DUBLIN INSTITUTE of TECHNOLOGY

Institiúid Teicneolaíochta Bhaile Átha Cliath

Implmenting a interpreter for a scripting lanaguge using Haskell

Final Year Project Report

Zhen Lao

zhen.lao@student.dit.ie

Supervisor: Richard Lawlor

2nd Reader: Cindy Liu

April 3, 2011

This Report is submitted in partial fulfillment of the requirements for the award of the degree of **BSc Computer Science** of the School of Computing, College of Sciences and Health, Dublin Institute of Technology.

Abstract

In this thesis,

Keywords:programming language YUN

Declaration

I **Zhen Lao** hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed _____
Zhen Lao

Acknowledgements

I would like to thank my supervisor Richard Lawor,for his valuable advice and useful suggestions on my project.
I am also deeply indebted to all the other tutors and teachers in Computer Science for their direct and indirect help to me.
Special thanks should go to my friends who have put considerable time and effort into their comments on the draft.

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective and Motivation

The objective of this project is to develop a dynamic and weak typing interpreted language using Haskell.This language is able to support the following features,

- basic for loop and while loop
- basic if-else statement
- functional invocation
- arbitrary dimension list
- polymorphic list

Furthermore, in this project,the monadic design approach is applied as Haskell is different from other object oriented language.

1.2 Benefits of using Haskell

Haskell is an advanced purely-functional programming language.By applying the used of Haskell to this project ,I have significantly reduce the coding time and spent most of my time to the design phrase.

A pure function is a function that accepts an input and produce an output.In Object-Oriented language,program is constructed using classes and instances which encapsulate computations and states.Haskell program is constructed by functions as function is the first class member in Haskell.Typically the

main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives. All of these functions are much like ordinary mathematical functions. [Why Functional Programming Matters] [Functional Programming with Overloading and Higher-Order Polymorphism]

1.3 Development Methodology

Agile development methodology is used in the entire development process.This project has been initially identified to multiple iterations and each iteration contains three major phrases inducing research ,development and testing.

Researches have been done on investing some useful Haskell library like Parsec ,Happy,Alex and HUnit.

Eclipse with EclipseFP plugin provide support for source code and package management in Eclipse.To keep tract of the development process,I make use of **git** a source control tool to version all the source file.

1.4 Code Formats

In this thesis,two types of code will be listed in different formats.

Code listed without line numbers illustrates the design of the interpreter or other Haskell concept.Most of it are Haskell code,which looks like,

```
return a >>= k  ==  k a
m >>= return  ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Code listed with line numbers are example of **yun** ,the programming language designed in this project.It looks like,

```
1  program main (){
2          result = fib (10);
3          sys.printLn(result);
4  }
5  function fib (num) {
6          if (num ==0){
7                  return 1;
8          }
9          if(num == 1){
10                 return 1;
```

```
11          }
12
13          part1 = fib(num-1);
14          part2 = fib(num-2);
15          return (part1+part2) ;
16  }
```

Chapter 2

Compiler and Interpreter Technologies

2.1 Parsing technologies

2.1.1 Formal Grammar

Mathematically, formal grammar consists of:

- a finite set of terminal symbols.
- a finite set of non-terminal symbols.
- a finite set of project rules.
- a start symbol.

[**?**] [Three Models for the Description of Language] From the formal grammar definition, legitimate production rules can be written as

$$S \mapsto aS \, and \, S \mapsto ab$$

In this example,we can assume that the grammar consists of two projection rules and the starting symbol is S.The terminal symbols are lower letters $\{a, b\}$. From this example, If we start from the either rule 1 or rule 2 ,we could derive a grammar of $\{a^n b | n > 1\}$,which can be enumerate like $\{aab, aaab, aaaab, \cdots\}$.

2.1.2 Context-Free Grammar

Context-Free Grammar (CFG) A context-free grammar is has four component 1. A set of terminal symbols, sometimes referred to as "tokens." The

terminals are the elementary symbols of the language defined by the grammar . 2. A set of non-terminals, sometimes called "syntactic variables." Each non- terminal represents a set of strings of terminals, in a manner we shall describe. 3. A set of productions, which are rules for replacing (or rewriting) non- terminal symbols (on the left side of the production) in a string with other non-terminal or terminal symbols (on the right side of the pro- duction). 4. A start symbol, which is a special non-terminal symbol that appears in the initial string generated by the grammar. [?] Context-free grammar can be recognized by pushdown automaton.

2.1.3 The Hierarchy of Grammars

Noam Chomsky has describe three model of grammar ["Three models for the description of language"] and this grammar model has significantly effect the design of computer programming language.

Chomsky define a set of rule upon the formal grammar and categorize them into different levels.

The Chomsky hierarchy consists of the 4 levels:

- Type-0 grammars (unrestricted grammar). It is a unrestricted grammars that include all possible grammar that are possible to recognize by Turning machine.
- Type-1 grammars (context-sensitive grammar).if all rules are of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ where $\alpha\,\beta\,\gamma$ are terminal symbols and A is non-terminal symbol.
- Type-2 grammars (context-free grammar).
- Type-3 grammars (regular grammar).

2.1.4 Backus–Naur Form and Extended Backus–Naur Form

The Backus-Naur Form(BNF) is a metalanguage to write the production rule that expressing the type-2 grammar (context-free grammar).It restricts the appearance of terminal and non-terminal in each side of the production equation.A canonical BNF production rule may like follow,

$$< symbol >::= \_\_expression\_\_$$

The left side of the equation can only be non-terminal thus enclosed with <> .The right hand side can be terminals and non-terminals,a vertical bar

'—' is used to represent choice between terminal and non-terminals.

The Extended Backus–Naur Form (EBNF) and extension upon the BNF.Three regular expression qualifier is added to simplified some expression,they are,

- ? : which means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional (it can appear zero or one times)
- * : which means that something can be repeated any number of times (and possibly be skipped altogether)
- + : which means that something can appear one or more times

[**?**]

Recrusive rules of BNF like

$$1. < exp >:=< exp > |sub$$

$$2. < exp >:= sub$$

that expressing a sequence of a particular syntactic element can be simplified using quantifier in EBNF as $< exp >:= sub+$

2.2 Parser Generator Haskell Happy

Happy is a parser generator system for Haskell, similar to the tool 'yacc' for C. Like 'yacc', it takes a file containing an annotated BNF specification of a grammar and produces a Haskell module containing a parser for the grammar. [The Parser Generator for Haskell]

By using its own EBNF like syntax,used could write an parser description.The happy parser generator are able to recognize and compile it into Haskell source code.

2.3 Monadic Parsing using Parsec

In the early stage of this project,parse is build using parse C, Parsec is an industrial strength, monadic parser combinator library for Haskell. It can parse context-sensitive, infinite look-ahead grammars but it performs best on predictive (LL[Compilers: principles, techniques and tools.]) grammars. Combinator parsing is well known in the literature and offers several advantages to YACC or event-based parsing. [Parsec, a fast combinator parser]

Compared toe parser generator , monadic parsing has two major benifits 1. No need to learn additional parser generator grammar since parser combinator is written in the same language. 2.parser can be adjust easily .

2.4 Lexical analysis

Before parsing,the lexical analyser will scan the source code and generate a sequence of tokens.The tokens are often defined by regular expressions.

For example,the statement **s3 = s1 + "a string"** will be parse to tokens,

s3	identifier ,variable name
=	operator
s1	identifier,variable name
+	opeartor
"a string"	a string constant

The regular expression $[a-zA-Z]1[a-zA-Z \setminus \_0-9]*$ can identify strings that begin will alphabetical character.

2.4.1 The Lexer Generator Alex

Alex is a tool for generating lexical analysers in Haskell, given a description of the tokens to be recognised in the form of regular expressions. It is similar to the tools lex and flex for C/C++.

Alex takes a description of tokens based on regular expressions and generates a Haskell module containing code for scanning text efficiently. Alex is designed to be familiar to exisiting lex users, although it does depart from lex in a number of ways. [Alex User Guide]

Chapter 3

Monads

3.1 Overview

3.1.1 Side effect

In imperative language like C/C++ , it is often the case that it access the variable outside the function,eg. the error flag.In Java, the keyword **synchronized** is used to acquire look to the share resource,most of time,they are the resource from out side word.Imperative language are unrestricted to side effect,which makes it hard to write function that allows parallelism .On the other hand,Haskell restricts side effects with a static type system; it uses the concept of monads to do stateful and IO computations.[Imperative Functional Programming]

3.1.2 Monad

sions. Another approach to introducing effects in a purely functional language is to make the use of effects explicit in the type system. Several methods have been proposed, but the most elegant and widely used is the concept of a monad.[Imperative Functional Programming] Monads in Haskel have been used to model IO,state,logger,error as well as List.

3.2 Haskell and Category Theory

Category theory is a general theory that examine and organize mathematical object like set ,function,function domains Cartesian-set.

A Category C in category theory is defined below :

1. a collection of objects

2. a collection of arrows (often call morphism)

3. operations assigning to each arrow f an object $dom\ f$,its domain ,and an object $cod\ f$,its co domain.

4. a composition operator assigning to each pair of arrows $fandg$,with $cod\ f = dom\ g$,a composite arrow $g \circ f : dom\ f \to cod\ g$, satisfying the following associative law:
   For any arrow $f : A \to B, g : B \to C, and\ h : C \to D$(with A,B,C and D not necessarily distinct),

$$h \circ (g \circ f) = (h \circ g) \circ f$$

5. for each object A, an identify arrow $id_a : A \to A$ satisfying the following identity law:
   For any arrow $f : A \to B$,

$$id_a \circ f = f\ and\ f \circ id_a = f.$$

[?]

Category in Haskell

In Haskell,all the type in can be view as objects and all the function can be view as arrows.id function has been defined as follow.

