Software Requirements Specification

for

JenPile

Version 1.0 approved

Prepared by Jennifer Felton

California State Fullerton - Compilers

December 15, 2020

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this compiler project is to demonstrate knowledge of how a compiler works. This compiler program was created for California State Fullerton's CPSC 323 Class, Compilers and Languages. Design specifications were given by the professor.

1.2 Document Conventions

There are no special document conventions to know about while reading.

1.3 Intended Audience and Reading Suggestions

This documentation is intended for any audience that is interested in how the compiler was created. It gives further information into design considerations and the project specifications from the professor. It can be read in any order, but is intended to be read in document order.

1.4 Product Scope

The scope of JenPile is to satisfy the requirements as given by the Professor. The goal is to make a simple compiler made up of a Lexer, Parser, and Intermediate Code Generator. Due to coronavirus considerations, the project was shortened and different options were given to satisfy the requirements for a completed compiler.

1.5 References

This Software Design Specification does not refer to any outside style guides, specifications or documents. A list of references used to create the project can be found in Appendix A.

A complete copy of the project can be found at:
https://github.com/jenniferafelton/CSUF323_Compiler/

2. Overall Description

2.1 Product Perspective

JenPile is a simple compiler written in C#. It is for the CSUF Compilers and Languages class for the Fall 2020 Semester. The compiler was written in three parts over the course of the semester.

The main function of the compiler is to translate a simple set of written instructions into machine code. This was done in three projects, the Lexer, the Parser, and Intermediate Code Generation. JenPile has all three projects integrated into one.

2.2 Product Functions

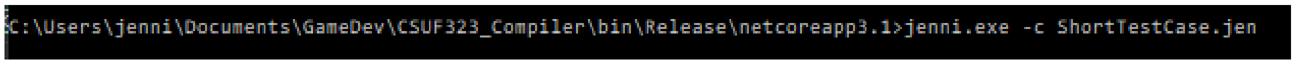
The main function of the compiler is to translate a simple set of written instructions into machine code. This was done in three projects, the Lexer, the Parser, and Intermediate Code Generation. JenPile has all three projects integrated into one.

2.3 User Classes and Characteristics

JenPile is expected to be used by the professor for grading purposes. Additional users may be individuals curious as to how a compiler works, or those interested in the construction of a compiler. It is not a working compiler for programming purposes.

2.4 Operating Environment

JenPile can be used on any operating system. It can be ran from the jenni.exe file, which will automatically compile the included .jen file, ShortTestCase.jen. If the project is run using the command line, using jenni.exe -c “YourFileNameHere” will allow the user to replace “YourFileNameHere” with any text file the user would like to compile. Including no file name and running JenPile from the command line will allow the user to type in text to be compiled. Below is an example of the command line.



```
C:\Users\jenni\Documents\GameDev\CSUF323_Compiler\bin\Release\netcoreapp3.1>jenni.exe -c ShortTestCase.jen
```

2.5 Design and Implementation Constraints

Due to the Coronavirus and online format, the compiler project was shortened. The Parser part of the project is only required to work for Arithmetic Expressions. The Intermediate Code Generation section of the project only required a Symbol Table for completion.

3. External Interface Requirements

3.1 User Interfaces

The only User Interface that is used is the command line.

3.2 Hardware Interfaces

There are no Hardware Interfaces.

3.3 Software Interfaces

JenPile uses no additional Software Interfaces

4. System Features

JenPile consists of three integrated project parts. The overall design follows SOLID programming practices to the greatest extent possible. There are also text files included for testing.

4.1 Lexer

4.1.1 Assignment Project Specifications

To write a lexical analyzer, using a FSM for the entire lexer, or using FSM for identifier, integer and real numbers.

The function lexer, should return a token when it is needed. The lexer should return a record, one field for the token and another field for the actual value (lexeme) of the token. The main program should read in a file containing the source code given to generate tokens and write the results to an output file.

4.1.2 Design of Lexer

The Lexer varies little from the assignment specifications. It reads in a text file, or input from the user. It processes the input through Regex expressions and a Token dictionary to create Tokens that are created as a token type and a value. This processing allows for the tokens to be used in the future for parsing and machine code generation.

The Lexer is composed of the following files:

Program.cs
Uses argos to pass in the file name, if the file to compile is null, then input is collected from the console. Prints the finished token list.

InputCollector.cs

Reads in a file, or from the Console, line by line. Places each line read in a List structure until an empty line is reached.

TokenType.cs
Contains the token types as an enum. The Token Types are none, keyword, identifier, separator, operator, integer, float and undefined.

