**Java to C Compiler Project**

**“CoopJa"**

**Group:**

**Nicholas Araklisianos**

**Miguel Cruz**

**Jacob Poersch**

**Carlos Sandoval**

**SPRING 2019 - May 16, 2019**

**COMP 430 – Language Design and Compilers**

**TABLE OF CONTENTS**

Cover Page ….…………………………………..…………………………………………..1

[Table of Contents](#qnfm90wnuvud)………………………………………………………………..……….....2

**Section 1. The Goal of the Project**………………….....…...………………..…….…...3

Project Overview…………………………….………...............................................……3

Definitions, Acronyms, and Abbreviations…………………………………...……….......4

General Constraints…………………….……………………………………………….......4

**Section 2. Language Specifications & Features (Syntax)** ……….…….……..…...5

**Section 3. Implementation Order & Descriptions …..**…….……...……….…….…..7

Retrospective….......………………………………………………………………..…….…9

**Section 4. Tokenizer** ………………………………..…..……………..…...……….…...10

Files Pertaining to the Tokenizer…………………………………………….….….…….10

Other Files ….……….......………………………………………………....………………10

Reserved Words / Symbols….……………………….…………………………………...11

Variable Naming Restrictions……….......…………………………...……………………13

Tokenizer Output……….......…………………………………….…...……………………13

**Section 5. Parser** ………………………………..…..……………..…...……….………..16

**Section . Code Generator** ……………………..…..……………..…...……….………..

# 

# Section 1. The Goal of the Project

## Project Overview

The goal of our project is to create our own programming language based on the existing programming language “Java.” Then, we will use this made-up language as input and create a Compiler that will translate the given program into valid C code.

Since we all are most familiar with the Java programming language, and our made-up language is based on Java, we decided to make this our Implementation Language. After some discussion between the group members, we decided we could add some things that would be meaningful to the C programming language. We were also all interested in working with the C language as well, so we made C our Target Language.

The name of our compiler “CoopJa” means and comes from the abbreviation: “C’s Cooperative Object Oriented Programming from Java.” This is because we plan bring Java’s Object Oriented Programming nature to C with our compiler. Since Java supports OOP and C does not, our compiler will bridge the gap and implement this feature in C.

## Definitions, Acronyms, and Abbreviations

This section outlines all definitions, acronyms, and abbreviations that may not be known to some / all of the readers of this documentation. These terms all pertain to our project specifically or relate to general terms we may use.

|  |  |
| --- | --- |
| **Term/Acronym** | **Definition** |
| **Implementation Language** | **The programming language our Compiler is written in. In our case: Java** |
| **Target Language** | **The language our Compiler will compile to / its output. In our case: C** |
| **OOP** | **Object Oriented Programming** |
| **“CoopJa”** | **The name of our compiler. It stands for: C’s Cooperative Object Oriented Programming from Java** |

## General Constraints

Due to the limited time of the semester, it is necessary to restrict our made-up language to have fewer features than the full Implementation Language. We will not be featuring any memory de-allocation, nor any automatic garbage collection. We also will not be featuring any Generics in our language.

# [Section 2.](#fv16t6z0eqsp) Language Specifications & Features (Syntax)

The following is our made-up language’s syntax:

*var* is a variable

*objectname* is the name of a class

*methodname* is the name of a method

*str* is a string

*i* is an integer

type ::= int | double | char | boolean | string | auto | [Built in types of variables]

objectname [Objects are also types]

op ::= + | - | \* | / | [Arithmetic operations]

> | < | >= | <= | == | != | ==| | [Comparison Operations]

| | & | ^ | >> | << | ~ [Bitwise Operators]

var++; | [Increments a variable]

var--; | [Decrements a variable]

vardec ::= type var [Variable declarations]

exp ::= var | str | i | [Basic expressions]

Exp op exp| [Arithmetic expression]

println(exp)| [prints to the terminal]

This [Refers to this instance]

objectname.Method(Var\*) [Call Method]

new objectName(exp\*) [Declare a new instance of an object]

access ::= Public | Private | Protected [access type for a method or var]  
stmt ::= vardec; | [Variable Declarations]

var = exp; | [assignment to variable]