```
id :: a -> a
```

It can be viewed as an identify arrow of all objects(types).

Functor in Haskell

Some high order function like fmap can be view as a functor.

```
fmap :: (a -> b) -> (f a -> f b)
```

From its signature we know that fmap maps arrows in category to another category.f a is the type constructor that takes a as its parameter an generate f a as a new type.So it maps the arrow from a $->$ b to f a $->$ f b.

Its instances types like Maybe and ReadP satisfy the functor law:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

Functions are the first member of the program in functional programming,since no size affect is not allow ,there should be a way to combine the all kinds of functions to from a new function instead of just simply chain the input output of each function as the former will generate intermediate output.

For instance ,counting the file of java source code in current directory can be written as follow:

$$ls - al.|grep * .txt|wc - l$$

To substantiate the this concept , let's use the map/fold fusion technique of Haskell as an example.

If we want to calculate the sum of the square of each element of a list eg. [1,3,4,6,7,9],the result of it is $1^2 + 3^2 + 4^2 + 6^2 + 7^2 + 9^2 = 192$.In Haskell ,we could use map and fold to address problem.

To avoid generating intermediate output from the first function to second function, the could rewite the hold function using a single fold

The all map/fusion is is equivalent to $foldrfe.mapg = foldr($
$xy- > f(gx)y)e$

therefore, the

$sum_of_square = foldr($
$xy- > x^2 + y)0$

3.3 Monadic Function

3.3.1 Data Type Constructor

Type constructors play a fundamental role in Haskell's monad support.

3.3.2 Monadic Function

A monadic function is function that produce , however,monadic function like **putStr :: String -> IO ()** can not be combined using
(.) :: (b -> c) -> (a -> b) -> a -> c.Monadic class constructor has tag

3.4 Monads

In Haskell,monad is used an abstract data type constructor to represent multiple kinds of computation such as a computation that will do IO action,or a computation that has state.Those computations are in-pure because that

manipulate the outside world.In Haskell.Mathematically, monads are governed by set of laws that should hold for the monadic operations [A Gentle Introduction to Haskell, Version 98]. There are two basic law in monads ,they are bind return .The Monad class is defined as follow:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

The return function can inject a value into monadic type. The bind function can combine two monadic function, one should be of type **m a** and another should be of type **a -> m b** .

Beside this two function,Haskell also provide other monadic operator which all derive from **return** and **bind**,they are:

```
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
liftM2  :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
ap :: (Monad m) => m (a -> b) -> m a -> m b
(=<<) :: (Monad m) => (a -> m b) -> m a -> m b
\$ ::(m a -> m b) -> m a -> m b
```

These monadic operation is define using the bind and return.For example ,liftM is defined by bind and return like

```
liftM f m1              = do { x1 <- m1; return (f x1) }
```

Therefore,when defining a monad,only bind and return need to be specified.

Monadic Characteristic

For all monad instance ,beside define three monadic operator ,they must apply three compulsory monad laws:

```
"Left identity": return a >>= f  ≡  f a
"Right identity": m >>= return  ≡  m
"Associativity": (m >>= f) >>= g  ≡  m >>= (\x -> f x >>= g)
```

In monad instance ,these monad laws will become a restriction of the operation when combining monadic function using monadic operation,these restriction will be discussed in following sections.

3.4.1 IO Monad

Haskell use IO monad to limit the IO sequence.Monadic operation are used to represent IO processing pipeline.

Haskell I/O has always been a source of confusion and surprises for new Haskellers.

Normally, the first thing to lean a new language to to print something on the screen,but it is not the case in Haskell.The first thing I do in Haskell is to write a quick sort algorithm which is a classic program to show the powerful Haskell list comprehension feature that brings the most elegant solution of quick sort compared to most of other language.On the other hand,its not that straight forward to build an IO system upon it inside language that emphasized purity.

It probably not the best monad for beginner to learn as it dost not provide escape function.The IO moand is implemented as part of language in **Base.GHC**,thus the logic of monad operator is not exposed to outside world.

```
unsafePerformIO :: IO a -> a
```

The above shows the code of an escape function which will break the purity of Haskell IO system.This is the "back door" into the IO monad, allowing IO computation to be performed at any time. For this to be safe, the IO computation should be free of side effects and independent of its environment.[haskell io document]

As we can guess from the nature of Haskell,IO monad is introduced to guarantee to following factor,

- The sequence of IO actions perform in a lazy Haskell context.
- perform IO as an function rather than a function or and command in functional programming context.
- combine IO,distinguish computation that with IO actions with the one without.
- file buffering , open and close file in a correct manner.

The code below shows combination of monad using bind operator.

```
ioAction
  =putStrLn "What is your name ?"
        >>= (_-> getLine >>=
               (\x -> putStrLn ("welcome" ++ x) ))
```

3.4.2 State Monad

The basic idea of a state monad is to allow us to represent functions which interact with local state variables (which are just called local variables in most languages), or global state variables (usually called global variables). Essentially, they allow us to simulate some aspects of imperative programming in a purely functional setting.

An alternate solution is that we can pass the initial state into each function and return the state together with the result,with may looks like,

```
add :: State -> Input -> (State ,Output)
-- implement

subtract :: State -> Input -> (State,Output)
```

To make it possible to composite computation with state we could rewrite all computation with follow.

