ELLUL

Table of Contents

Contents

Task 1 – A table-driven lexer in C++

The lexer is implemented as a class which is stored in the file **Lexer.h** in the directory **Lexer** under the directory **lib**. The class **Lexer** in **Lexer.h** includes the following properties:

- **inputString**: stores the source file (code) as a string of characters
- **currentChar**: stores the current character that is being processed by the lexer. This character is part of the input string.
- **currentPosition**: the current position of the character being processed in the input string
- **end**: signals whether the lexer has processed the end of the inputted string or not
- **ClassifierTable**: a pointer to a C++ map that stores a mapping of characters and their respective Token types. For example, the character '**1**' is mapped to the Token type **DIGIT**
- **TokenTypeTable:** stores a pointer to a C++ array that maps each token type to its respective state in automaton and transition table. For example, state 0 represents a LETTER and state 2 represents a WHITE_SPACE.
- **TransitionTable**: stores a pointer to a 2-dimensional C++ array of token types that represents a transition table for the automaton. Each array in the transition table contains the states that we need to go to from a state given a particular character.

The **lib** directory contains a file called **Token.h**. This file contains an enum called **TokenType** that that includes different type of tokens. Some examples are **LETTER**, **DIGIT**, **WHITE_SPACE**, etc. There are also types for **EXPRESSION**, **STATEMENT**, etc. **Token.h** also includes a struct defining a **Token**. A **Token** has a type, defined as a **TokenType** and a string which stores the actual characters that are represented by that token.

Besides representing different tokens for different character sequences, Token types are also seen as states by the lexer. For example, the token type **LETTER** holds the value of 0 in reality and represents state 0 in the transition table. So, when the lexer processes a character in the alphabet, that character is mapped to the **LETTER** token type and then, the lexer goes to state 0 in the transition table since **LETTER** holds 0.

The first element of the transition table represents the starting state of the automaton. Each element contains an array of token types (states) that the lexer goes to when it receives the next character. This is an array of states of the starting state:

```
/*START */ {
        (TokenType)LETTER,
        (TokenType)INTEGER_LITERAL,
        (TokenType)WHITE_SPACE,
        (TokenType)MULTIPLICATIVE_OP,
        (TokenType)ADDITIVE_OP,
```

```
        (TokenType)RELATIONAL_OP,
        (TokenType)LEFT_BRACKET,
        (TokenType)RIGHT_BRACKET,
        (TokenType)COLON,
        (TokenType)SEMI_COLON,
        (TokenType)INVALID,
        (TokenType)COMMA,
        (TokenType)EQUALS,
        (TokenType)IDENTIFIER,
        (TokenType)RELATIONAL_OP,
        (TokenType)INVERTED_COMMA,
        (TokenType)TAB,
        (TokenType)NEW_LINE,
        (TokenType)INVALID
    },
```

For example, when the next character is a multiplicative operator, the character is mapped to its appropriate token (in this case the **MULTIPLICATIVE_OP**) which is actually the number 3. The lexer than finds the element at index 3 in the starting state array (in this case it's **MULTIPLICATIVE_OP**) which holds 3, therefore we must go to state 3 in the transition table. The same process is repeated for the array at state 3.

The file **TokenTypeTable.h** contains an array of token types representing the final states. Each time the lexer makes a transition to a new state, it checks the array of final states to check if the newly transition state is final or not, so we can know if the sequence should be accepted or not and which token type it should be assigned.

A lexer can be initiated either by providing a string path to where the source code is located or by providing a string of source code, for example an expression. If the lexer receives a file path, it pushes all characters from the source file to the **inputString** property. If it receives a string of source code, the constructor points the **inputString** pointer to the string of source code.

The lexer' s most fundamental method is the **NextToken()** method. **NextToken()** makes use of the transition table to return a sequence of tokens generated from the inputted string of source code. **NextToken()** returns a **Token** that holds the type of the token and a string containing the sequence of characters from the source code that the string represents. Each time **NextToken()** is called the **currentPosition** and **currentChar** properties are updated.

The lexer also has the functionality to recognize special keywords (like **if, while, var,** etc). The **Lexer** directory contains a file called **KeywordTable.h** that includes a C++ map that maps strings of special keywords to their respective token types. For example, the string **"return"** is mapped

to the token type **RETURN**. These special keyword tokens are recognized in the **NextToken()** method. Whenever the final state in the transition table is a **LETTER**, that means that the lexer found a sequence of characters in the alphabet. The lexer checks if this sequence of characters is included as a key in the map located in **KeywordTable.h**. If the map returns a token type, we return this as a special keyword token. If an exception occurs and we don't find a key, this means that the sequence of characters is simply a normal string, so we return the token as a **STRING_LITERAL**.