If (exp) stmt else stmt | [standard if/else statement]

while (exp) stmt | [loop statement with restriction]

for (vardec; exp; exp) stmt | [for loop statement]

break; | [escape loop statement]

{stmt\*} [block]

return exp| [return an expression]

return; | [Empty return]

instancedec ::= access vardec;

result\_type ::= type | void [Return types]

methodef::= Access result\_type methodname (vardec) stmt [Method declarations]

objectdefheader ::= access class objectname | access class objectname extends objectname

objectdef::= objectdefheader {

vardec\* [Variable declarations]

public objectname {smt\*} [Constructor]

methodef\*

}

program ::= objectdef\* exp [exp is an entry point]

# Section 3. Implementation Order & Descriptions

This section details the order in which we implemented each portion of our compiler and some details about why this is the case. Each specific part of the compiler will be further discussed in its own section later in the document. Note that the technical aspects of our code and implementation are outlined in these later sections.

The first step in the creation of our compiler was working on the Tokenizer / Lexer. The purpose of the Tokenizer is to go through our input program and find recognized keywords and terms in a “reserved words” list. For example, in order to have classes in our program, we must reserve the word “class” to create a class. The reserved words list includes the types of variables (like int, string, boolean, etc.), access modifiers (like public & private), symbol names (like equals, less than, slash, etc.) and other general reserved words (like void, null, true, etc.). A complete list of our reserved words will be in the Tokenizer section of the documentation.

The Tokenizer then identifies those reserved words and replaces those words with a Token object corresponding to that word. However, this list is impossible to be exhaustive, since a user can create a variable object with any name, within the restrictions of our language. For this, when the Tokenizer finds a word that it does not recognize, it classifies this as an “Identifier,” or the name of a variable, class, or method. The Tokenizer translates the entire input program into Tokens. This way, our compiler will be able to read and manipulate the given program to create meaningful output.

The next step in the process of writing our compiler was to write the Parser. The Parser takes the output from the Tokenizer and is able to decipher what is being asked from the original program and whether it is valid or not. For example, when declaring an “if” statement, the “if” keyword must immediately be followed by a left parenthesis, then an expression that resolves to a Boolean, then a right parenthesis and braces (Java & our language specific). If someone did not have the left parenthesis, the statement declaration would not be valid. The Parser uses the Tokens to determine if all that needs to be there in a declaration is in fact there. So the Parser will see an “if” token, and if it does not see a left parenthesis token immediately, it will call the statement invalid. However, the Parser is not able to tell if the statement within the parentheses actually resolves to a Boolean or not. This is because the Parser only cares about syntactic validity, and not content. The Parser is able to detect all that is defined within our language and checks the entire given program. When the Parser detects a proper-form “if” statement (for example), it puts this “if” statement into an object for ease of access later down the line. The Parser is able to define things such as variable declarations, method declarations, class declarations, and general statements. If the Parser passes, we can assume the program is syntactically valid.

After the Parser has checked the input program, the Typechecker will begin its check. The Typechecker’s job is to check the things the Parser does not, and ensure that there is no disparity among types. The Typechecker checks declared variables and makes sure they are not declared again or used incorrectly. For example, if the user declares an int variable but then tries to assign a string to it, this action would be caught as invalid by the Typechecker. The Typechecker would also be able to resolve the expression within an “if” statement (as mentioned previously), since it is keeping track of the names of all variables. This is to say the Typechecker is able to check the logic of the code. If there is no error in the Typechecker, the input program is valid.

The last step in the process of our compiler is the Code Generation. For this portion, we add some code to each object that the Parser creates. This code will be used to transform the given statement into C code. Code Generation is the process of checking each object, since we have determined that they are valid syntactically and logically, and outputting its C code. Once the entire program has been transferred to C code, our compiler can also output this program to a .c file and run it through a C compiler and check the output. This is used in testing to see if the expected output matches the actual output of the program we inputted.

## Retrospective

Looking back on our project, our group is very pleased with the results. It is amazing to see the project come together, from learning what exactly a compiler does, and what each component entails, to actually building those components. It was a very interesting process through which we learned a lot.

