Implementing an interpreter for a scripting language using Haskell

Final Year Project Report

Zhen Lao

zhen.lao@student.dit.ie

Supervisor: Richard Lawlor

2nd Reader: Cindy Liu

May 11, 2011

Abstract

Haskell is purely functional programming language.Programmer can build larger program from small modular and building block.In thesis,we will talk about the process and technologies to build a scripting programming language called **yun** using Haskell.Monad and monadic transformer are to fundamental concept for us to build the programming we will be discuss in chapter 4-5.

Keywords:programming language,monad,monad transformer,functional programming,**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 scripting 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 does not support object-orient paradigm.

1.2 Challenges

As I am using Haskell, the functional programming to do my project,the first challenge is that I have to abandon the object-oriented and imperative analysis to model design my code.The second challenge is that I have to learn a new language of a completely different paradigm before I can actually start working on the project.Learning how a compiler or interpreter work is also a major challenge of this project

1.3 Benefits of using Haskell

Haskell is a purely functional programming that provide promotes a more abstract style of programming,based upon the idea of applying functions to argument. [17] .By applying the used of Haskell to this project ,I have significantly reduce the coding time and spent most of my time in the design phrase.

A pure function is a function that accepts an input and produce an output.In Object-Oriented language,programs are 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. [16] [18]

1.4 Development Methodology

Iterative and incremental 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.

The iterative and incremental development is a cyclic development process. It's also the heart of rational unified process, Extreme programming and agile software development. However, I would like to used the iterative and incremental approach in my project rather a completed agile development or RUP cycle for the reason that it is not possible to fulfil the requirement of a completed industrial software development methodology. For instance, it's hard to visualize my architecture using the UML by following RUP guide line in my project.

Researches have been done on investing some useful Haskell library like Parsec ,Happy,Alex and HUnit which will be discussed in chapter 2.

Eclipse with EclipseFP plug-in provides support for source code and package management in functionality.To track the development process,I make use of **git** a source control tool to version all the source files.

1.5 Development Environment

All the development will be based on the Haskell platform.The Haskell Platform is a comprehensive, robust and cross platform environment for Haskell developer. The Haskell platform contains the following component:

1. Glasgow Haskell compiler (GHC) and its runtime environment.

2. Cabal.An package management system.

3. Haddock.A document tool like javadoc to jdk.

4. The GHCi debugger.An interactive, imperative-style debugger for Haskell.

The platform can be installed in different platform include Windows,Linux and Mac OS X.I use Ubuntu Linux 10.10 as my development platform.

The Haskell platform made it convenient to manage package dependency since its release in 2010,while at the early 00s,the motto of Haskell community was "avoid success at all cost" [21].

1.6 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 like monad.Most of them 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 is 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

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.

[2] [3] 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 The Hierarchy of Grammars

Noam Chomsky has described three models of grammar [3] and these grammar models have significantly affected the design of computer programming language.

Chomsky defines a set of rules 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).

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 production).

4. A start symbol, which is a special non-terminal symbol that appears in the initial string generated by the grammar.

[2] it can be recognized by pushdown automaton.

Content-free grammar can represent the algebraic expressions.The table below specify a set o rules:

Rules	Description
$1. S \rightarrow number$	S produce a number, may be a integer or a float ,defined by regular expression
$2. S \rightarrow boolean$	produce a boolean value
$3. S \rightarrow string$	produce a string value
$4. S \rightarrow S + S$	adding of two S expression
$5. S \rightarrow S - S$	substrate of two S expression
$6. S \rightarrow S\%S$	mod of two S expression
$7. S \rightarrow S > S$	logical greater of two S expression
$8. S \rightarrow S\&\&S$	logical and of two S expression
$9. S \rightarrow (S)$	bracket of the S expression
$10. S \rightarrow S == S$	logical and of two S expression

This grammar can, for example, generate the string

$$((a + 2) == (a * 2))\&\&(a > 0)$$

can be derived from the projection rules as follow,

S	the start symbol
S && S	rule 8
(S) &&S	rule 9
(S==S) && S	rule 10
((S)==S)&& S	rule 9
((S+S))==S)&& S	rule 4
((S+S))==S)&& (S)	rule 9
((S+S))==(S*S))&& (S>S)	rule 7
((a+S))==(S*S))&&(S>S)	rule 3
((a+2))==(S*S))&&(S>S)	rule 1
((a+2))==(a*S))&&(S>S)	rule 3
((a+2))==(a*2))&&(S>S)	rule 1
((a+2))==(a*2))&&(a>S)	rule 3
((a+2))==(a*2))&&(a>0)	rule 1

2.1.3 Backus–Naur Form and Extended Backus–Naur Form

The Backus-Naur Form(BNF) is a metalanguage to write production rules that express 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 look like :

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

The left side of the equation can only be non-terminal thus enclosed with $<>$ .The right side can be terminals and non-terminals,a vertical bar '—' is used to represent choice between terminals and non-terminals.

