**Software Design Document for Mini-Pascal Compiler**

**Sam Horvath**

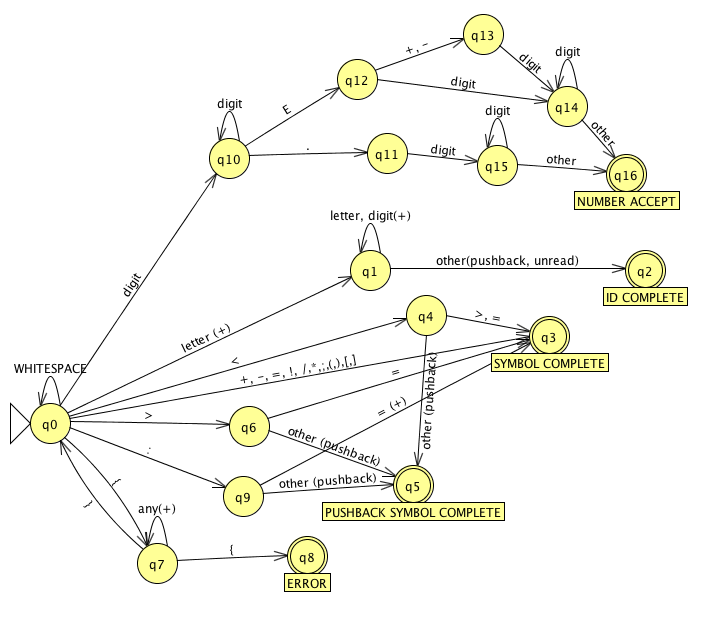
**Introduction**

In 1958, Navy Electronics Laboratory created the first compiler, to compile the ALGOL 58 programming language (Wikipedia). Since then, many compiled languages have been created, necessitating the building of many more compilers. In 1975 Professor Ken Thompson of Berkley created the Pascal language and its compiler, with the help of Bill Joy, among others (Wikipedia).

A compiler has 4-5 essential parts to it, depending on how one divides up the work of its creation. These parts include the scanner, the recognizer, the parser, the syntax tree, and the code generator. After code has been run all the way through these five parts of the compiler, it will be in another language. For CSC450, the task was to create a scanner and a recognizer throughout the semester, written in Java, to begin the process of creating a compiler that is able to convert mini-Pascal to MIPS assembly. In CSC451, we finished the creation of the compiler, building the parser, syntax tree, and code generator. At this point, the compiler is able to generate two important files when given a complete Pascal program as input. These files are a syntax tree, and working assembly code.