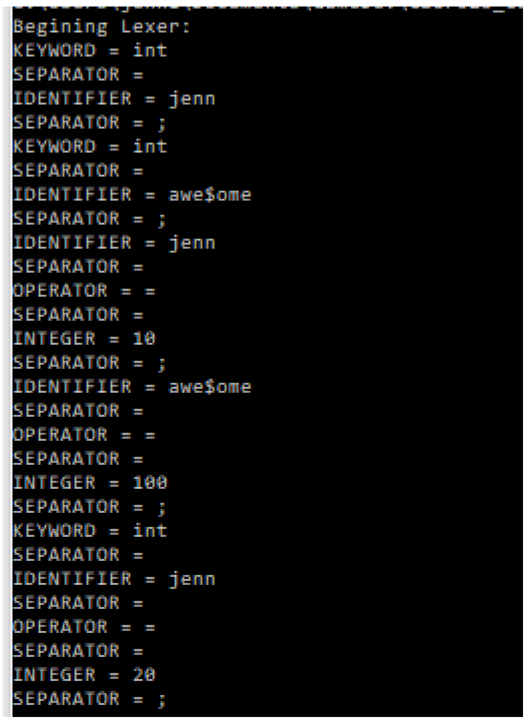
Token.cs
Contains the struct for the token type, value pair

TokenDictionary.cs
Contains a Dictionary of given keywords and operators paired with their token types. The token types are the ones identified in the given keyword and operator list.

Lexer.cs
Using the List from the InputCollector class, for each char in each line; first it checks for comments. If a ! is found, characters are not appended until another ! is reached.

Then the Lexer iterates through each character, appending each character to evalLine until it matches a separator. When a separator is found, the evalLine is checked to see if it is in the TokenDictionary. If it is not, evalLine is checked against each Regex pattern until a match is found. Regex is a regular expression Finite State Machine that does pattern matching for the defined patterns. The defined pattern types are for separators, identifiers, floats and integers. If a token is not identified, evalLine assigns it the undefined token.

Lastly, the token type and its value are paired together and added to a new List, along with the separator and it's token type. A list was chosen so it would be easier to iterate backwards and forwards during the Syntax stage. The List currently prints to the console with the Token Lexeme pair.



```
Begining Lexer:
KEYWORD = int
SEPARATOR =
IDENTIFIER = jenn
SEPARATOR = ;
KEYWORD = int
SEPARATOR =
IDENTIFIER = awe$ome
SEPARATOR = ;
IDENTIFIER = jenn
SEPARATOR =
OPERATOR = =
SEPARATOR =
INTEGER = 10
SEPARATOR = ;
IDENTIFIER = awe$ome
SEPARATOR =
OPERATOR = =
SEPARATOR =
INTEGER = 100
SEPARATOR = ;
KEYWORD = int
SEPARATOR =
IDENTIFIER = jenn
SEPARATOR =
OPERATOR = =
SEPARATOR =
INTEGER = 20
SEPARATOR = ;
```

4.1.3 Limitations of Lexer

When read in from the console, if two identifiers are typed on two lines with no separator, it will read them together as one lexeme.

The = sign can be both an operator and a seperator, so when used with no spaces, it is classified as an undefined token. When space is given on both sides of the =, it is correctly identified as an operator.

4.2 Parser

4.2.1 Assignment Project Specifications

The second assignment is to write a syntax analyzer. It can use any top-down parser such as a RDP, a predictive recursive descent parser or a table driven predictive parser. All grammar must be rewritten to remove left recursion. Arithmetic expressions should be done first, then Assignment and declarations. The Parser should print to an output file the token, lexeme, and production rules.

4.2.2 Design of Parser

The parser is designed to be a Bottom to Top Shift Reduce Parser. It starts from the leaves of the parse tree to the root. The list from the Lexer is used as the input buffer, and a list named theStack stores and accesses the production rules. The stack was originally a stack, but traversing it to check for grammar quickly became difficult.

The variable “theStack” was originally a stack. It was also a stack of strings for the parse types, but this removed the value and token so they couldn’t be used together on part 3 if it was needed. I also briefly considered making each token also have a third type called parse to organize the parse types. For now the choice was made to add the parse types to the TokenType list. There is an enum set up with the Parse types from that attempt, though it is not used in the submitted version. It was also debated to make a stack that consisted of a ParseType enum and a token, but that got complicated quickly.

The Parser uses the existing Lexer as input. The list or Token - Value pairs created in the Lexer is first passed into the driver for parsing. The following methods handle the parsing of the pairs.

The Parser is entirely contained in Parser.cs. The Following methods are used in implementing the Parser.

Driver:
The driver method is used to drive the reading of the Token - Value pairs. It adds a Token at the end to designate the end of the file to be parsed. The driver skips white spaces in parsing, and calls two functions, Shift and Reduce.

Shift:
Shift handles the adding of the tokens to the stack for processing. It also writes the line.