The algorithm for implementing the **NextToken()** method was inspired from the book **Engineering a Compiler – 2nd Edition** by **K. Cooper, L. Torczon,** page 61 in the **Implementing Scanners** section.

Task 2 – Hand-crafted Recursive Descent Parser in C++

Like the lexer, the parser is also implemented as a class which is defined in **Parser.h** under the **Parser** directory under the **lib** directory. When the parser is initiated it initiates a new lexer and stores a reference to it in the **lexer** property. The parser has a **nextToken()** method that when called, calls the lexer's **nextToken()** method and returns the token returned by the lexer. A reference to this token is stored in the parser's **currentToken** property.

The parser stores tokens as **ASTNodes**. An **ASTNode** is defined as a class and stored in the file **ASTNode.h** located in the **Parser** directory under the **lib** directory. An **ASTNode** class contains two private properties:

- **ASTNodes**: a pointer to a C++ stack that stores ASTNodes that are children of the current ASTNode. For example, an ASTNode representing a **SimpleExpression** contains a **Term**, an **AdditiveOp**, and a **Term** as ASTNode children.
- **token**: a pointer to a token that the ASTNode represents. The token contains a string representing the source code associated with the ASTNode and the token type of the ASTNode. For example, a **SimpleExpression** contains a token with a string like this: **4 + 2**; and **SIMPLE_EXPRESSION** as the token type.

An ASTNode's constructor accepts a token as an argument. The methods **getNodes()** and **getToken()** return pointers to the **ASTNodes** stack and to the **token** respectively.

The **Parser** class contains the following properties:

- **rootNode**: a pointer to the **ASTNode** that represents the root of the tree
- **lexer**: a pointer to the lexer that the parser will use to get tokens from the provided input source code string
- **currentToken**: a pointer to the last token that was returned by the lexer. The current token is also the token that is currently being processed by the parser

The parser class contains several public methods that parse the tokens returned by the lexer and returns the appropriate **ASTNode** for the parsed tokens. There is a method responsible for parsing almost each production rule defined in the MiniLang Assignment Spec EBNF rules. Each method returns a pointer to an **ASTNode** representing a token with the type representing the production rule, the value of the source code associate with the parsed production rule, and the children which represent **ASTNodes** that make up the production rule.

$$\langle SubExpression \rangle \quad ::= \text{`('} \ \langle Expression \rangle \ \text{`)'}$$

For example, the method **ParseSubExpression()** checks if the current token (the last token returned by the lexer) is a left bracket by checking the token type. If it's not, then the parser calls the **Fail()** method with the current token, the token representing the sub expression, and the current node (the node representing the sub expression). The **Fail()** method returns an **ASTNode** with a property, **fail**, set to **true** to signal that there was an error while parsing the current tokens.

If the current token is a left bracket, the left bracket token is converted to an **ASTNode** and is added to the stack of **ASTNode** children of the sub expression. We are done with the bracket so we call **NextToken()** method and update the **currentToken**. We now need to check if the next tokens return an expression so we call the parser's **ParseExpression()** method. Like all other methods, **ParseExpression()** returns an **ASTNode**. We check if the method returned a failed node by checking the **fail** property of the returned node. If it's true, then we return a failed node. If it's false then we add the **ASTNode** returned by **ParseExpression()** to the stack of **ASTNode** children of sub expression. We then check if the **currentToken** is a right bracket. If it is, we convert it to an **ASTNode** and add it to the stack of children and return the **ASTNode** representing the sub expression. If the **currentToken** is not a right bracket we return a failed **ASTNode**.

All the other methods work in a similar fashion. Like the lexer, the parser's constructor can accept a file path to a file containing the source code or a string of source code.

Task 4 – Semantic Analysis Pass

Like the lexer and parser, the semantic analyzer is also defined as a class which is defined in **SemanticAnalyzer.h** under the directory **SemanticAnalyzer** in **lib**. One of the properties of the **SemanticAnalyzer** class is **st** that points to an instance of a **SymbolTable**. The SymbolTable is also defined as a class in **SymbolTable.h** located in the **SemanticAnalyzer** directory under the **lib** directory.

The **SymbolTable** class contains a property called **scopes** that stores a pointer to a C++ stack of maps of identifier strings mapped to their respective token type. Each map represents a scope. For example, the following code:

```
var x : real = 8.1 + 2.2;
```

adds a new key: "x"; to the current scope with token type **real**.