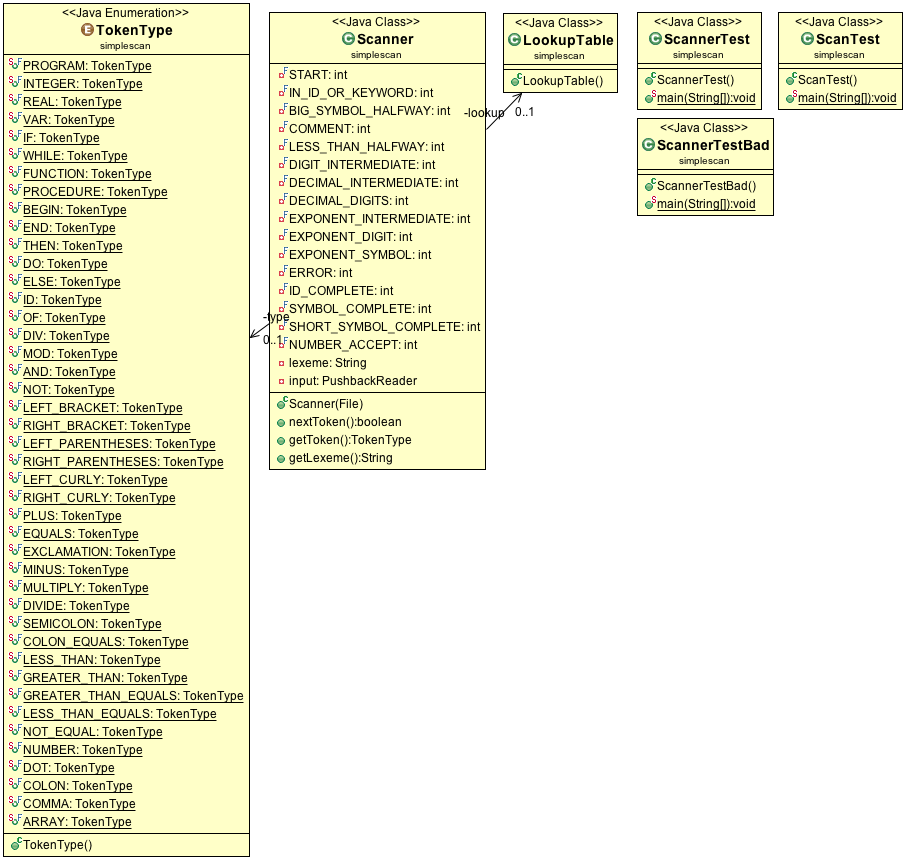
**I. Scanner**

The first step in creating a scanner is to create a Distinct Finite Automata (DFA) that maps out how any and all applicable symbols and keywords in the language will be built up in to their correct complete stages. A DFA resolves different characters, symbols, and keywords in to more general types that are major parts of the mini-Pascal grammar. The CSC450 class created the DFA pictured below in order to begin writing the scanner for their mini-Pascal compiler.



The DFA has five final, or acceptance states. These states are “NUMBER ACCEPT”, “ID COMPLETE”, “SYMBOL COMPLETE”, “PUSHBACK SYMBOL COMPLETE”, AND “ERROR”. The error state can only be reached by incorrectly creating a comment. Pushback symbol complete is an important and interesting completion state. It exists for those symbols which have been created within a larger line of text, and recognizes them after they have been created. The most recent character is then “pushed back”, leaving only the recognized symbol.

Following the creation of the DFA, the scanner was coded. The scanner is a coded replica of the DFA, using if/else blocks and switch statements to designate the state of the tokens as they are being created. The scanner then outputs tokens to be evaluated by the recognizer/parser. The UML for the scanner is included below.

