CSCI 468 Compilers
Project Report
Group 7: Ryan Darnell and Stephanie McLaren

Introduction

The purpose of this project is to demonstrate practical knowledge of the java programming language, regular expressions, finite automata, software engineering and design, as well as general problem solving skills in a group dynamic. Throughout this project we have applied knowledge acquired during our undergraduate studies to design, build, test and deploy a working compiler for a LITTLE language.

A working compiler, for even a LITTLE language, is a positive for future endeavors and job searches. It serves to show that we not only understand how to program but that we also understand the foundations of the computer science field. By completing this project, we have gained insight into each of the elements that make up computer science and how they fit together to create a finished product.

In the course of this report we will touch on the background and purpose of a compiler, each of the components that make up a compiler and a summary of what we learned, what we could have done better, and where we will go from here.

Background

In general terms, a compiler takes instructions, written in a specific programming language, and translates them into a machine language also known as an assembly language. This translation is necessary because a computer processor cannot 'read' instructions written in a programming language. The processor needs instructions written in a very basic language. This is why a compiler is necessary for any, non-assembly, programming language.

A compiler is made up of several different sub-routines or steps. The first of those steps is scanning, the second is parsing, the third is creating a symbol table, the fourth is token conversion using a semantic routine and the fifth is optimization which leads to a functional compiler.

A scanner reads character input, in this case a computer program, and converts the incoming characters into tokens that are then read by the parser using a grammar made up of regular expressions.

> The scanner represents an interface between the source program and the syntactic analyzer or parser. The scanner, through a character-by-character examination of the input text, separates the source program into pieces called

tokens which represent the variable names, operators, labels, and so on that comprise the source program[1].

This process is known as a lexical analysis. Should the scanner not recognize the character sequence being read, it will return ‘invalid’ and not convert that sequence into a token. In this case, the input or the grammar will need to be corrected so that each character sequence is converted into a valid token. Breaking the language up into smaller pieces speeds up the parsing step.

The second step is parsing the tokens using the defined grammar for the language. During this step a data structure, usually a parse tree, is created using the input. The parser looks at the token and compares it to the language grammar. If the token is a part of the grammar it is accepted. If it is not a part of the grammar it is rejected. There are two common parsing implementations, top-down and bottom-up parsing.

> Top-Down Parsing: Involves searching a parse tree to find the left most derivations of an input stream by using a top-down expansion. Examples include LL parsers and recursive-descent parsers.
> Bottom-Up Parsing: Involves rewriting the input back to the start symbol. This type of parsing is also known as shift-reduce parsing. One example is a LR parser.[2]

Regardless of the parser type, a parse tree is built to represent the variables and functions being tokenized by the parser. The parse tree is used in the next step.

The third step is creating a symbol table. The parse tree built in step two is traversed and symbol tables are built based on the tokens encountered at each of the nodes. An updated symbol table is built for each new node.

> A symbol table may serve the following purposes depending upon the language in hand:
>
> - To store the names of all entities in a structured form at one place.
> - To verify if a variable has been declared.
> - To implement type checking, by verifying assignments and expressions in the source code are semantically correct.

[1] http://what-when-how.com/compilier-writing/scanners-compiler-writing-part-1/
[2] https://www.techopedia.com/definition/3854/parser

- To determine the scope of a name (scope resolution).

> A symbol table is simply a table which can be either linear or a hash table.[3]

As each new node is encountered an updated symbol table is created and stored. The outer table is known as the global symbol table. The inner tables are scope symbol tables. The global table represents the structure of the program. In other words, it contains global methods and global variables that can be used by any part of the program. The scope symbol tables represent the commands that make up the global methods and local variables. The scope tables can be made up of many sub-tables representing the layers of a method.

The fourth step in building a compiler is analyzing the semantics of the language. The parser created a parse tree by turning blocks of words into tokens. The parser did not check that the blocks were properly structured or ordered. The semantic routines make sure the structure of the program is valid.

In English, verbs, nouns and adjectives are the building blocks of a sentence. Grammar is the order those blocks should have. For example, "Green run fox." is a sentence made up of a noun, a verb, and an adjective but it is not a grammatically correct sentence. The order is wrong and therefore the sentence makes no sense. In the same way semantic routines make sure that the tokens created by the parser are in the proper syntax.

> Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.[4]

If the tokens are not in the proper syntax an error will occur and the token will be rejected. With most compilers a syntax error will result in a failed build and the code will need to be corrected before the program can be successfully compiled.

The final step creates an intermediate code, IR for short, from the original source code. The intermediate code can be either high level or low level. A high level IR code is semantically close to the original code. A low level IR code is closer to the assembly

[3] https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.html

[4] https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.html

code the compiler should generate. Some compilers process the IR code multiple times until the high level IR is turned into a low level IR.

