Compilers Design Assignment 1

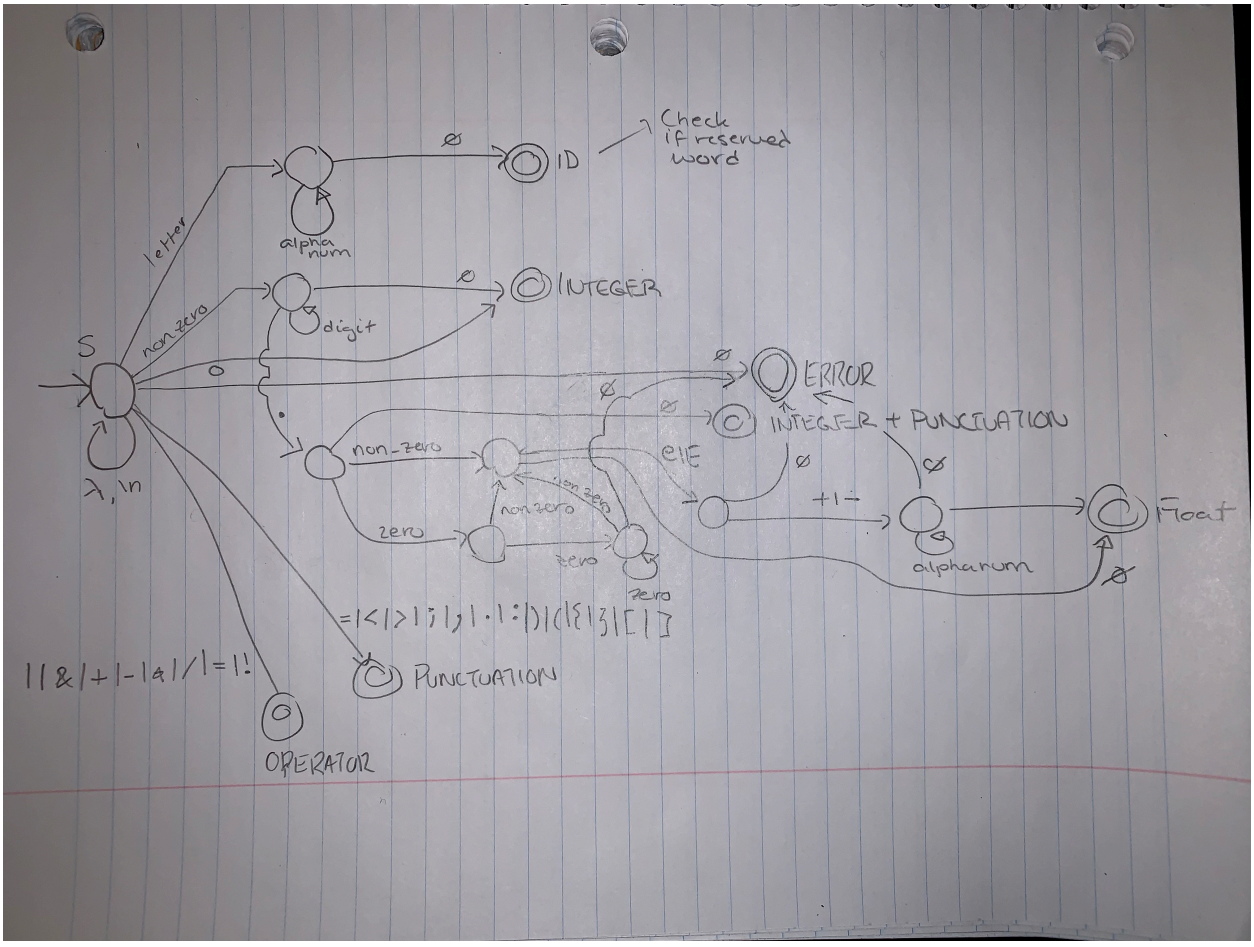
COMP 442 - Jacob Gagné - 40003704

LEXICAL SPECIFICATIONS

The lexical specifications in this assignment adhere to the handout specifications. Here is a copy of those specifications.

```
      id ::= letter alphanum*
alphanum ::= letter | digit | _
 integer ::= nonzero digit* | 0
   float ::= integer fraction [e[+|-] integer]
fraction ::= .digit* nonzero | .0
  letter ::= a..z |A..Z
   digit ::= 0..9
 nonzero ::= 1..9
```