```
add :: (Input,State) -> (Output,State)
-- implementation
subtract :: (Input,State) -> (Output,State)
combine = add.subtract
```

However,its tedious to write all the function with computation this way.The state monad provides a way to encapsulate the state logic into a monad without explicitly specify a State type as input and output.Like IO, the computations with state are tagged with the type constructor **State** indicating this is a state monadic function.Most importantly, the monad law and state monad implementation guarantee the follow fact of a computation with state , they are

- It possible to construct a new state monadic function without initial state.
- when combine two state,from the nature of state transformation ,the change of state of the second function is always after the first function.
- For a computation with state,we need to need the both the result of computation as well as the final state as an output of the computation.

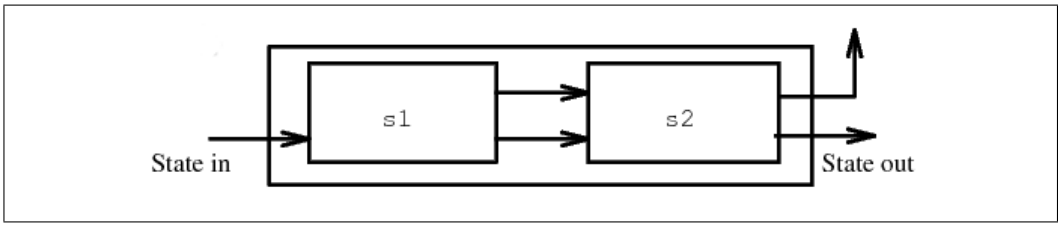

Figure 3.1: Combining Two "Stateful" Monadic Functions

3.4.3 List Comprehension and List Monad

3.4.4 List Comprehension

Haskell added syntax sugar to allow programmer to write code in a more readable manner.The list comprehension allow us to write a list using mathematics favour.

A Fibonacci series can be written as follow,

```
fibs :: [Int]
fibs = 0 :1 : [ a + b | (a, b) <- zip fibs (tail fibs)]
```

If we view if as a mathematical expression,we can easily figure out it means $\{0, 1, (0+1), (1+1), (1+2), (2+3), (3+5), ...\}$, which will generate a Fibonacci series.In addition,it is also an example of how useful of lazy evaluation is in Haskell.

3.4.5 List Monad

We can view a list as monad.In list comprehension ,we have the <- operator which we can view it as $\in$,which has a mathematical *in* semantic.

Alternatively ,

3.5 Using Monad Operator to Combine Monadic Function

3.6 Type system in Haskell

3.7 Do Notation

The do notation is a syntax sugar to write monadic operation especially bind in a imperative way ,to be more specified,only monadic function with the

same return type can write together.An IO action that return **IO ()** can not put with and state monadic function that return **State s a**.

To write a get line and put line logic using bind, the sequence of IO is guaranteed by monad laws.

```
getLine >>= (\x -> putStrLn x)
```

Following code shows that the same functionalities are written using do-notation,which shows a strong similarity to a imperative language.

```
do line <- getLine
   putStrLn line
```

Chapter 4

Monad Transformers

Monad transformers offer an additional benefit to monadic programming: by providing a library of different monads and types and functions for combining these monads, it is possible to create custom monads simply by composing the necessary monad transformers. [Monad Transformers Step by Step]

In real world application, it is often the case that a combination different type of action is required to put into one function.Monad transformer provide a new way to glue them together.

4.1 State Transformer

A value of type **(ST a s)** is a computation which transforms a state index by type s ,and delivers a value of type a ,You can think of it as a box ,like this

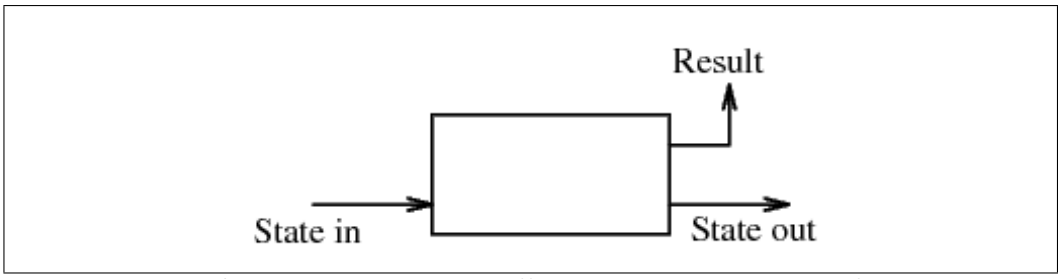

Figure 4.1: Diagram illustrating a state monad

[lazy functional state thread]

First , a state transformer is a monad, other than encapsulating a data behind signature , it encapsulated a computation,the **runState** function

provide by the state monad is an escape function that allow us to extract is state logic from the computation,by feeding the state computation function with an initial state,we are able to get the final state.Other type of escape function is provided to extract computation as well.

Second,a state transformer can accept other monads to combine the computation which state as well as other logic says the IO logic.

4.2 Error Transformer

ErrorT monad transformer can be used to add error handling to another monad. Below is an example how to combine it with an IO monad:

```
main = do
  -- runErrorT removes the ErrorT wrapper
  r <- runErrorT calculateLength
  reportResult r