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

[2]

Recursive rules of BNF like

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

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

that expressing a sequence of a particular syntactic elements can be simplified using quantifiers in EBNF to $< exp >:= sub+$

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. [8]

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

2.3   Monadic Parsing using Parsec

In the early stage of this project,parser is built using **ParseC**, 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 [2]) grammars. Combinator parsing is well known in the literature and offers several advantages to YACC or event-based parsing. [23]

LL grammars have the distinctive properties that they are non-ambiguous and not left-recursive. A grammar is LL(k) if the associated predictive parser needs at most k tokens lookahead to disambiguate a parse tree. [24]

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. [6]

2.5   Testing and Evaluation Technologies

There are two type of testing in Haskell.The type testing and unit testing.There correspond to two package in Haskell,QuickCheck and HUnit.

QucikCheck

For pure functions programmer can write logical expression or functional specification to prove the code is correct ranter than provide a specify test case as an input of the function. One of the example to test a function that takes the first five element of a list is using a length function $if \ \forall \ s. \ length(take5 \ s) = 5$ is true,then the $take5$ function is probably true. The QuickCheck package allow us to write the assertion in Haskell code this way.

```
quickCheck (\s -> length (take5 s) == 5)
```

HUnit

HUnit is a unit testing framework for Haskell to test the impure code.it is similar to the JUnit tool for Java.In unit testing,it is required to provide an input and verify the output.
Test specification in HUnit is even more concise and flexible than in JUnit, thanks to the nature of the Haskell language. HUnit currently includes only a text-based test controller, but the framework is designed for easy extension. [12]

Chapter 3

Haskell

3.1 History

Lambda calculus was introduced by Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics. [4] In 1936 Church isolated and published just the portion relevant to computa- tion, what is now called the untyped lambda calculus. [5]

Following the release of Miranda by in 1985, interest in lazy functional languages grew: by 1987, more than a dozen non-strict, purely functional programming languages existed. Of these, Miranda was the most widely used, but was not in the public domain. At the conference on Functional Programming Languages and Computer Ar- chitecture (FPCA '87) in Portland, Oregon, a meeting was held during which participants formed a strong consensus that a committee should be formed to define an open standard for such languages.[20]

The first version of Haskell ("Haskell 1.0") was defined in 1990. [14] The latest Haskell version describe in the Haskell report 2010,the implementation of the latest version Haskell Platform is released in Feb 2011. To summarize it ,Haskell has evolved continuously in recent years.

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

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 $f and g$,with $cod\ f = dom\ g$,a composite arrow $g \circ f : dom\ f \rightarrow cod\ g$, satisfying the following associative law:
   For any arrow $f : A \rightarrow B, g : B \rightarrow C, and\ h : C \rightarrow D$(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 $id_a : A \rightarrow A$ satisfying the following identity law:
   For any arrow $f : A \rightarrow B$,

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

[28]

3.3 Algebraic Data Type

The algebraic data type allow programming to define a data type that has more than one constructor. For example , a boolean value can be defined as,

3.4 Type Classes

3.5 Higher Order Functions

Chapter 4

Monads

4.1 Overview

4.1.1 History

In May 1996, the Haskell report 1.3 is published ,in this version , monadic IO as well as do-notation made its first appearance. [14]

In April 1997,the list comprehension is generalised into monad in the Haskell report 1.4. [14]

Nowadays ,a monad has become a de-facto pattern for Haskell programmer who wants to monad and write computation with side effect,and this pattern is explicitly supported by Haskell.

Pure languages are easier to reason about and may benefit from lazy evaluation, while impure languages offer efficiency benefits and sometimes make possible a more compact mode of expression. [29]

4.1.2 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. [19]

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. [19] Monads in Haskell have been used to model IO,state,logger,error as well as List.

4.2   Introduction to Monad

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. [13] 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.

Monad Laws

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.

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

4.3 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.

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) ))
```

4.4   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.
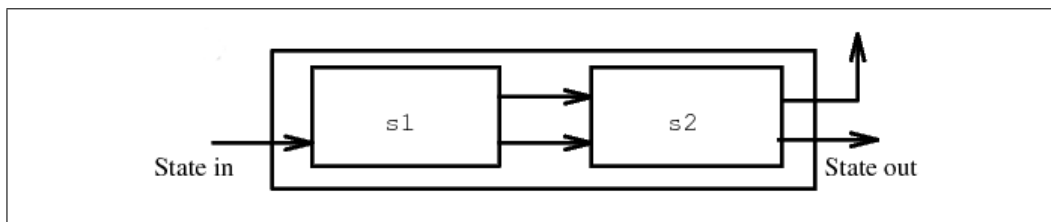

Figure 4.1: Combining Two "Stateful" Monadic Functions

4.5 List Comprehension and List Monad

4.5.1 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 in Haskell.

23

4.5.2   List Monad

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

Alternatively , we can view list as an monad.