FINITE STATE AUTOMATON



ERROR REPORTING AND RECOVERY

In this assignment, errors are encountered when tokens cannot be identified. This means that any character that cannot fit in a lexical specification is then marked as an error. Errors are reported in the terminal output with a character position. This is only an estimate since the comments are stripped from the file before parsing the content. This is done to increase the speed of the parser by removing useless iterations. Errors are then ignored by the parser itself and it continues and attempts to pickup its pattern matching at the next character.

DESIGN

The application is divided into a compiler, which acts as the driver and a tokenizer, which removes comments and tokenizes an input string. The tokenizer returns a sequence of token objects which are identified by their case class names and a value parameter. This assignment was written in Scala to leverage the powerful pattern matching mechanics of the langue. Scala allows the extraction of regular expression groups into variables with simple match/case statements which makes parsing tokens much easier than in traditional languages. Scala unit testing is also very easy using the scalatest library. English like spec files were created to validate the functioning of the tokenizer. The automata object stores key tags that represent automata states as well as a list of reserved words that are matched once an ID token is located. The AtoCCConverter takes a sequence of tokens as input and converts it to a string of AtoCC formatted values that are then outputted to a file by the compiler. Right now the program is setup to read from the resource folder where 3 program files are stored however, additional program files can easily be added and analyzed by dropping a new program with program#.txt naming convention and incrementing the programFiles variable.

USE OF TOOLS

Scala test was the framework used for unit testing which made it easier to validate the tokenizer. Regular expressions were also heavily utilized to strip comments and identify incoming characters.

TEST CASES

INPUT	OUTPUT A2CC	

main {
  this = help.me(123 + 444)
}
	RESERVED { ID = ID . ID
	(INTEGER + INTEGER) }
		[reserved:main][punctuation:{]
		[id:this][punctuation:=][id:help]
		[punctuation:.][id:me]
		[punctuation:(][integer:123]
		[operator:+][integer:444]
		[punctuation:)][punctuation:}]

/* main project */
 	
main {
  $## oops
  testing = 123
  return testing
  // hopefully this works
}
 	
/*
ramble on //test
/* */
	RESERVED { ID ID =
	INTEGER RESERVED
	ID }
		[reserved:main][punctuation:{]
		[id:oops][id:testing]
		[punctuation:=][integer:123]
		[reserved:return][id:testing]
		[punctuation:}]

/* float operations $test */
 	
test = 2
_WhatIsLove = 2
 	
main {
  integer float this = 3.124e-123
  invalidFloat = 324+425e
  read write *&#(
}
	ID = INTEGER ID =
	INTEGER RESERVED
	{ RESERVED RESERVED
	ID = FLOAT ID =
	INTEGER + INTEGER ID
	RESERVED RESERVED *
	& (}
		[id:test][punctuation:=][integer:
		2][id:WhatIsLove]
		[punctuation:=][integer:2]
		[reserved:main][punctuation:{]
		[reserved:integer]
		[reserved:float][id:this]
		[punctuation:=][float:
		3.124e-123][id:invalidFloat]
		[punctuation:=][integer:324]
		[operator:+][integer:425][id:e]
		[reserved:read][reserved:write]
		[operator:*][operator:&]
		[punctuation:(][punctuation:}]

ADDITIONAL TESTS

Unit tests can be found including all assignment sample tests in the TokenizerSpec file.

INSTRUCTIONS TO RUN

In order to run this project, install scala build tools (sbt) and run `sbt run` or `sbt test`