-- Asks user for a non-empty string and returns its length.
-- Throws an error if user enters an empty string.
calculateLength :: LengthMonad Int
calculateLength = do
  -- all the IO operations have to be lifted to the IO monad in the monad stack
  liftIO $ putStrLn "Please enter a non-empty string: "
  s <- liftIO getLine
  if null s
    then throwError "The string was empty!"
    else return $ length s
```

4.3 Putting All Together

Chapter 5

Language Interpreter Design

5.1 Architecture

the interpreter contains two part,the parser and the execution engine. the parser will parse the source code into a parser tree,the execution engine will then execute the code according to the parser tree.

5.1.1 Parser

the parser that the source code as input and generate a tree structure as output.

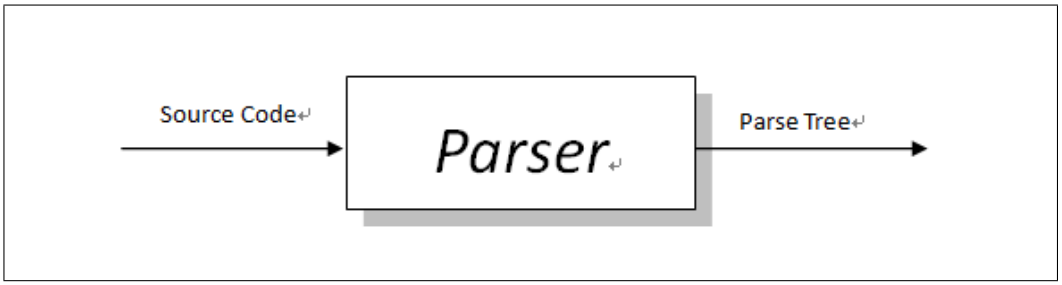

Figure 5.1: The Parser

the parser will contains different type of automata so that it could recognize different types of grammar.the grammar using to design the language is content-free grammar. the parser was build by combine simple and atomic parsers.The parser can be view as a function that a parser is a function that takes a string of characters as its argument, and returns a list of results. [?]

5.1.2 The Execution Engine

The execution engine of the parser is basic on Haskell.the execution engine will accept a parse tree and execute the command basic on this parse tree.

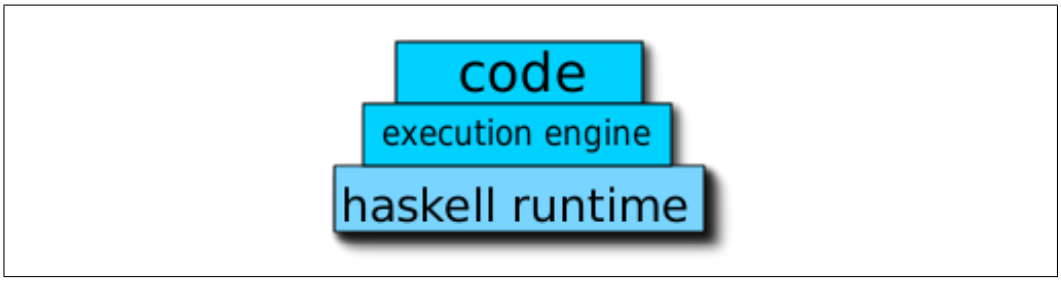

Figure 5.2: The Parser

5.2 Imperative and Declarative language

Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow [Practical Advantages of Declarative Programming] while imperative programming paradigm is mainly about statement and manipulation of states.

In this project, the goal is to design an imperative language.Therefore,the design and interpretation of statement and assignment is the main focusing point in this project. In addition,sub-function definition and function invocations support are also considered in this project to provide a certain level of abstraction.The table below has listed the fundamental statements.

Code	Description
a = 1	assignment statement
a = fib ()	assignment statement with function invocation
while ()	while block statement
if ()	if block statement
for (; ;)	for block statement
fib(num1)	function invocation
break	break loop control statement
continue	continue loop control statement

5.3 Type System

5.3.1 Dynamic Typing and Static Typing

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. For example , you have to specify the type explicitly and the compile will the the type correctness of a variable.A variable of a specified type can not assign to another value of other type.

Static Typing	Dynamic Typing
int a =1; **/*a is of type int */** int a="a string"; **/* its not valid to assign a string to a variable that has type int */**	a=1; **/* does not need to specified a type for this variable */** a ="a string"; **/* it valid to change the type of the variable */**

In my project, I used the dynamic typing scheme,which is ,does not need to specify any type of variable and able to assign any type of primitives to a variable.

5.3.2 Strong Typing and Weak Typing

a language is said to be strong typing is that it place restriction in operation where data type can not be intermix.

Strong Typing	Weak Typing
a=123; **/* a is a number */** b="123" **/* b is a string */** c=a+b **/* return type error */**	a = 123; b = "123"; c = a+b; **/* either a will be convert to a string or b will be convert to a number */**

In my project,I have implemented a weak typing system .I have design an statement call generic expression , which allow different kinds of value to intermix with each other. An expression like "12343" + 1232 -324 can be parse as follow syntax tree.

5.4 Problem and resolution in writing BNF/EBNF rules

For a parser , there are four operation it will do when encounters a terminal/non-terminal symbol.They are,

- Shift - push token onto stack
- Reduce - remove handle from stack and push on corresponding nonterminal
- Accept - recognize sentence when stack contains only the distinguished symbol and input is empty
- Error - happens when none of the above is possible; means original input was not a sentence

Conflicts arise from ambiguities in the grammar when two or more operations and rules that apply to the same sequence of input.[A final solution to the Dangling else of ALGOL 60 and related languages]

5.4.1 Shift//Reduce Problem

A shift conflict occurs if there are two or more rules that apply to the same sequence of input for the same operation reduce. This usually indicates a serious error in the grammar.

5.4.2 Reduce//Reduce Problem

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input for the same operation reduce. This usually indicates a serious error in the grammar.

5.5 Syntax Design

5.5.1 The Main Structure

A module is the minimum executable unit in **yun**.It is composed by one main function and serveral functions.The main function is a entry point,it may invoke other functions.

Possible program code may looks like follow,

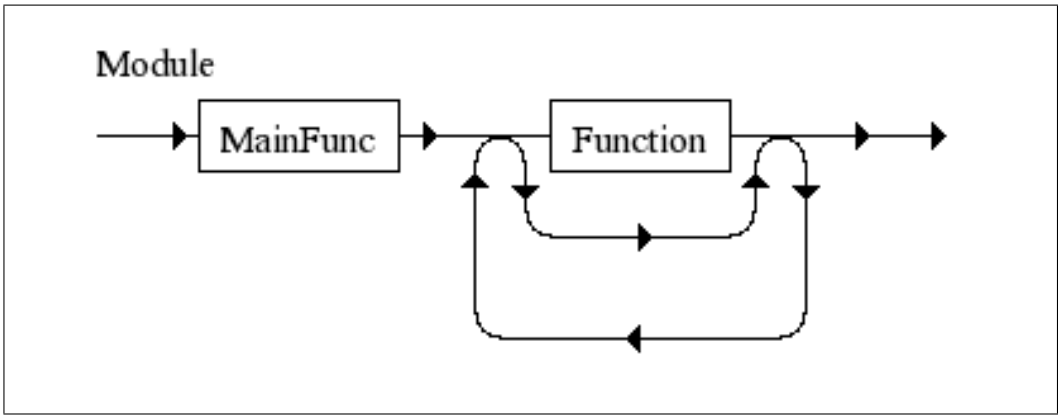

Figure 5.3: Module Syntax Diagram

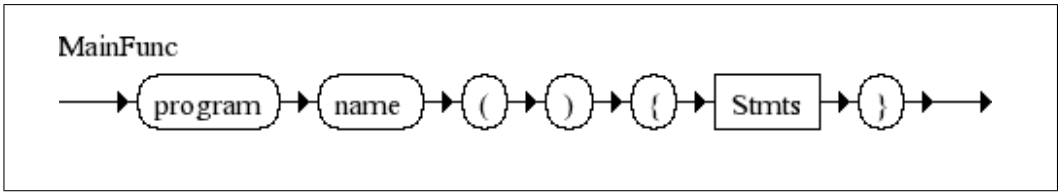

Figure 5.4: Main Function Syntax Diagram

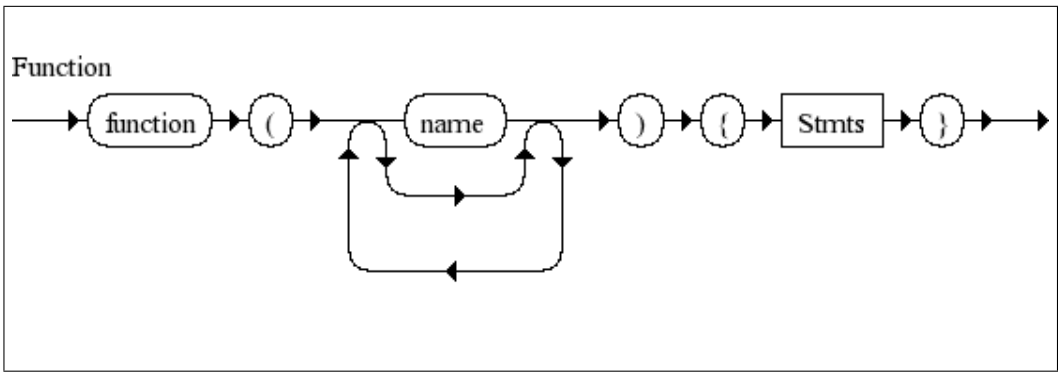

Figure 5.5: Function Syntax Diagram

```
1 program main ()
2 {
3         result = sum(1,2,3);
4         // more code
5
6 }
7
8
9 function sum (var1,var2,var3)
10 {
```

```
11          // body of the functions
12 }
13
14 // may be more functions
```

5.5.2 Statements

Statements comprise the body of a function.A statement may be a assignment,break and continue sentence,return sentence.WhileBlocks ,IfBlocks,ForBlocks are all statements.What's more , WhilcBlock,IfBlock,ForBlock are recursively defined by statements as well.This allow nested for loop ,nested while loop and etc.

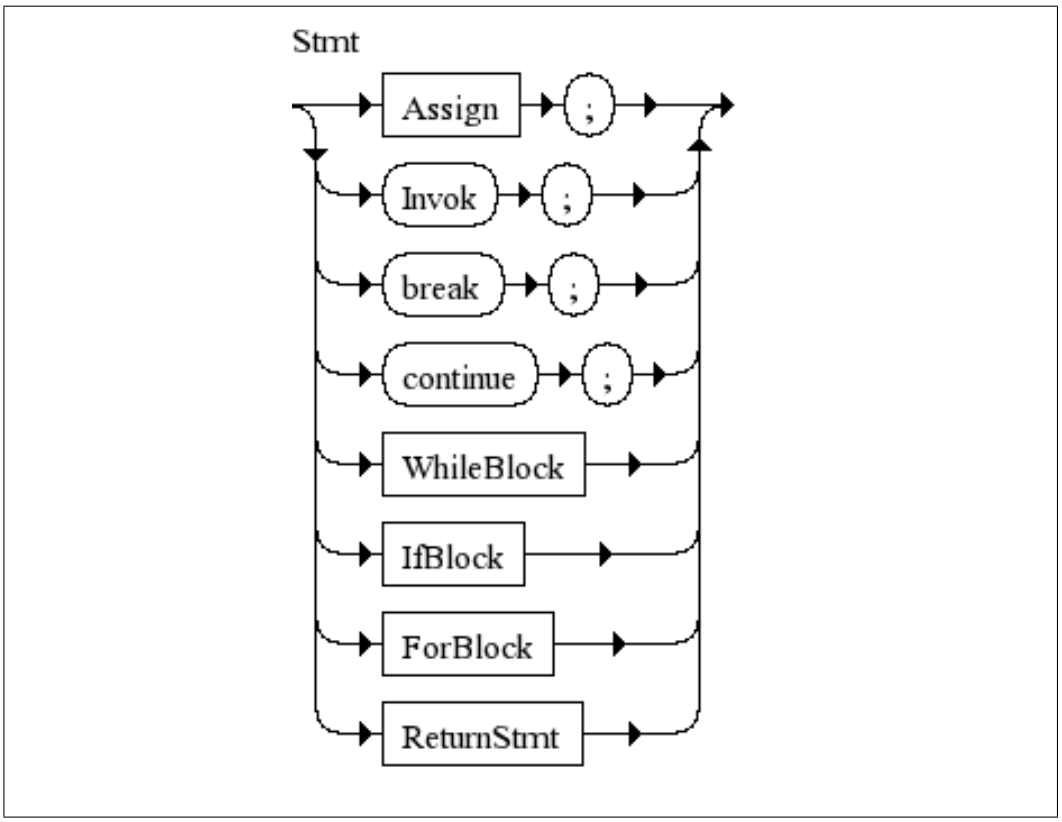

Figure 5.6: Statement Syntax Diagram

The following code can be consider as a statement of the language

