# Pish Compiler Checkpoint 1


## Mohamad Abboud

## Thomas Grove


### Compiler Functionality

**Lexer**: The Lexer is finished as far as we know; it will most likely change when something that we haven't thought of comes up forcing us to add or modify it. We have chosen to "filter out" the comments with the Lexer so we don't have to deal with them in the Grammar.

**Parser**: The Parser is finished to the point of parsing most of the Pish Language standard and building an Abstract Syntax Tree. The grammar provided by the Pish Standard has been mostly split up into smaller statements and the missing constant, type, loop and other section have all been added except for a few of the more specific concepts. The built-in functions writeInt, writeReal and writeChar are not directly handled as built in functions and True/False are also not handled as reserved words. These will be handled when we decide upon the best place for them.

**Abstract Syntax Tree**: The tree contains a node types for each major part of the Pish language such as Statements, Expressions, Variables, Declarations, Subprograms. These nodes in turn have several nodes that inherit from them to act as specific pieces of the grammar of the larger categories (Ex:// Statement and StatementLists).

**Error handling**: There is currently minimal Error handling and checking aside from the Lexer or Parser from throwing core-dumps. The only error checking implemented purposely is; Cup allows for overriding error handling functions and we have overridden one to print what caused the parser to fail and what line it was on. And the recovery of missing semicolons on Constants is implemented to show we have looked into possible error resolution (This fails under certain situations).

### Implementation and design

**Language**: Java

Java was chosen based on the amount of experience both group members have with and the fact objects lend themselves so well to the nature of Tokens produced by the grammar/scanner and nodes for the Abstract Syntax Tree.

**Lexer and Parser**: JFlex and Cup

JFlex and Cup were developed for Java using the Java language so they made the most sense and we didn't really look around for a comparable alternative. Since JFlex and Cup are written with java in mind it is easy to add-in Java Functions and Objects. JFlex allows the creation of a Symbol object that can be extended which provided us with the ability to make our own symbol object for storing and passing Tokens and values to Cup. The grammar in Cup allows for code insertion after each major rule which can be used to create AST nodes right away as needed.

**Abstract Syntax Tree**

As mentioned in the above section, the Syntax tree contains nodes to represent the main areas of the grammar and child nodes to represent the more specific. The main area nodes all inherit from an Abstract ASTNode Object as a way of standardizing the tree nodes so they can all be interchanged. These nodes are themselves Abstract so that their children can have independent implementation while allowing for comparison and interchangeability of node types. As an example; every node has a print method that is specific to each node type which can print itself and calls the print of its children. It is our impression that Objects are going to make error checking the nodes in the parse tree and intermediate code generation easier to understand and create.

**A Short Note on Testing**

The Lexer was easy to test, we threw a file with all the reserved words combinations that where easy to moderate to handle.

The Grammar had to be inputted mostly all at once but was tested in logical chunks. We started with a simple program then added constants, types, variables, functions, loops and ifs in increments, fixing and correcting as we went along.

The Abstract Syntax Tree was tested in a similar fashion as the Grammar, aka piece by piece .

**Member Assessment**

Most of the JFLex and Cup setup and programming was done on Mohamad's laptop by him while the SVN repository was still not set up. Mohamad also made most of the Grammar changes and additions, with some input and subjections from Tom.

The Abstract Syntax tree was a join creation which was created and coded over the weekend and Monday. The development and design was done on paper in the Reynolds Mac lab, coding was mostly on Mohamad's laptop to avoid conflicts and merges in SVN.

The Makefile was implemented by Tom with input from Mohamad on Monday/Tuesday since development was mostly done on Windows with Eclipse.

Testing was also done as a team. As mentioned above tested was done piece by piece on Windows. The Makefile was tested on Tom's Virtual Machined Ubuntu and then the compiler was tested with the final test files from the Windows tests.

The Checkpoint 1 Document was Written by Tom.