A three address code is created from the IR code. The three address code uses no more than three memory locations to calculate each function. Should all memory locations be full variables that are not currently being used will be stored and their memory location will be reassigned.

Once the three address code is completed an optimization, if desired, can be performed. Optimizing the code is typically done to remove redundant computations and reduce memory slots needed. This results in faster run times with less memory required. The three address code can be read by the computer and the computations written in the original programming language will be carried out. The compiler is functional and complete.

Methods and Discussion

Scanner

To begin, we looked at many options for implementing the given grammar and reading, we decided on a strategy that would simplify development. We determined that Java was the programming language we are the most comfortable with, so we decided to use it. While researching ANTLR, we saw that setting it up in an IDE, Integrated Development Environment, was complex and IDE specific.

Given our familiarity with Linux and OSX we decided to do all of our development in the terminal using a text editor. This made running the code simple with no extra IDE configuration. Given the differences between Windows and Linux shell commands, we decided to use Linux as our default environment for creating, running and testing our bash scripts. So far, no issues have been found when using OSX to run and test our bash scripts. However, if issues arise later in development Linux will become the only operating system used.

After researching how to use Antlr and run the scanner in general, we ran into few difficulties. Some minor grammatical errors and difficulties with regular expressions were the worst issues we ran into getting our grammar set up. At first, getting everything to communicate with Antlr properly caused some minor trouble. Running demonstration files, when they were available, helped us work through those difficulties.

It was the demonstration files and what we learned from them that led us to use Linux and OSX rather than Windows for our development.

Parser

Implementing the parser provided in the ANTLR4 library was a very straight forward process. The Driver class was updated to invoke the ANTLR parser method, add buffer stream to read the input grammar, and add a method to store the parse tree. Having smoothed out the ANTLR4 integration during the first step, the only difficulty we encountered was finding the appropriate method call for the parser.

Using the Linux and OSX environment has posed no difficulties, so we will continue to develop our application using them.

Symbol Table

Several different options were discussed and debated for storing the symbol table information. Early in the design process, stacks and lists were discarded as being difficult to keep track of, especially as they grew. The structure options were narrowed down to either a tree structure or a ‘routing table’ structure. Each option was easy to build and access allowing us to visualize what the tree walker function was actually doing. In theory, this should have made testing and debugging the code an easier process.

In reality, creating the table after walking the parse tree was laborious. Debugging and testing required a lot of thought in order to see which part of the method was failing. To debug the symbol table methods a system out print call was included to print the output of the method. This let us see what each method was sending to the symbol table.

Once we knew what each method was sending to the table we were able to adjust the code to achieve the correct output. This required patience to get the correct output for each method. Once each method worked correctly on it’s own the task of getting the methods to work together began. Starting with two methods the code was adjusted until, as a unit, they presented the correct output, then another method was added and adjusted. Eventually, all of the methods were fine tuned and the output going to the table was correct.

Semantic Routines

This step was the most challenging. Generating the IR code from the parse tree was difficult. Some of the difficulty arose from the design choices we made within the Listener Class.

To reduce time complexity and space needs we opted to use hashmaps to store symbols inside of the symbol tables. This design choice allowed us to keep much of the key functionality for generating the symbol tables inside the Listener class. The drawback to this design choice was building the IR code. The programming code to generate the IR code in our compiler became too complex to leave inside the Listener class.

To account for that we moved created a separate IRBuilder class and altered the methods in the Listener class to call the IRBuilder class. This also caused some of the functionality of the Listener class to become redundant and unnecessary. Rather than having the methods in the Listener class keep track of the method and context of the data that was being sent to the IRBuilder class, we were able to send chunks of data to the IRBuilder class where it could be processed by the appropriate method and returned to the caller.

Both use id_list non-terminal method which also refers to ids containing the functionality of handling SymbolTable behavior along with other things. Trying to add behavior at the enterId() listener would lead to confusing code, be redundant, and be time consuming.