```
1 while ()
2 {
3 } /* the while statement */
4
```

```
5 if ()
6 {
7 }
8 else
9 {
10 } // the if else statement
11
12 for ( )
13 {
14         for ()
15         {
16         }
17 } // nested for loop
18
19
20 a = 1 ; // assignment statement, assign a value or and an
       expression to a variable
21
22 fib(5);  // function invocation statement
```

Assignment

There are three type of assignment in the language .They are ,

- Function invocation assignment.Assign the return result to a variable.
- Generic expression assignment.Assign the result of an expression to a variable.
- List assignment.Assign a list to a variable.

As **yun** is an imperative language,the right side of the assignment operator will be evaluation immediately. In other word,the language is strict.

5.5.3 Generic Expression

Generic expression represent the computation between primitives.As mention above, **yun** is a weak typing and dynamic typing language,the generic expression accepts variant types of primitive and will do the conversion internally.

Valid expression could be ,

```
1 "string"
2 1
```

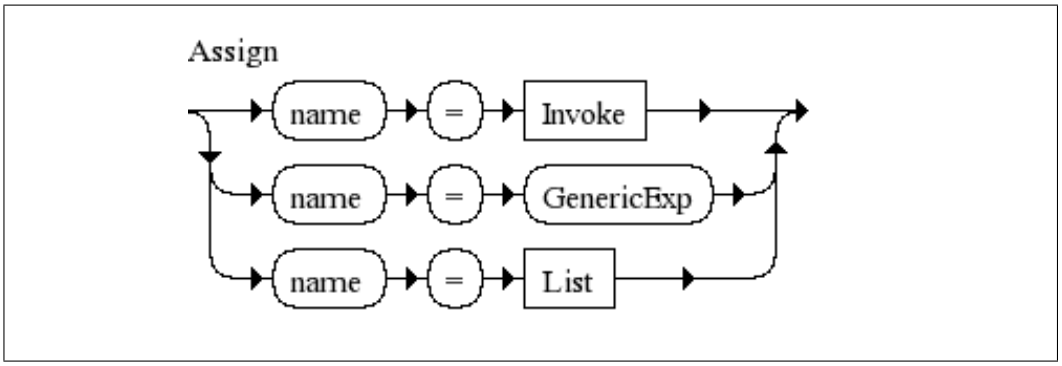

Figure 5.7: Assign Statement Syntax Diagram

```
-10
1.1
True
False
True
False // primitives

1 + "a string"
1 * 2 + 1
a && b
a || b
! a
a[2] //get the third element of a list
a[1,2]  // get the element in a multi dimension list

(a+b)+c // operation between variables
a[num]
```

List

The language support polymorphic list,the element of a list can be an expression or an list,thus the list support multi dimension list.If the element is an expression,the interpreter will evaluate it and store its result value in its internal format.

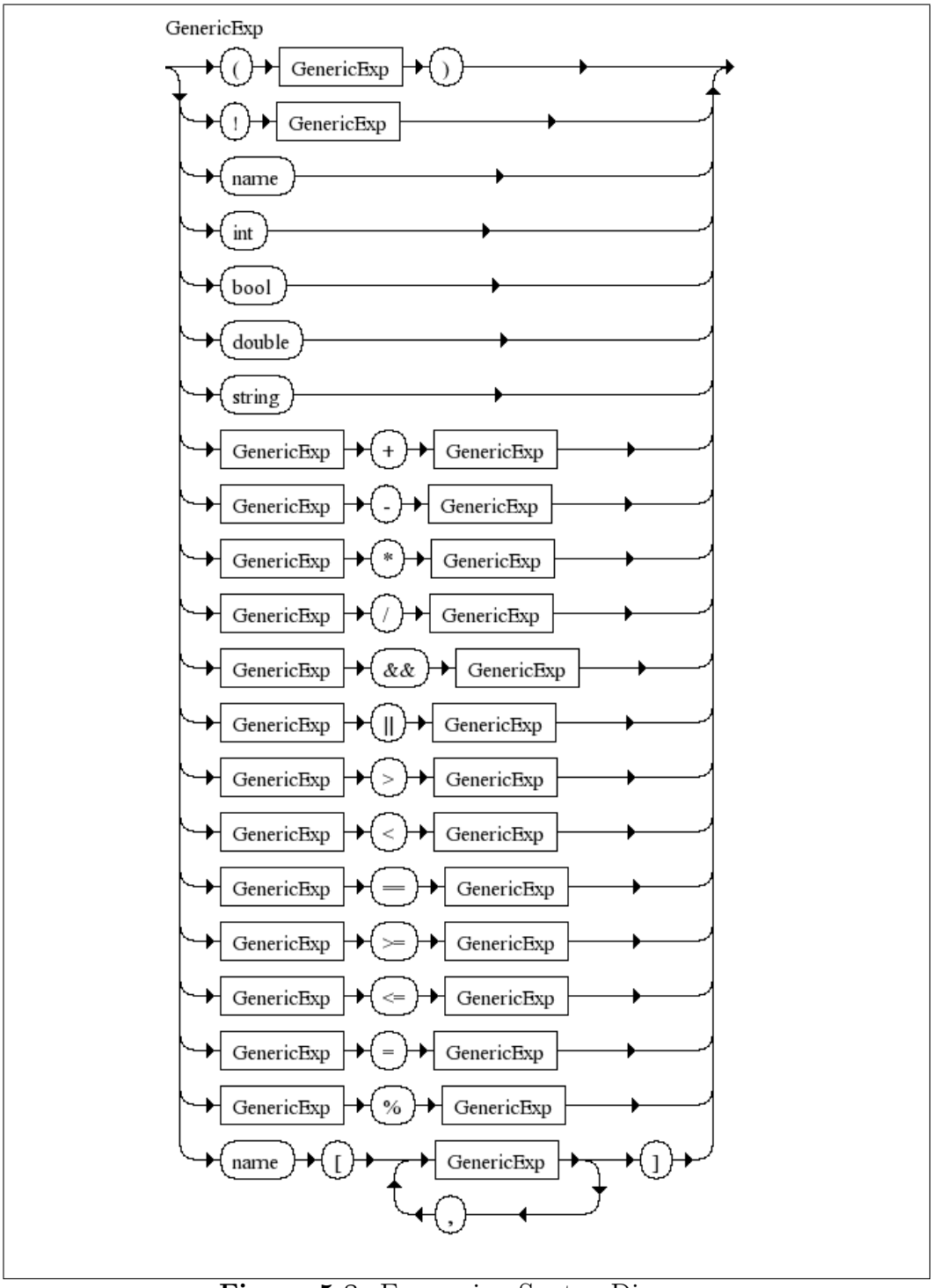

Figure 5.8: Expression Syntax Diagram

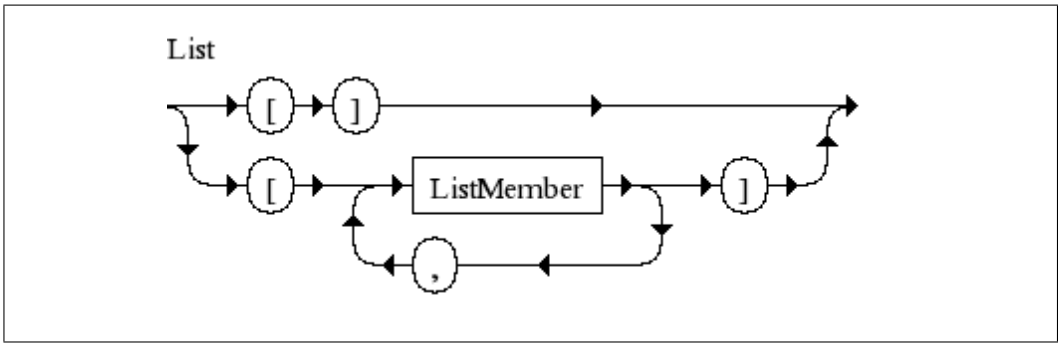

Figure 5.9: Expression Syntax Diagram

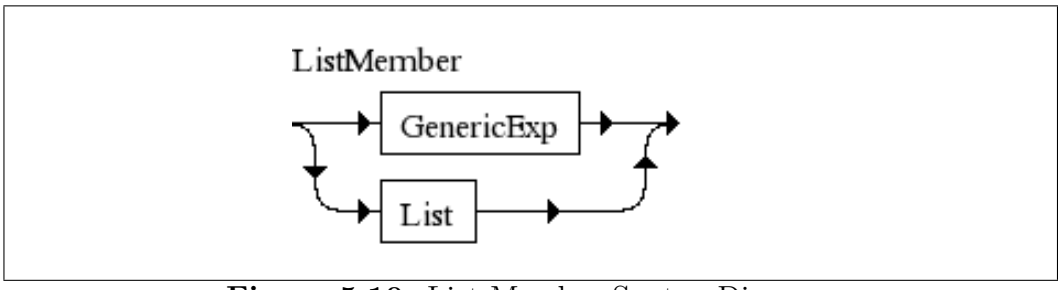

Figure 5.10: List Member Syntax Diagram

Code Example