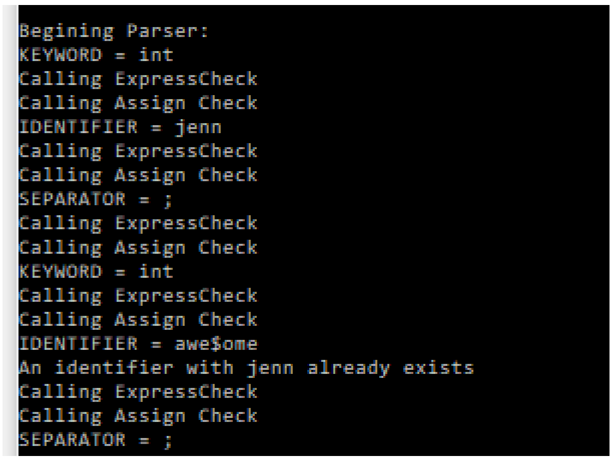
Reduce:
Reduce calls three methods that do the reducing of expressions, assignments and statements on the stack.

CheckForExpressions:
This method checks for an identifier, operator, identifier set of tokens and reduces them to an expression. This also takes the three tokens and reduces them to one.

CheckForAssignment:
This method checks for either a keyword, identifier =, expression set of tokens, or a identifier, =, expression set of tokens. When the correct grouping is found, it reduces it to one token.

CheckForStatement:
This method checks if there is a statement, which would be an assignment or a declaration with a ; at the end. The statement function is designed to be the topmost piece that almost everything should reduce too, ending with a semicolon. This function would eventually handle if, while and do.

PrintRule:
Print Rule prints the rules when a correct rule is found.



```
Begining Parser:
KEYWORD = int
Calling ExpressCheck
Calling Assign Check
IDENTIFIER = jenn
Calling ExpressCheck
Calling Assign Check
SEPARATOR = ;
Calling ExpressCheck
Calling Assign Check
KEYWORD = int
Calling ExpressCheck
Calling Assign Check
IDENTIFIER = awe$ome
An identifier with jenn already exists
Calling ExpressCheck
Calling Assign Check
SEPARATOR = ;
```

4.2.3 Limitations of Lexer

Rules are only printed when successfully used. This could be changed by calling the rule method when the method checking for it is called.

A limitation brought over from the Lexer and also apparent in the parser is that letters with no spaces in the middle, or no line break after them, are read together into one identifier.

In the Expression function- in arithmetic expressions, an operator is used between the two identifiers or expressions. The operator can currently include < and > , so more logic would be needed to correctly handle what operator it is. It was also considered that an expression can be an identifier, float or integer by itself, but that would not be syntactically correct to just have a word, float or integer without anything else paired with it. The choice was made to leave it as an identifier operator identifier for simplicity.

Check for assignment operates on the limitation that an assignment operator will always be located at the beginning of a statement, and always has an “id = expression”, “id = float/int/id”, or “keyword id = expression/float/int or id”. It is constricted to only look at the beginning of the list when the list is long enough to have these tokens.

CheckForExpression does not change individual tokens to expression, since it is not an expression if it was by itself. It would need to be assigned to something or some operations would need to take place making it an expression.

There is no end of file token on the input string. When the input string is at the end, that is the end of the file.

In the future, the keyword token would have to be handled differently, as the parser uses certain keywords in different ways. Though, this could be handled by looking at the value after sorting out the token types.

4.3 Intermediate Code Generator & Symbol Table

4.3.1 Assignment Project Specifications

The third assignment consists of Documentation and Specifications for the compiler project, a symbol table with type checking, and one of the following options: generating intermediate code from the grammar from the Parser, or implementing a different approach for the Syntax Analyzer. Due to Coronavirus considerations, the third assignment was shortened in class to consist of the documentation and symbol table, with all other parts of the assignment being extra credit.

Documentation should include explaining approach and function, program flow and diagrams for maximum points.

The Symbol Table should take every identifier declared in the program and place it in a symbol table. The symbol table should hold the lexeme and a "memory address" where it can be found. The table should be checked every time an identifier is declared and return an error message if it is found. The symbol table should also make sure the type matches.

4.3.2 Design of Symbol Table

_____Due to time limitations, no extra credit was implemented for assignment three.

The symbol table uses a Symbol struct which consists of a symbol type and an identifier. The Symbols are then placed into the SymbolTable.The Symbol Table was implemented as a Dictionary, since duplicate entries are not permitted. The identifier name was used as a key, with a Symbol Type being used to identify if the identifier is an int or a float. TokenType would not be useful in the SymbolTable, since a symbol would consist of the keyword, identifier, and assigned value combo.

The Symbol Table was added to the Parser. Whenever an identifier token is found, the keyword - identifier pair is converted to a Symbol. This Symbol is then added to the SymbolTable with the pretend memory location int. Each successful add increases the Memory address by one.