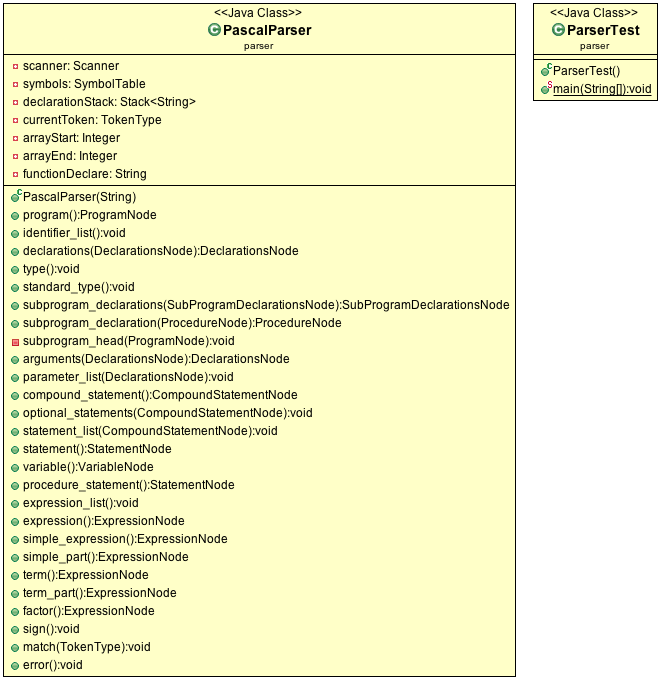


**II. Parser/Recognizer**

**a. Recognizer**

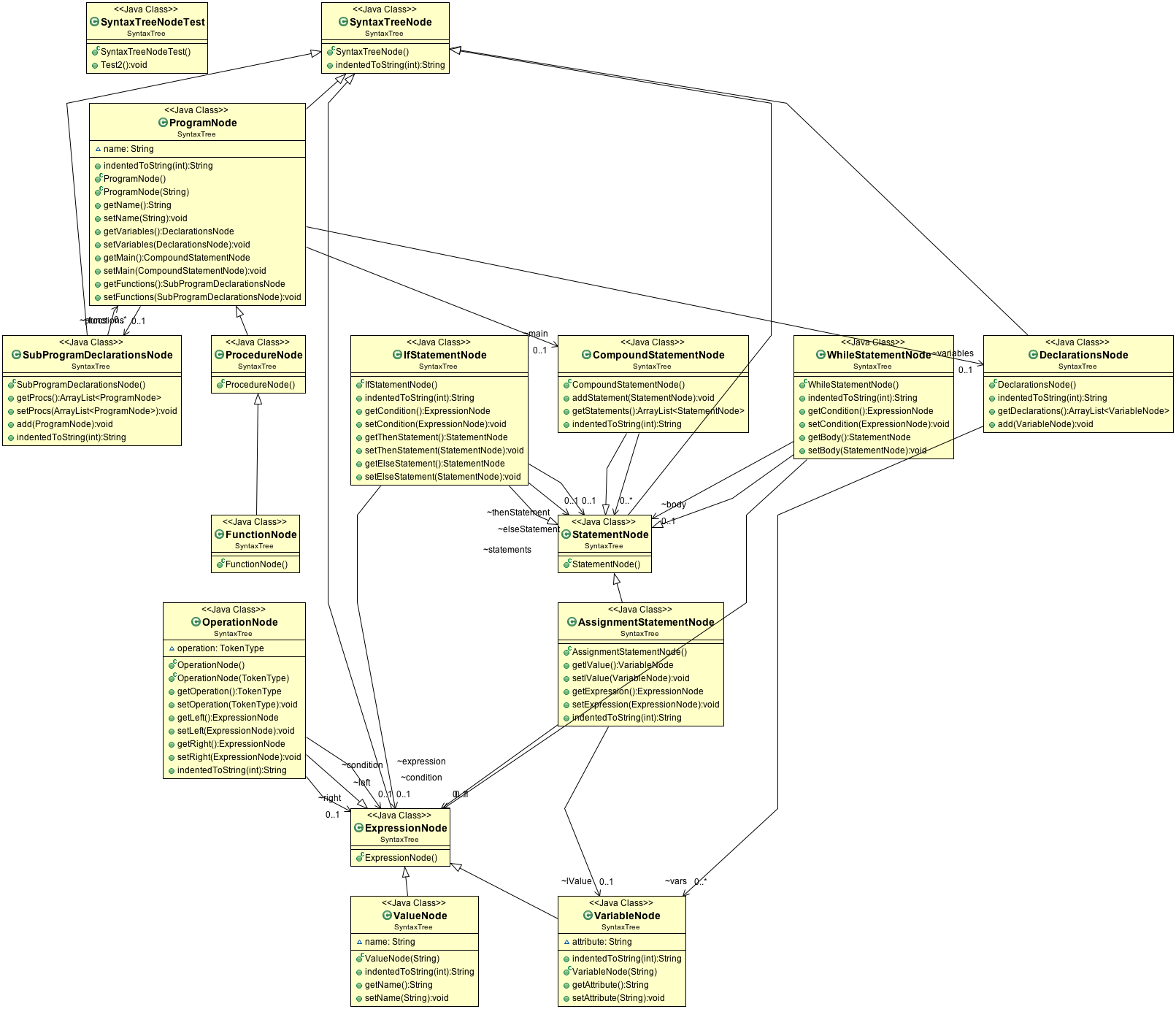
While the scanner’s job was to take in and identify the most basic types of tokens being created, the recognizer’s job is making sure that the grammar of the mini-Pascal language is correctly followed. By directly coding the grammar of the mini-Pascal language, we are able to ensure that we are following all of the rules that govern a complete mini-Pascal program. As the recognizer goes through the code, it validates or invalidates the code and outputs a Boolean result as to whether or not the code is Pascal code. In another iteration, the recognizer was replaced by the parser, which has a much more practical output.

**b. Parser**

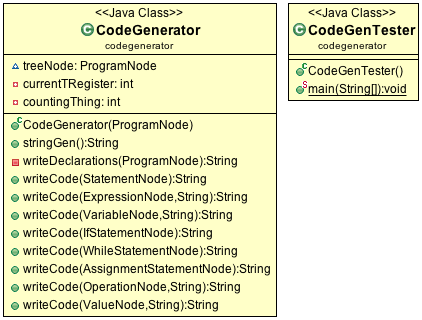
Unlike the recognizer, the parser begins to interpret what the code means to prepare to put it in to the assembly language. The parser uses the scanner to create tokenized input, and then uses recursive descent parsing on these tokens in order to generate nodes. In other words, the parser works by sorting through the code, looking for patterns that match the pattern of the grammar. These nodes are then used by the next section of the compiler to create a syntax tree. The UML for the parser is shown below. 

**III. Syntax Tree**

The syntax tree section of the compiler takes in the nodes generated by the parser, and uses them to create a syntax tree. The syntax tree is then used by the code generator to generate assembly code. The UML for the syntax tree is pictured below.

****

**IV. Code Generator**

The function of the code generator is, unsurprisingly, to actually generate the MIPS assembly code. This is achieved by moving through the nodes of syntax tree and using if/else blocks to determine what should be printed to the output file. The UML for the code generator is pictured below. 

**Appendix**

Below is the grammar, or the set of rules that govern the mini-Pascal language.

Macintosh HD:Users:SJHorvath:Downloads:Grammar.pdf





Works Cited

Wikipedia. Accessed December 16, 2015. [https://en.wikipedia.org/wiki/History\_of\_compiler\_construction - Self- hosting\_compilers](https://en.wikipedia.org/wiki/History_of_compiler_construction%20-%20Self-%09hosting_compilers)