```
1 a = [1,2,3];
2 a = ["string",1,2,3];
3 a = [var1,var2,"other",var1+var2,True&&False];
4 a = [[1,2,3,4,5,6,7] , 1, 23.32, 5];
5 a = [1,2,3,4,5],[1,2,3,4],[3,45,43]]; // declare a multi
    dimension list
6
7 member = a[1] // get the second element of a list
```

5.5.4 Code Block

Code blocks including if block,while block and for block are statements.Code block may contains other statements which allow nested code block to be defined.

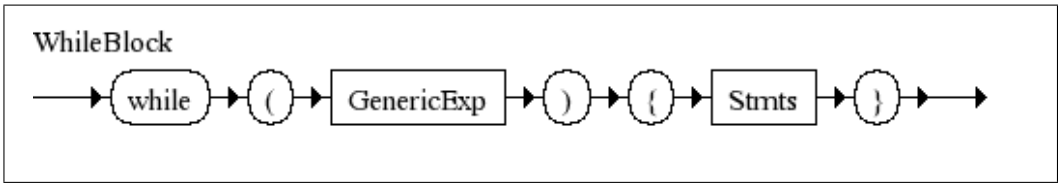

Figure 5.11: While Block Syntax Diagram

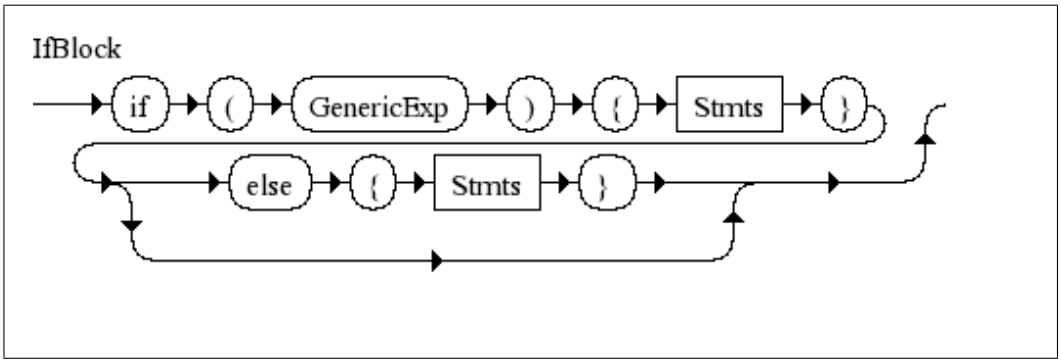

Figure 5.12: If Block Syntax Diagram

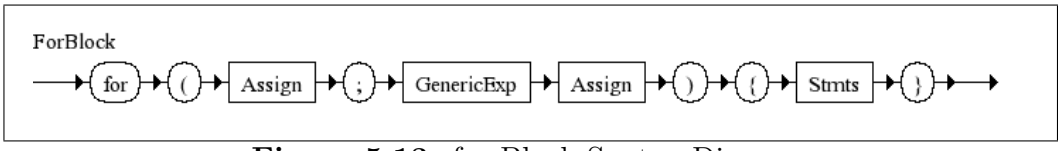

Figure 5.13: for Block Syntax Diagram

Code Example,

```
1  if (a < 1){
2          a = a+1;
3          if ( a == 1){
4                  a = a-1;
5          }
6  }
7  else{
8          a = a-1;
9  } // nested if loop
10
11
12 while (a<10){
13         a= a+1;
14 } // while block
15
```

```
16
17 for (i =0;i<10;i=i+1){
18             for (j=0;j<10;j=j+1){
19                         sum=j+i;
20             }
21 }//nested for block
```

Chapter 6

Language Implementation

6.1 Data Type

6.1.1 Tokenizer Type

All tokens are define using the Haskell Lexer Generator Alex.Tokens will be recognized and converted to Haskell data types.For example , a valid variable is comprised with one or more alphabetic characters and digit and the first character must be an alphabetic characters. This rule can be defined as ,

```
$digit = 0-9                        -- digits
$lowerCase = [a-z]
$alpha = [a-zA-Z]                   -- alphabetic characters

tokens :-
  $alpha [$alpha $digit \_ \' \.]* { \s -> TName s }
-- data definition
data Token =
        TName String |
        TBool Bool  |
        TInt Int  |
        ...  -- more definition
```

The variable **var1** will be parse into Haskell data type **TName "var1"**

By using the lexer , all the code will be generated into tokens streams.

```
program myProgram ()
{
        var1 = 1;
        result = var1 + "a string";
}
```

The following codes show the invocation of the lexer and the returned tokens stream.