The **SymbolTable** contains the following methods:

- **push()**: pushes a new map (scope) on the stack
- **insert()**: inserts a new entry in the current scope map. The method accepts an identifier token and its token type. It gets the identifier value from the identifier token.
- **update()**: acts like **insert** but instead it updates an already existing key with a new value
- **lookup()**: takes an identifier as an argument and returns its token type. The **lookup** method keeps checking each scope in the **scopes** stack until it finds the identifier. If the identifier is not found, it returns a token type **INVALID**.
- **pop()**: pops a scope from the scopes stack
- **currentStack()**: returns a pointer to the top scope in the **scopes** stack

Through type inference, the Semantic Analyzer can determine the type of an expression and check whether a variable declaration for example is valid. The Semantic Analyzer class contains methods like **AnalyzeExpression()** or **AnalyzeFactor()** that accepts an **ASTNode** and return the type of an expression or a factor as a **TokenType**. For example, **AnalyzeExpression()** takes a pointer an **ASTNode** representing an expression and returns the type (**real, string, int** or **bool**) of that expression.

The **SemanticAnalyzer** class also contains methods like **checkVariableDecl()** that take an **ASTNode** representing a variable declaration and checks whether the type of the expression and the type declared for the variable declaration match. If they match it returns **true**, else it returns **false**.

The method **GetType()** accepts a string and returns its token type. For example, if called with the string "**real**", the method returns **REAL**.

```
var x : real = 8.1 + 2.2;
```

Using the above example, we call the **checkVariableDecl()** method to check if the above variable declaration is correct. The method checks the top **ASTNode** child of the variable declaration's children, which is an expression. Then, it calls the **AnalyzeExpression** method to get the type of the expression and uses the **GetType()** method to get the type that the string **"real"** in the source code represents. If both types match, then the method returns **true**, else it returns **false**.

Besides, type checking and inference, the Semantic Analyzer uses the **SymbolTable** to check if multiple declarations of the same identifiers occurred in the same scope. For example, the **checkVariableDecl()** method checks if the identifier is already defined in the current scope by using the **currentScope()** method explained earlier of the **SymbolTable**. If the identifier is not defined in the current scope, we use the **insert()** method in the **SymbolTable** to insert the identifier and its token type in the current scope.

When doing an assignment, we use the **lookup()** method of the **SymbolTable** to get the type of the already declared identifier and check if the types match.

The **push()** and **pop()** method of the **SymbolTable** are used when the analyzer enters a new block or opts out of a block or checks the actual parameters of a function.

Task 5 – Interpreter Execution Pass

Like the previous sections, the Interpreter is also defined as a class. The **Interpreter** class, which is defined in the class **Interpreter.h** located in the **Execution** directory under the **lib** directory contains two important properties:

- **rf**: stores a pointer to a reference table that is similar to the Symbol Table but instead stores a stack of C++ maps that store a mapping of identifier strings to strings of results that are returned from the execution
- **functionDefs**: stores a pointer to a C++ map that stores a mapping of function identifiers to their function **ASTNodes**

The folder **Reference.h** is also stored under the **Execution** directory under the **lib** directory. The **Reference** table works exactly like the **SymbolTable** and has very similar methods but stores a map of identifiers and their respective results that are returned after the execution. For example, the following variable declaration:

```
var x : real = 8.1 + 2.2;
```

Is stored in the reference table as **"x"** as the key and the string **"10.3"** as the value. Like, the Symbol Table, the Reference table also stores identifiers (keys) in their relative scope.

Like the previous classes, the **Interpreter** class contains methods for executing different production rules. Rules that return a value (like **expressions**), when interpreted their value is returned as a string irrespective of their type. So, the expression: **8.1 + 2.2**; will return a string containing: **"10.3"**; as its value even though it's a number. The reason I took this design decision is, so methods can communicate better with each other if they all use a single type.

To explain how the interpreter works, I'm going to use the above variable declaration as an example. The method **InterpretVariableDecl()** is called which accepts an **ASTNode** representing the variable declaration as its argument and returns a pointer to a string containing the value of the variable (the expression). The method gets the first **ASTNode** at the top of the stack of children of the variable declaration **ASTNode** which contains an expression and calls the method **InterpretExpression** with that expression. The method **InterpretExpression** returns a string with the value of the expression, in this case **"10.3"**. The method then finds the identifier **"x"** in the variable declaration child **ASTNodes** and inserts **"x"** into the **ReferenceTable** mapped to the value **"10.3"**.