The Symbol Table is composed of the following files:

SymbolType.cs

This contains the enum for the different symbol types. For this project, only int and float are used.

Symbol.cs

This is a struct Symbol made up of a SymbolType Keyword and a string Identifier.

SymbolTable.cs

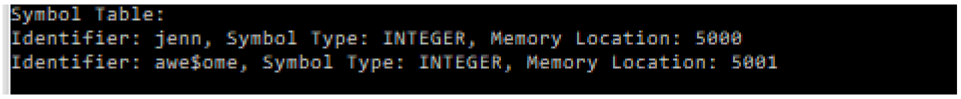
This contains the variable for the simulated memory location. It also contains the Symbol Table dictionary that is made up of a Symbol and an int. It contains a method to check if the identifier is in the table. If the identifier is not in the table, it adds the identifier and the current memory location to the table, then increments the memory location by one. If the identifier is in the table, it outputs an error message. There is also a method to print the current table with its symbol, identifier, and memory location values.

Parser.cs

In the Parser.cs file, the method CheckSymbolTable is called by the Driver method whenever an identifier is found. CheckSymbolTable checks if there is a Keyword token before the identifier, and checks what keyword it is. If it is an int or a float, it creates a Symbol using the keyword - identifier pair and calls the AddToTable method from SymbolTable.

Lexer.cs

In Lexer.cs, during token creation all input is identified using Regex to determine if it is an int or float independent of it’s declaration. The Parser.cs can check in the method CheckForExpression if the keyword and identifier is being declared and assigned a value. If it is assigned a value, the TokenType of the value can be checked to see if it matches the Keyword value. If it does, then the Symbol is syntactically correct.


```
Symbol Table:
Identifier: jenn, Symbol Type: INTEGER, Memory Location: 5000
Identifier: awe$ome, Symbol Type: INTEGER, Memory Location: 5001
```

4.3.3 Limitations of Symbol Table

There can be instances where the symbol can be declared and placed into the SymbolTable, but not have a value assigned yet. The design choice was made to only have the dictionary contain the memory address and the Symbol. If the SymbolTable was to contain both the declared type, identifier name and current value along with the memory location, the Symbol struct could easily be modified to contain that.

4.5 Test Files

The following files are included in the JenPile project and were used for testing purposes, or as example input/output files.

AssignTestCode.jen
This is a simple test file given for testing the Parser for correct grammar.

HelloWorld.jen
This is a simple file for testing the Lexer.

OutputAssignment.jen
This file was given to show the example output if the Intermediate Code Generator was done. It was used for reference for the Symbol Table output. It was included with the project for future reference and use.

OutputTestCase1.jen
This file was given to show the example output if the Intermediate Code Generator was done. It was used for reference for the Symbol Table output. It was included with the project for future reference and use.

SampleInputFile.jen
This file was given to test the Lexer and the SymbolTable.

ShortMathTest.jen
This file is for simple Parser grammar rules. It is the most basic test for the parser.

ShortTestCase.jen
This file was created for JenPile to test the Lexer and the SymbolTable.

TestArithExpressions.jen
This file was given to test Parser grammar rules.

TestCase1.jen
This file was given to test Lexer token generation

TestCaseAssignment.jen
This file was given to test the Lexer token generation.

---

5. Other Nonfunctional Requirements

5.1 Performance Requirements

There are no performance requirements for JenPile. No performance requirements were given.

5.2 Safety Requirements

There are no safety consideration or requirements in using JenPile.

5.3 Security Requirements

There are security requirements for JenPile. No security requirements were given. There is no encryption on any text. It is not recommended to enter sensitive information into JenPile.

5.4 Software Quality Attributes

JenPile is designed to be easily expanded and modified. The program is modular. Token Types, Symbol Types, and the Token Dictionary are easy to find in the compiler, and can be added to indefinitely.

Appendix A: References

This is a list of references that were referred to when creating JenPile:

GeeksForGeeks Shift Reduce Parser In Compiler
https://www.geeksforgeeks.org/shift-reduce-parser-compiler/

GeeksForGeeks Bottom Up or Shift Reduce Parsers
https://www.geeksforgeeks.org/bottom-up-or-shift-reduce-parsers-set-2/

Compiler Design Lecture 9 -- Operator grammar and Operator precedence parser
https://www.youtube.com/watch?v=n5UWAaw_byw&t=283s

Compiler Design Lecture 8 -- Recursive descent parser
https://www.youtube.com/watch?v=SH5F-rwWEog&t=3s

Parser and Lexer — How to Create a Compiler part 1/5 — Converting text into an Abstract Syntax Tree
https://www.youtube.com/watch?v=eF9qWbuQLuw&t=1782s

Wikipedia: Bottom-up Parsing
https://en.wikipedia.org/wiki/Bottom-up_parsing