Full-fledged Compiler

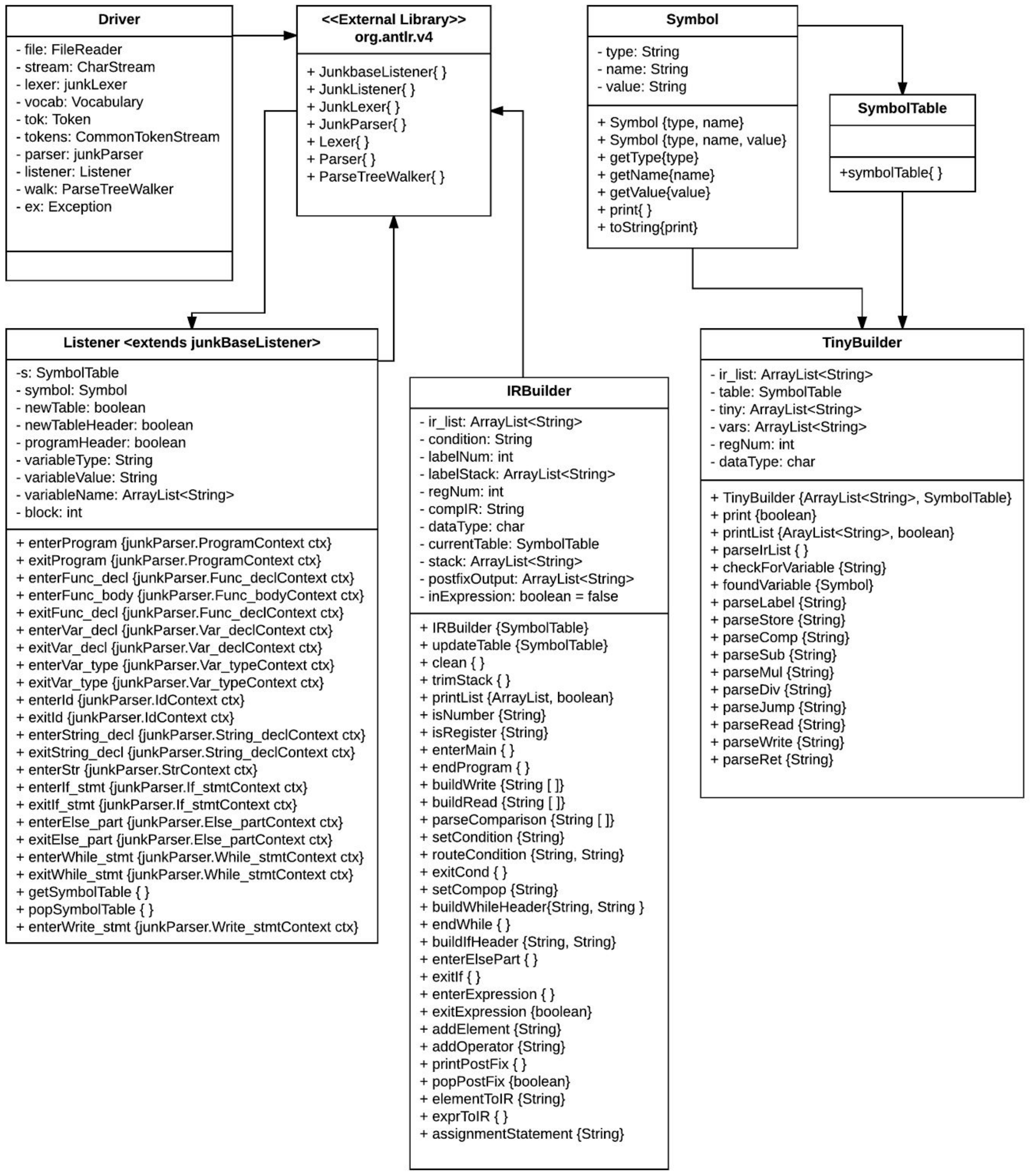
After working through generating the IR code we were able to focus on converting it to the assembly language we wanted. We decided to use a stack and an array to hold the IR code that was being passed in. This proved to be difficult to implement.

To account for order of precedence when dealing with add, subtract and multiply commands we needed to be able to pop items off of the stack and into the arraylist in the proper order. The stack design does not allow for pulling items out of the middle of the stack, so another way to deal with that had to be found. Instead of pre-sorting items going into the stack, we opted to use arraylist data structures to hold characters being read.

By using if/else statements and switches we were able to sort the incoming data into elements and operators and assign order of precedence for the operators. Once the functionality for sorting the characters was figured out completing the compiler came down to testing and debugging. This took more time than expected. When the output error was universal tracking down the issue was pretty straight forward. When the error was more random tracking down the issue was laborious.

Throughout the project we used Github for version control and sharing. This allowed us to make changes while debugging at each step without fear of making the issue worse. That was especially true while completing final step to achieve a full-fledged compiler. The hard work and testing paid off when we finally worked out the bugs and completed our compiler.