To calculate the expression **"8.1 + 2.2"**, the interpreter divides the expression into its children. In this case, we have three **ASTNodes: "8.1", "+",** and **"2.2"**. To convert the strings **"8.2"** and **"2.2"** to numbers the Interpreter uses the C++ function **stod** that accepts a string and returns a double

if that string contains a number. Then, the interpreter checks the operator type by using simple string comparison. In this case, it's **"+"**, so we add the two numbers together and return the calculated value as a string.

Other operations work in a similar fashion. For example, to calculate a relational operator, the interpreter does a simple string comparison to check which relational operator we must use and performs the operation in C++.

Not all methods return a string. For example, methods that interpret while statements and if statements just update the **ReferenceTable** and return nothing.

Now, I'm going to use the following example to explain the interpreter interprets functions and function calls:

```
def func(x : real, y : int) : real
{
    return x * y;
}

func(2.3, 4);
```

When the interpreter encounters a function definition, the method **InterpretFunctionDecl** is used to store a mapping of the function identifier, in this case **"func"** and the **ASTNode** of the function declaration in the C++ map **functionDefs**.

When the interpreter encounters a function call the method **InterpretFunctionCall** is called with the **ASTNode** representing the function call. The method first gets the function identifier stored in the function call **ASTNode**. Then it gets the function definition **ASTNode** associated with the identifier from the **functionDefs** map. Then it enters a new scope by pushing a new scope on the **ReferenceTable** scopes. The method then maps each formal parameter from the function definition (the identifiers) to its actual parameter (values) stored in the actual parameters list in the function call **ASTNode** and stores these values in the **ReferenceTable**. Then the method interprets the statements stored in the function block and returns the value returned by the last return statement.

Task 6 – The REPL

The REPL is also defined as a class in the file **REPL.h** which is located in the **REPL** directory under the **lib** directory. The REPL class contains the following properties:

- **inputString**: stores a pointer to a string that stores the source code that is being processed as a string
- **ans**: stores a pointer to a token that stores the value of the **ans** variable
- **parser**: stores a pointer to a parser
- **analyzer**: stores a pointer to a semantic analyzer
- **interpreter**: stores a pointer to an interpreter

When a REPL is initiated, the REPL constructor initiates an interpreter and a sematic analyzer to be used by the REPL instance. Every input source code string read by the REPL is read by the **ReadLine()** method.

The **ReadLine()** method reads a string of source code as input. If the input contains the substring **"#load"**, then it calls the method **LoadProgram()** with the remaining string (the file location). The **LoadProgram()** method creates a new parser with the location of the source code as input to the parser constructor and parses the program. The **ASTNode** returned by the parser is then checked to see if it is a failed node or not. If it's not a failed node, the ASTNode is fed to the semantic analyzer to check if the source code is semantically correct. If program is semantically correct, the ASTNode is fed to the interpreter and the interpreter updates the reference table accordingly.

Now, if the **ReadLine()** method doesn't find a substring **"#load"**, then it means that the input is a string of source code. The method calls the method **ParseInput()** with the input string source code as an argument. The **ParseInput()** method creates a new parser by calling the parser constructor with the input string and the parser returns an ASTNode. The ASTNode is checked for failure.

If the node failed, then it means the string doesn't contain a statement and probably contains an expression. So, the previously created parser is deleted, and a new parser is created again by calling the parser constructor with the input string. Now, instead of calling the **ParseProgram()** method of the parser, instead we call the **ParseExpression()** method of the parser to parse the expression and the method returns the ASTNode of the expression.

We then check if the expression is semantically correct by calling the **AnalyzeExpression()** method of the semantic analyzer. If the expression is semantically correct, we call the **InterpretExpression()** method of the interpreter and interpret the expression. The string returned by the interpreter is assigned to the **ans** token and displayed to the user.

If the ASTNode checked by the parser didn't fail, then it means that the inputted string is a statement. We call the **checkStatament()** method of the semantic analyzer to check if the statement is semantically correct. If it is, we interpret the statement and assign the string returned by the interpreter to the **ans** token.

It's important to note that the parser is the only instance that is changed with different inputs. The interpreter and analyzer stay the same and are simply updated with new input strings. With each update, the interpreter updates the ReferenceTable and the semantic analyzer updates the SymbolTable.

Installing, Testing and Running

The **build** directory contains the Make files used to build the project. CMake was used to build and compile all the files. I compiled the files into Unix Makefiles so to rebuild the programs, you need to have **Make** installed. On Windows, I used the **make** file that is installed with MinGW.

Under the **build** directory there are two Windows executables: **minilang.exe** and **repl.exe**. To generate Unix executables (you must be running on a Linux OS), simply run the **make** command in the **build** directory. This will generate a new **repl** and **minilang** executables.

