Pish Compiler Final Document

Mohamad Abboud

Thomas Grove

Contents

Compiler Functionality

Lexer

The Lexer is set up to tokenize the entire Pish Language's reserved words with a few exceptions that we have chosen to handle at a different stage. Comments are "filtered out" so we don't have to deal with them in the Grammar. WRITEINT, WRITEREAL, WRITECHAR and TRUE/FALSE are treated as IDs like variables and functions, and are then checked against preloaded values in the Symbol Table.

Parser

The Parser is set up to parse the entire Pish Language into 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.

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.

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. A Symbol Table will be generated regardless of a unrecoverable error and type checking will be preformed.

Semantic Analysis

All the expected type checking is preformed such as variables and functions being undefined, redefined or being used with the wrong types. It should be noted that we do not support operations involving both real and integer numbers only one type or the other.

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.

Intermediate Code

The intermediate code generated by our Pish compliers contains all the temps required by the code as well as instructions such as arithmetic operations, function calls, loops, and compares.

Assembly Code

Assembly code is generated for everything but what is listed in the limitation sections.

Limitations

Our Pish compiler was designed and built to support all the main staples of the Pish Language such as nested functions, pass by reference and variable access above current scope. Some of these could not be implemented in assembly either because of the way we designed the tree and or because we ran out of time. These limitations include;

1. Function Parameters are pass-by-value.
2. There are only 2 scopes, Local and global and variables can be accessed from either scope. (There is a problem with the way we deal with temps and variable referencing).
3. A Function declaration within Functions is not supported.
4. A record within a record is not supported. Although records of arrays within records are supported.
5. Comparing Reals is not supported, this was a product of running short on time, it actually does sort of work but it is by no means reliable.
6. Type Promotion is not supported.
7. Accessing the counter on FOR loops is broken, the loop still works but the counter variable is not updated, only the temp running in the background (again a problem with the way we deal with temps and variable referencing).

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. Each node can accept a Print Visitor, Symbol Visitor and an Intermediate Visitor and contains methods unique to itself based on the type of visitor and required operation. A node can;

1. Print its place and information within the AST to a file.
2. Register itself in the symbol table (when appropriate).
3. Type check itself.
4. Generate its Intermediate Code.
5. Generate Temps required for Assembly and Intermediate code.

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.
   3.1. A Symbol contains a pointer to the AST node and one of the 6 "System" Types.
   3.2. The Symbol Map contains an ID String and a Symbol associated with it.
   3.3. 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.

3.4. 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.

3.5. 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.

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).

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.

Intermediate Code

Intermediate code is generated by the Intermediate Visitor. Each node knows how to generate required temps and the proper statements. The Visitor is run after the symbol table is fully built. When it encounters a node it will usually first printout the statement for that node, such as add or function call, and then recurs through the lower nodes to generate a list of required temps. These temps are then given an offset starting from the last offset recorded in the current scope (Variable offsets). This results in each node knowing how many temps that it requires and there offsets from the current frame pointer. It also allows the Visitor to get these temps in a list at whatever point in the tree that it is at.

Assembly

Assembly code is also generated by the Intermediate visitor but not on the same pass as the Intermediary code generation. Assembly can only be generated after the Intermediary code has been generated because that is where the temp allocation and offset take place. Since each node has a list of temps with offsets we can recursively generate code as we go along. While allowing for variable and temp collection for Function definitions, function calls and other nodes that require more than the information stored in a single node. For a good example of work is done can be found in IDnode, AssingmentExpression, BinaryOpExpression as they are the key bits to any statement.

Main and Functions start by collecting all variables and temps required in their scope. Main store variables in the .data section and temps on the system stack. Functions start by saving the return address and old frame pointer onto the stack (this is to implement recursion) and then loading all the variables and temps onto the stack. The frame pointer is then set to the top off variable section so that the stack can change leaving offsets off of the frame pointer unchanged. At the end of a function the stack pointer is set to the current frame pointer (this effectively destroys the function memory) and then the old frame pointer and return address are restored.

Problems Encountered

Lexer

The way we deal with comments seems to have a few hiccups and doesn't work 100% but it's not something we had time to fully fix.

Parser

We had a hard time adding common errors and general Syntactical errors which as in part JFlex but we got it working in the end.

Abstract Syntax Tree

The tree had to undergo several restructures, modifications and additions during the period between the first and second checkpoints. The first major change was a complete restructure of the tree nodes and the java class inheritance. This was brought about by a bad first design and problems with java inheritance which ended up putting everything in a chain like structure instead of a tree. The other changes and modifications were minor. The most noteworthy was the addition of a Constant section to the tree under Variables.

Symbol Table

There are only two noteworthy problems we had while designing and implementing the symbol table was Records. The first one was how we stored Records in general, in the end we had to give them their own mini Symbol table which in turn caused some type checking errors. That error was fixed by setting up a record “stack” that the visitor could use to return to the current node that called the type checking. The second problem was when we started generating Assembly code. The mini Symbol Tables caused offset calculation within records and the offsets of the temps required to calculate the record offsets to wrong. We almost had it fixed but in the end we were forced to drop nested records from Assembly, although you can have an array of records within a record.

Assembly

As stated in the Limitation section, Assembly cod generation is where we ran out of time and realized that our representation of the AST and Symbol Table prevented us from implementing everything in the Pish Spec.

When we dropped pass by reference we also had to drop referencing a higher scope (other than global) and nested functions. This was caused by the way were generating temps for intermediate code. Temps had to be generated with to mostly hold direct instead of addresses, and in certain situations we needed an address but were unable to pass it around properly. We latter found out that it also effected For loops to a degree (unable to update or access counter variables) but at that point it was too late to implement a solution.

The solution we could have implemented but didn’t have time for would have been as follows. Everything would get two temps, one for a direct value store and then the second for a pointer to the memory of the direct value. The tree nodes would then be set up to always pass a pointer instead of a direct value and only a pointer sometimes. All that would be needed would be an extra set to get to the value pointed at which would have solved all our problems and allowed for accessing a higher scope and nested functions.

Member Assessment

All the heavy design and planning was done as a group and a lot of the programming was done as pair programming. We designed the lexer, parser, AST and Symbol Table and Syntactic and Semantic error handling together as well as some of the coding for each. The final makefile was also written as a team

Mohamad did most of the heavy coding and implementation work for the AST, Visitors, Symbol Table and the type checking. Mohamad also implemented the Intermediate Code and Temp generation based on the notes provided in the lectures/seminars and some input from Tom. He was also the one that implemented any quick fix we needed during the assembly phase.

Tom helped code anything that Mohamad didn’t have time for and did most of the testing. Tom was also did all the planning and implementation for the assembly phase (getting Mohamad to implement anything quick fix that was needed). The Documentation was written by Tom with input from some Mohamad.