```
lexer "program myProgram() { var1 = 1; result = var1 +
    False; return 0; }"
[TProgram,TName "myProgram",TOPB,TCPB,TOCB,TName
    "var1",TAssign,TInt 1,TSC,TName "result",TAssign,TName
    "var1",TPlus,TBool False,TSC,TReturn,TInt 0,TSC,TCCB]
```

6.1.2 Parser internal data type

The Parser internal data type typically represents a parse tree.To represent a generic expression,we can define data type as follow,

```
data GenericExp  =
                    Var String
                  | Int Int
                  | Double Double
                  | Plus GenericExp GenericExp
                  | ListIndex String [GenericExp]
                  | Brack GenericExp  deriving
                  ... -- more definition
```

The following code

```
( a +1 )  /=  3
myArray[ 1+1,2]
```

are parsed into Haskell data structure,

```
NotEq  (Brack ( Plus (Var "a") (Int 1)))  (Int 3)
ListIndex "myIndex" [ Plus (Int 1) (Int 1) , Int 2 ]
```

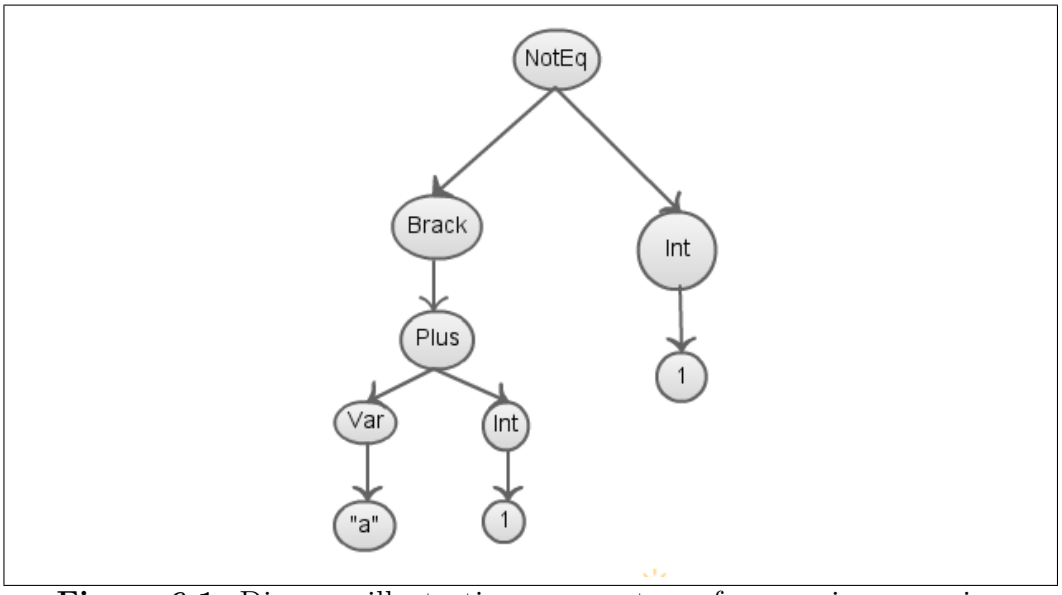

Figure 6.1: Diagram illustrating a parse tree of a generic expression

In Haskell,a parse tree data structure that support polymorphic list can be defined using recursive data type as follow,

```
data List = ListGenericExp GenericExp
                        | ListList [List] deriving (Show,Eq,Read)
```

A polymorphic list,

```
1  [1,2,["string",2,3]];
```

Can be represent in Haskell data type.

```
ListList  [ListGenericExp (Int 1),ListGenericExp (Int 2),
        ListList [ListGenericExp String "string",
                ListGenericExp (Int 2),ListGenericExp (Int 3)]]
```

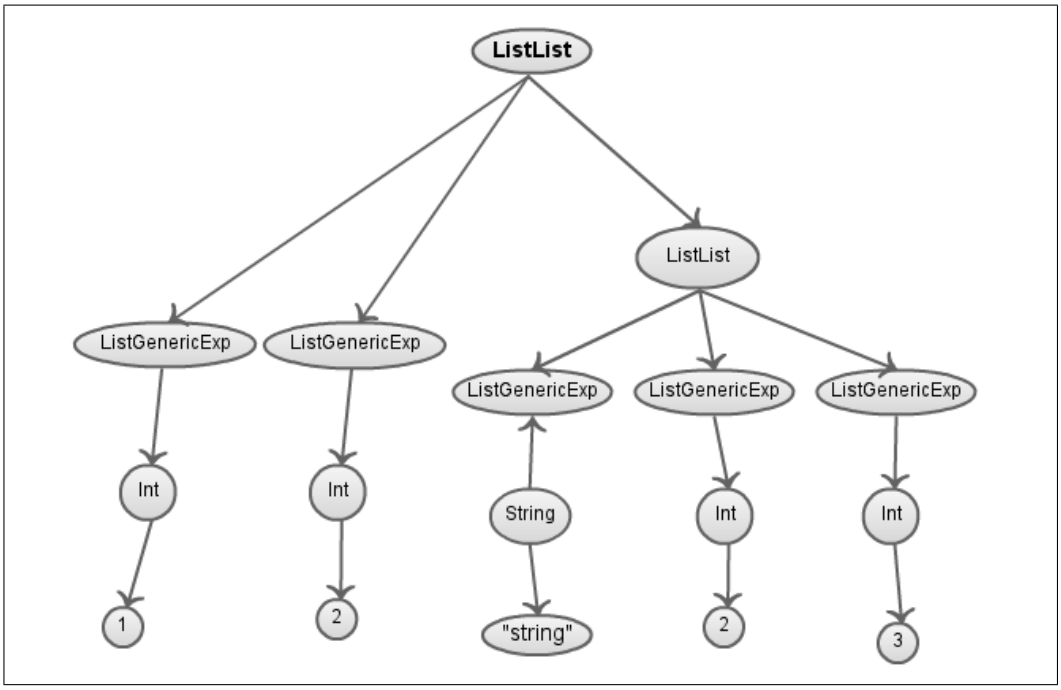

Figure 6.2: Diagram illustrating a parse tree of a list expression

6.1.3 Interpreter Internal Data Type

The data type defined as below shows that the possible data type of result that a sub-interpreter will return.There is a interpreter correspond it.The sub-interpreter will return for example an interger to represent the result of a generic expression.The recursive data type is use for a the purpose of nested list support.

```
data Value =  Int Int
            | Bool Bool
            | Double Double
            | String String
            | List [Value]
            | NULL    deriving Show
```

The Interpreter Monad

The interpreter monad is a combination of Error Monad ,State Monad and IO monad.

```
type Interp a  = (ErrorT String (StateT Storage IO )) a
```

```
type SymbolTable = M.Map String Value
type Storage = (SymbolTable,[Func])
```

For each sub-interpreter ,it must accept an expression and return a **Interp** type.The **Interp** type is a monad constructor that offer encapsulation support to return value.One of the **StateT** parameter is the type **Storage** which represents the internal storage of an interpreter like symbol table and parse tree of sub-function.

6.2 Module Evaluator

Module is the minimum executable unit in **yun** programming language.A module is comprised by one main function and multiple sub functions.What the Module Evaluator will do is ,first initial the symbol table using a empty map, second, extract the parse tree data and put in two together with the symbol table,third,extract all statements in main function and hand it over to statement evaluator.This Evaluator will not check the error message from other sub-evaluator it invoked,all the error will be passed to the main function.

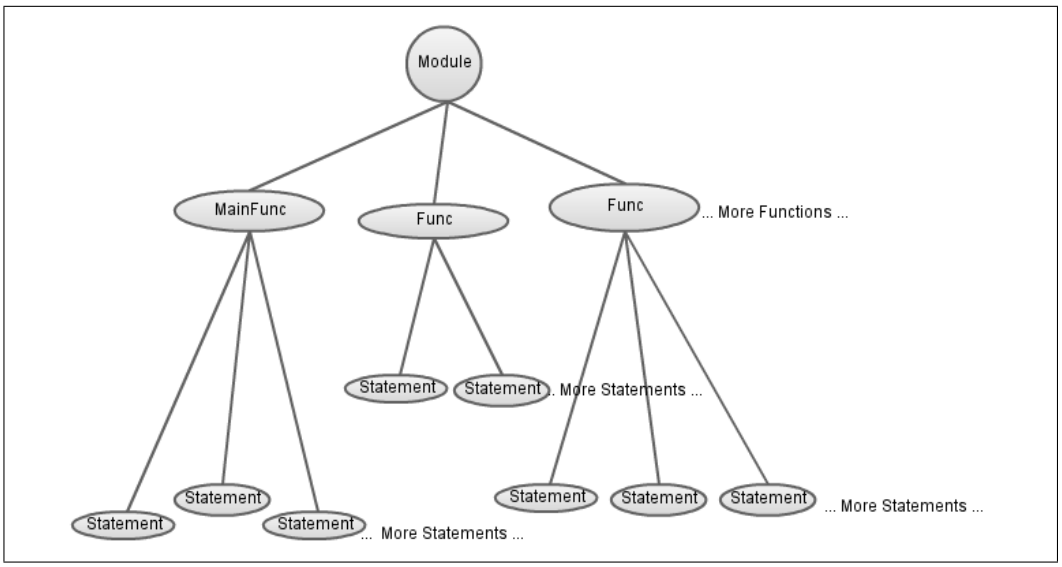

Figure 6.3: Diagram illustrating a parse tree of a module

6.3 Statement Evaluator