**Section 4.** **Tokenizer**

## Files Pertaining to the Tokenizer

* Token.java – The whole Tokenizer file is stored here, including a list of reserved words the Tokenizer uses.
* test\TokenizerUnitTests.java – The Tokenizer Unit Tests. Note “test\” refers to the fact that this file is not in the normal directory of .java files, and is in the test directory since it contains unit tests.

## Other Files

* TokenizerExampleTest.java – This file explains the usage of the Tokenizer and gives some examples of it being used. This file could also be considered the Main() method for the Token file, since Token.java does not have one.

## Reserved Words / Symbols

Our Tokenizer checks to see if the input program, word by word, is one of the listed reserved words. Complete list of reserved words here:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Keyword | Token Name | | void | KEYWORD\_VOID | | int | KEYWORD\_INT | | double | KEYWORD\_DOUBLE | | char | KEYWORD\_CHAR | | boolean | KEYWORD\_BOOLEAN | | String | KEYWORD\_STRING | | auto | KEYWORD\_AUTO | | if | KEYWORD\_IF | | extends | KEYWORD\_EXTENDS | | this | KEYWORD\_THIS | | static | KEYWORD\_STATIC | | class | KEYWORD\_CLASS | | true | KEYWORD\_TRUE | | |  |  | | --- | --- | | Keyword | Token Name | | public | KEYWORD\_PUBLIC | | private | KEYWORD\_PRIVATE | | protected | KEYWORD\_PROTECTED | | break | KEYWORD\_BREAK | | return | KEYWORD\_RETURN | | while | KEYWORD\_WHILE | | for | KEYWORD\_FOR | | false | KEYWORD\_FALSE | | null | KEYWORD\_NULL | | println | KEYWORD\_PRINTLN | | else | KEYWORD\_ELSE | | new | KEYWORD\_NEW | |
| |  |  | | --- | --- | | Symbol | Token Name | | + | SYMBOL\_PLUS | | - | SYMBOL\_MINUS | | \* | SYMBOL\_ASTERISK | | / | SYMBOL\_SLASH | | \ | SYMBOL\_BACKSLASH | | > | SYMBOL\_GREATERTHAN | | < | SYMBOL\_LESSTHAN | | ! | SYMBOL\_EXCLAMATION | | = | SYMBOL\_EQUALS | | | | SYMBOL\_BAR | | & | SYMBOL\_AMPERSAND | | ^ | SYMBOL\_CARET | | ~ | SYMBOL\_TILDE | | “ | SYMBOL\_QUOTE | | ; | SYMBOL\_SEMICOLON | | ( | SYMBOL\_LEFTPAREN | | ) | SYMBOL\_RIGHTPAREN | | |  |  | | --- | --- | | Symbol | Token Name | | { | SYMBOL\_LEFTCURLY | | } | SYMBOL\_RIGHTCURLY | | [ | SYMBOL\_LEFTBRACKET | | ] | SYMBOL\_RIGHTBRACKET | | , | SYMBOL\_COMMA | | . | SYMBOL\_PERIOD | | >= | SYMBOL\_GREATERTHANEQUAL | | <= | SYMBOL\_LESSTHANEQUAL | | ++ | SYMBOL\_DOUBLEEQUALS | | != | SYMBOL\_NOTEQUAL | | || | SYMBOL\_DOUBLEBAR | | && | SYMBOL\_DOUBLEAMPERSAND | | >> | SYMBOL\_SHIFTRIGHT | | << | SYMBOL\_SHIFTLEFT | | ++ | SYMBOL\_DOUBLEPLUS | | -- | SYMBOL\_DOUBLEMINUS | | |

## Variable Naming Restrictions

The Tokenizer is able to determine which inputted words are a variable / method / class name if it does not fall into one of the reserved words list. If the word does not appear in the list, it must be a user-defined variable name, also called an Identifier. After the Tokenizer checks the reserved words list, we use Regular Expressions to determine if this is the case. Restricting certain symbols from being part of the Identifier, and requiring the name begin with a letter or an underscore, but may contain numbers or more underscore characters after, our Regular Expression Pattern to determine this is:

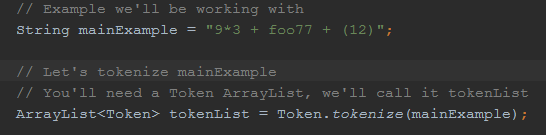
*Pattern p = Pattern.compile("\\\"(\\\\.|[^\"\\\\])\*\\\"|(>=|<=|==|!=|\\|\\||&&|>>|<<|\\+\\+|--)|[a-zA-Z\_]+[a-zA-Z0-9\_]\*|[0-9]+|\\S");*

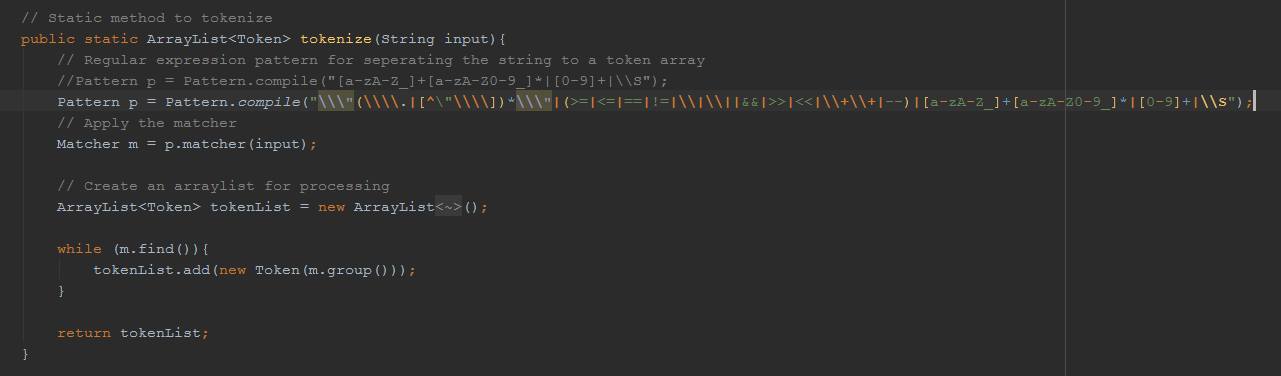
## Tokenizer Output

The Tokenizer converts string literal input into objects of type “Token.” But the ultimate result of the Tokenizer is an ArrayList object filled with these Token Objects. Below are some examples of input and the Tokenizer’s resulting output:

Example:

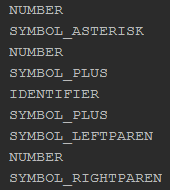
Here, let’s use the example file we discussed earlier named “TokenizerExampleTest.java.” In it, we see this:



The first line “String myExample” can be thought of as our input program. We take this input program and resolve it to a string. In this case, although it is a string literal, in other instances it may be a text file. This string is then sent to the static method “tokenize()” in class Token as the parameter. The result of this function is an ArrayList<Token> (and in this example) called “tokenList.” Here is the tokenize() function:  
  


The cut off portion is already listed in the “Variable Naming Restrictions” section. We define a Regular Expression pattern to match against our input String. The result is a Matcher object, which we convert to an ArrayList using the while loop. The final output is an ArrayList<Token>, which is returned at the end of the method.

So, our original input string was “9\*3 + foo77 + (12)” and our output from the Tokenizer is an ArrayList. When outputted, this is our result:



The Tokenizer was able to recognize all symbols and numbers, but the word “foo77,” since it is not in our reserved words list, is found to be an Identifier. At this stage, the Tokenizer does not know, or care, whether “foo77” is a class name, variable name, or method name. This is handled within the Parser.

**Section 5. Parser**

**Section . Code Generator**

## Files Pertaining to the Code Generator

## Converting Object Oriented Code and Other Designs to Functional

The job of the Code Generator (CG) is to convert the input program in our language into valid code in the Target Language so that it is executable. The CG must do its work after the parser and type-checker in order for the code to be valid in the Target Language. The CG takes as input the input program file. It goes through the code and, if necessary, modifies it to work in C.

Because C is not an Object Oriented programming language, our team had to figure out ways of making Object Oriented programming viable in C. It was decided that classes would be made into structs. Class members that were not methods become members of the struct, but method members are declared outside the struct as a global function. The name of the class that the function belonged to is included in the identifier. These (member) functions accept a pointer to the struct type they are associated with.