It's important to note that you need a C++11 compiler to build the program because I used C++11 features.

The **build** directory contains a directory called **scripts** which contains a number of scripts that I'm going to use as test files. To be recognized by the program, minilang scripts must end with the extension "**.minilang"**.

The first script **s1.minilang** contains functions in the following listing described in the Assignment Spec:

```
def funcSquare(x:real) : real {
        return x*x;
}

def funcGreaterThan(x:real, y:real) : bool {
        var ans : bool = true;
        if (y > x) { set ans = false; }
        return ans;
}

var x : real = 2.4;
var y : real = funcSquare(2.5);
print y;
print funcGreaterThan(x, 2.3);
print funcGreaterThan(x, funcSquare(1.555));
```

The script **s3.minilang** contains the following statements:

```
1 var y : int  = 2;
2 var x : real = 2.0;
3
4 set y = x + 5.8;
```

When we run the script, the program gives a semantic error as expected:

```
PS C:\Users\ellul\Documents\my-shit\school\minilang\build> ./minilang ./scripts/s3.minilang
You have semantic error/s
```

The **minilang** executable loads a scripts given to it as a command line argument and after parsing and checking if it's semantically correct, it interprets it. To interpret the above listing, we issue:

```
PS C:\Users\ellul\Documents\my-shit\school\minilang\build> ./minilang ./scripts/s1.minilang
6.25
true
false
PS C:\Users\ellul\Documents\my-shit\school\minilang\build>
```

The interpret outputs the print statements to the console.

The second script, **s2.minilang**, contains the **funcPow** function also defined in the Assignment Spec. The script adds a print statement to output the result of 2 power 4:

```
PS C:\Users\ellul\Documents\my-shit\school\minilang\build> ./minilang ./scripts/s2.minilang
16
PS C:\Users\ellul\Documents\my-shit\school\minilang\build>
```

And as expected, we get 16 as output.

The **repl** executable when run can start accepting statements or expressions. Expressions don't require a semi-colon at the end. To load a script into the context, issue the command **#load** followed by the location of the script. When a script is loaded, it's functions and statements can be used throughout the whole REPL session. You can also load multiple scripts into the same session. The REPL contains a special variable (which takes any type) called **ans** that stores the result of an expression or the result of a variable declaration or assignment. The variable **ans** can also be used by other statements or expressions.

To start the REPL simply issue the command: **./repl** to run the REPL executable:

```
PS C:\Users\ellul\Documents\my-shit\school\minilang\build> ./repl
MLi> var x : real = 8 + 2;
Ans: 10

MLi> 24 + ans
Ans: 34

MLi> #load "./scripts/s1.minilang"
6.25
true
true
Ans: 34

MLi> funcSquare(ans)
Ans: 1156

MLi> funcGreaterThan(ans, funcSquare(3))
Ans: true

MLi> _
```

In the above demo, we set a variable **x** to 10. As you can see, variable **ans** changes with every returned expression or declaration. Then, we load **s1.minilang** and use its functions in the interpreter.

```
PS C:\Users\ellul\Documents\my-shit\school\minilang\build> ./repl
MLi> var x : real = 8 + 2;
Ans: 10

MLi> 24 + ans
Ans: 34

MLi> #load "./scripts/s1.minilang"
6.25
true
true
Ans: 34

MLi> funcSquare(ans)
Ans: 1156

MLi> funcGreaterThan(ans, funcSquare(3))
Ans: true

MLi> #load "./scripts/s2.minilang"
16
Ans: true

MLi> var y : int = 5;
Ans: 5

MLi> funcPow(funcSquare(y), y)
Ans: 1.38778e+011

MLi>
```

Now, we loaded the script **s2.minilang** and used its **funcPow** function together with the **funcSquare** function from the previous file.

Due to a technical issue, the REPL has functionality to check for semantic errors in its statements but is commented out because the **ans** functionality doesn't work when we check for semantic errors. Basically, **ans** is not given the correct type when it's being updated so whenever the semantic analyzer analyzes the **ans** variable it always gives a semantic error. I know how to fix the problem, but I don't have the time at the moment.

Deviations and Bugs

The compiler follows almost all of the rules defined in the Assignment Spec. The rules that currently don't work include:

- 'and' and 'or' for the multiplicative and additive operators
- The unary operator occasionally works but the "not" doesn't work
- Printable characters may yield an error
- These operators currently don't work: "!=", "<=", and ">="

Also, the interpreter currently doesn't have functionality to check where bugs occurred within the program.