**Pish Compiler Checkpoint 2**

**Mohamad Abboud**

**Thomas Grove**

# Compiler Functionality

## Parser:

The Parser is mostly unchanged except for some errors in the grammar which were detected and fixed;

1. Record field recognition was fixed.
2. Records within Arrays.
3. Arrays within Records.

## Abstract Syntax Tree:

The functionality was extended to add more specific nodes for constants, including Chars and Numbers and Constant IDs. The node class hierarchy was heavily redefined.

## Syntactic Analysis

A few Recoverable errors have been implemented all other grammatical errors are treated as unrecoverable errors. The compiler will still attempt to parse the entire file when an unrecoverable error is found and will display further errors, but will not generate a symbol table or type check. Unrecoverable errors that we find a way to recover from will be implemented as we go along.

## Symbol Table

The Symbol Table can contain all the Pish variables, types, constants, functions, procedures, records, arrays and nesting of arrays, records and functions. The Tables is only built when the compiler can parse the file without an unrecoverable error.

## Semantic Analysis

The Semantic Analysis can discover all the normal errors such as

# Implementation and design

**Language**: Still Java

## Syntactic Analysis

Recoverable errors are handled with duplicated grammar that allows for the missing token to be ignored or filled in with a blank. Unrecoverable errors are handled by the abuse of the Cup “error” token and the built in recovery (panic mode; consume until a rule can be matched).

## Symbol Table

The Symbol Table construction is done in one pass of the abstract syntax tree using a Symbol Visitor. This is accomplished with heavy amounts of recursion over the Symbol Table as it is built. The table has been designed with these key points in mind. Records are handled slightly differently in some cases to support nesting.

1. The Symbol Table has six absolute “System” types that a variable can be mapped to. These are String, Char, Integer, Real, Null and Record.
2. The Symbol Table has a reference to the Scope (Symbol Table) above it (aka Parent) and its depth.
3. The Symbol Table holds a hashmap for each of the following; Symbols, Methods, Record Tables and Child Tables.
   1. A Symbol contains a pointer to the AST node and one of the 6 “System” Types.
   2. The Symbol Map contains an ID String and a Symbol associated with it.
   3. The Method Map contains an ID String and a Symbol list (function arguments) associated with it. A Symbol with the method ID (name) and return Type is also contained in the Symbol Map. A reference to the list of function arguments will be given to the function scope/table when it is generated.
   4. The Record map contains an ID String and a Symbol Table associated with it and like Methods a Symbol with an ID and Type = Record is added to the Symbol Table.
   5. The Child Table contains an ID String and a Symbol Table associated with it. The Tables are each a scope of one depth below and contained within the current scope.
4. It should be noted that if for some strange reason a table does not have a parent other than scope 0, something terrible has gone wrong and the compiler will exit.

## Semantic Analysis

The Semantic Analysis is implemented with the same Symbol Visitor that was used to build the Symbol Table. Each node will accept the Symbol Visitor and will call the appropriate checking methods for its node type and then pass the Visitor on to the node’s children.

When the Visitor encounters a variable, method, record, type, etc that is being used in a statement (compare, assignment, etc), the visitor will first check if the variable(s) exists, and then perform a type check. To accomplish this, the Visitor will first check its current scope’s Symbol table for a declaration. If no declaration can be found, the Visitor will start to recurs up through the scopes until it can find a declaration or throw an error if it cannot. Records are a treated as a special case and are given their own symbol table. To accommodate this nesting the Visitor will look for a record type in the Symbol Map and start a stack of Record Scopes until it finds the member of the correct Record, or not in which case it will throw a “variable does not exist or is not a member” error to the terminal.

# Member Assessment

Symbol Table generation and Semantic Analysis methods were designed as a team, and the coding was done by Mohamad.