```
data Control = Break | Continue | DoNothing | Return Value deriving (Show)
```

```
stmtsEval :: [Stmt] -> Interp Control
stmtsEval --   function body

stmtEval :: Stmt -> Interp Control
stmtEval  -- the function body
```

Stmt is a data type that represents a Parse Tree of a statement.The first statement evaluator will accept a list of statement and for each statement,it invoke the statement evaluator.By applying patter matching,the statement will do correspond action toward each type of the statement.For example, for the **break** or **continue** statements,it will return the corresponding control command.The diagram below illustrate the how the statement evaluator works.

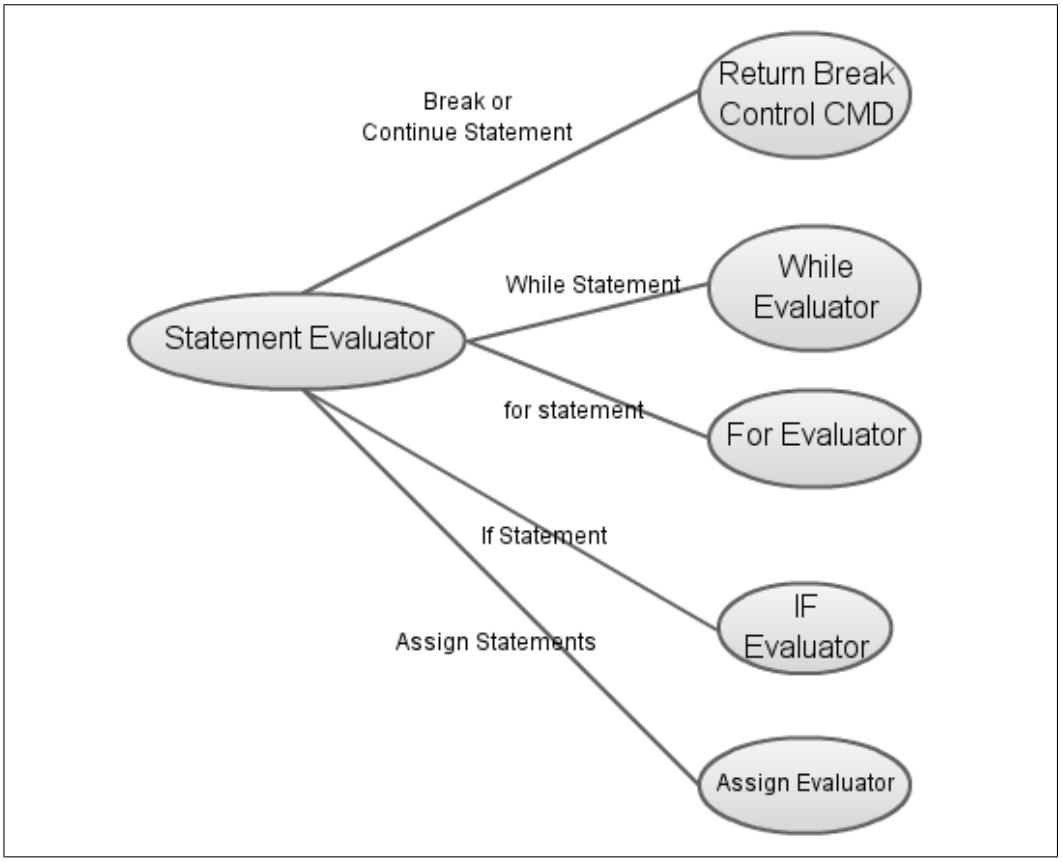

Figure 6.4: Diagram illustrating guard condition and function invocation

6.4 Symbol Table and Parse Tree

The symbol table together with the parse trees of sub-functions as a tuple will be stored behind the state monad.Symbol table is a map that mapping from a string to the type **Value** mentioned above.Monadic function composed to support add ,update from the symbol table.

```
type SymbolTable = M.Map String Value
type Storage = (SymbolTable,[Func])
type Interp a  = (ErrorT String (StateT Storage IO )) a
```

6.5 Generic Expression Evaluator

The generic evaluator supports evaluation of internal data type like string ,integer ,double and list.As **yun** is a weak typing language ,the generic expression will do the internal type conversions. For example,a expression like **"string" + 1** will be evaluator to **"string1"**.All value can be converted to a string generally,but not all the value can be convert to an integer.The follow expression **1+True** will cause an error.

Importantly,as weak typing languages have their own conversion rule,type conversion is somewhat unpredictable thus relying on type conversion is not recommended.

6.6 Function Invocation

Different from most of imperative language, **yun** only supports pass by value parameter passing.To invoke a function,it is needed to save the current scene and initialize the variable table for the function.

6.7 Main Function

The main function will load the source code, the it will invoke the parser which turns source code into a parse tree.Afterwards,the module evaluator will be invoke.

Chapter 7

Analysis,Conclusion and Future Work

7.1 Analysis

In the initial stage of this project,lots of possibility has been considered ,for instance,the parsec parser library and happy parser generator library.What's more, I also need to design which this programming language need to be a static type of dynamic type language.As there is lots of debate in pros and cons on various type system, I make the choice simple base on the difficult of implementation of a certain manifold.

The puerility and monad also help me a lot in writing this project.In this project,on a small amount of function is pure and most of other are monadic function as in this project,I made use of the monadic design approach.The monadic design approach helps me to limit to side-effect into monad and I can specify to limit the type of side-effect.

Above all,Haskell is a suitable language to do this project which is a important reason for the success of this project.

7.2 Conclusion

Haskell is a high-level language and provide a lots of tools to glue functions together thus can implements functionality elegantly.

Haskell's type system has placed numerous restrictions on my code and it turn out to be when I try to implements something new in project , most of

time, when I passed the compiling phrase I can have correct implementation.

In the passed three years, I have learnt quit few of programming language.By doing this project,I can be on a on the side of language design rather than a user, to review other languages' features and characteristic I have leant.It allows me that think about how a language is implemented and why it was implemented this way.

There are great amount of disputes on pros and cons between different programming languages.Those disputes, to a large extend, have ignored the fact that programming languages are created for computer scientist to resolve problem.

7.3 Future Work

This project can be extended to a Domain-specific language (DSL) which is a language designed delicately to particular program domain.Logo a dialect of Lisp and was created for education purpose.This language allow programming to draw shapes on the screen.To build a DSL,a research needed to be conduct for a specified domain ,additional syntax support and library needed to be added into the language.

With Haskell powerful support abstraction support like monad , high-order function and function composition, implementing a language is not so painful. Though the current project is only focus on a generic purpose language , other people has implemented lots of Domain-specific language hosted by Haskell runtime,for instance ,KURE is a Haskell hosted Domain-specific Language (DSL) for writing transformation systems based on rewrite strategies[A Haskell Hosted DSL for Writing Transformation Systems].

A DSL is created for resolving problem that is easy to design and tedious to implements and Haskell is the best candidate to do this job.