Working UML


Driver
- file: FileReader
- stream: CharStream
- lexer: junkLexer
- vocab: Vocabulary
- tok: Token
- tokens: CommonTokenStream
- parser: junkParser
- listener: Listener
- walk: ParseTreeWalker
- ex: Exception
<<External Library>> org.antlr.v4
+ JunkbaseListener{ }
+ JunkListener{ }
+ JunkLexer{ }
+ JunkParser{ }
+ Lexer{ }
+ Parser{ }
+ ParseTreeWalker{ }
Symbol
- type: String
- name: String
- value: String
+ Symbol {type, name}
+ Symbol {type, name, value}
+ getType{type}
+ getName{name}
+ getValue{value}
+ print{ }
+ toString{print}
SymbolTable
+symbolTable{ }
Listener <extends junkBaseListener>
-s: SymbolTable
- symbol: Symbol
- newTable: boolean
- newTableHeader: boolean
- programHeader: boolean
- variableType: String
- variableValue: String
- variableName: ArrayList<String>
- block: int
+ enterProgram {junkParser.ProgramContext ctx}
+ exitProgram {junkParser.ProgramContext ctx}
+ enterFunc_decl {junkParser.Func_declContext ctx}
+ enterFunc_body {junkParser.Func_bodyContext ctx}
+ exitFunc_decl {junkParser.Func_declContext ctx}
+ enterVar_decl {junkParser.Var_declContext ctx}
+ exitVar_decl {junkParser.Var_declContext ctx}
+ enterVar_type {junkParser.Var_typeContext ctx}
+ exitVar_type {junkParser.Var_typeContext ctx}
+ enterId {junkParser.IdContext ctx}
+ exitId {junkParser.IdContext ctx}
+ enterString_decl {junkParser.String_declContext ctx}
+ exitString_decl {junkParser.String_declContext ctx}
+ enterStr {junkParser.StrContext ctx}
+ enterIf_stmt {junkParser.If_stmtContext ctx}
+ exitIf_stmt {junkParser.If_stmtContext ctx}
+ enterElse_part {junkParser.Else_partContext ctx}
+ exitElse_part {junkParser.Else_partContext ctx}
+ enterWhile_stmt {junkParser.While_stmtContext ctx}
+ exitWhile_stmt {junkParser.While_stmtContext ctx}
+ getSymbolTable { }
+ popSymbolTable { }
+ enterWrite_stmt {junkParser.Write_stmtContext ctx}
IRBuilder
- ir_list: ArrayList<String>
- condition: String
- labelNum: int
- labelStack: ArrayList<String>
- regNum: int
- compIR: String
- dataType: char
- currentTable: SymbolTable
- stack: ArrayList<String>
- postfixOutput: ArrayList<String>
- inExpression: boolean = false
+ IRBuilder {SymbolTable}
+ updateTable {SymbolTable}
+ clean { }
+ trimStack { }
+ printList {ArrayList, boolean}
+ isNumber {String}
+ isRegister {String}
+ enterMain { }
+ endProgram { }
+ buildWrite {String [ ]}
+ buildRead {String [ ]}
+ parseComparison {String [ ]}
+ setCondition {String}
+ routeCondition {String, String}
+ exitCond { }
+ setCompop {String}
+ buildWhileHeader{String, String }
+ endWhile { }
+ buildIfHeader {String, String}
+ enterElsePart { }
+ exitIf { }
+ enterExpression { }
+ exitExpression {boolean}
+ addElement {String}
+ addOperator {String}
+ printPostFix { }
+ popPostFix {boolean}
+ elementToIR {String}
+ exprToIR { }
+ assignmentStatement {String}
TinyBuilder
- ir_list: ArrayList<String>
- table: SymbolTable
- tiny: ArrayList<String>
- vars: ArrayList<String>
- regNum: int
- dataType: char
+ TinyBuilder {ArrayList<String>, SymbolTable}
+ print {boolean}
+ printList {ArayList<String>, boolean}
+ parseIrList { }
+ checkForVariable {String}
+ foundVariable {Symbol}
+ parseLabel {String}
+ parseStore {String}
+ parseComp {String}
+ parseSub {String}
+ parseMul {String}
+ parseDiv {String}
+ parseJump {String}
+ parseRead {String}
+ parseWrite {String}
+ parseRet {String}

Conclusion and Future Work

While we were able to keep our code to a minimum and conserve space, we did not have enough time to optimize the IR code. A future endeavor will be to write methods for reducing redundant operations when converting the IR code to the three address code. By reducing redundant operations we can reduce the number of memory registers needed to store variables and results. This also reduces needing to spill registers when creating the final assembly code.

In conclusion, this project brought together many facets of the computer science discipline. Working in incremental steps allowed us to focus on the task at hand without being overwhelmed by the the final goal. Understanding how a compiler is built has uses outside of building another compiler. For example, reading, parsing and placing data it into tables is a useful tool.