```
instance  Monad []  where
  m >>= k  = concat (map k m)
  return x = [x]
```

The code above shows the definition of monad,the type construct is the list notation [],and the bind operator means that applying and monadic computation to a given list which is also a monadic data type. Equally,the list monad satisfy the three monad law ,namely ,left identity,right identity and associativity.
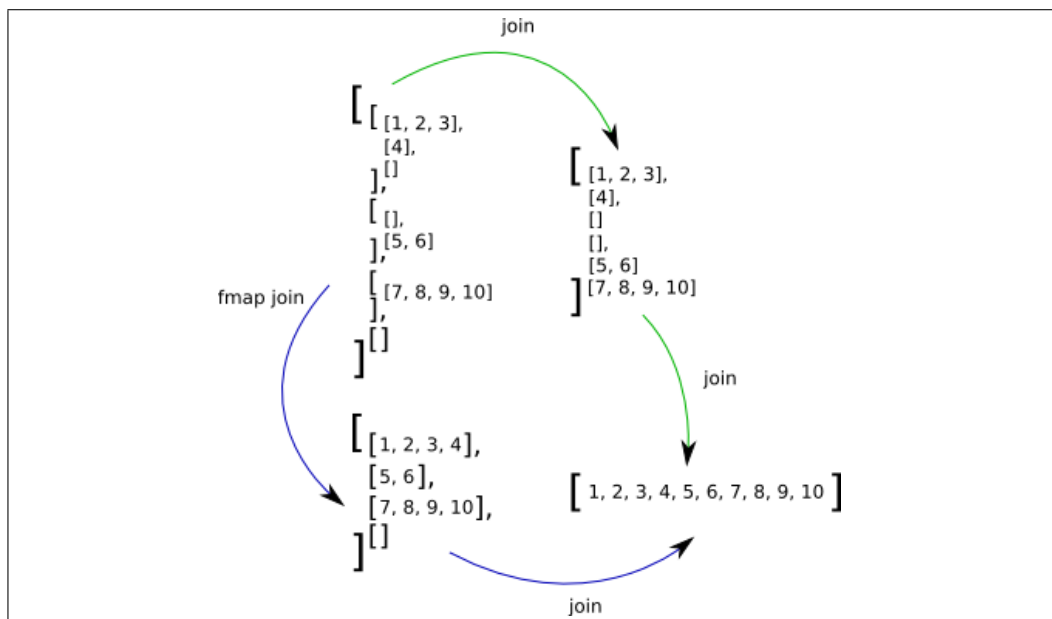

Figure 4.2: The First Monad Law illustration

4.6   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**.

24

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
```

The <- notation in this context means unwrap the content in the monad and put it in to a variable,for example ,a monadic function returns **IO String** ,the <- notation means unwraping the content of type String from IO container.

4.7   Error Monad

The error monad can models a computation which may fail or throw exception.The construct or a error monad is **Either a b** where a represent the type of error message and b represents the normal return value.

A left value will be returned if a computation or a combination of a computation fail whereas a right will value will be returned if succeed.The code below shows the bind strategy.

```
instance Monad (Either a) where
    return = Right
    x >>= f = case x of
        Left _ -> x
        Right r -> f r
```

From the above implementation ,we knows that for a computation x and f ,if the previous computation ,namely x has error,the second computation f will be abandoned,thus and error of x will be return.The first computation will be attempted only when the first computation succeed,the result of the first computation will be extracted from the right value and applied from the second computation. Finally ,we can know that, for a composition of computation which may fail,the earliest failure message will be recorded.

4.8   Other Monads

Other monads includes Maybe Monad ,Reader Monad,Writer Monads , are designed to model a certain type of computation.

The Maybe monad is similar to the Error,though its just a simplified version that can not return any useful error message.

Reader Monad are designed to model computation that read values from a shared environment.In fact,for computations that read values from outside word,it is possible that we pass the environment variable as the parameter to functions.But it would be simplified if we use the reader monad. The bind monad operation,which we can guess that will allow to computation to access that shared environment.

Equally,the Writer Monad are designed module computation that write to a shared configuration.As we can guess,the bind operator all both computation to write into the environment.

Chapter 5

Monad Transformers

5.1 Overview

As we already have monad designed to model side-effect with restricted type,it would not be too much problem to perform an action using monad in Haskell.New monad might be needed form operation that are able to perform state manipulation as well as IO action.

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. [9]

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.

5.2 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
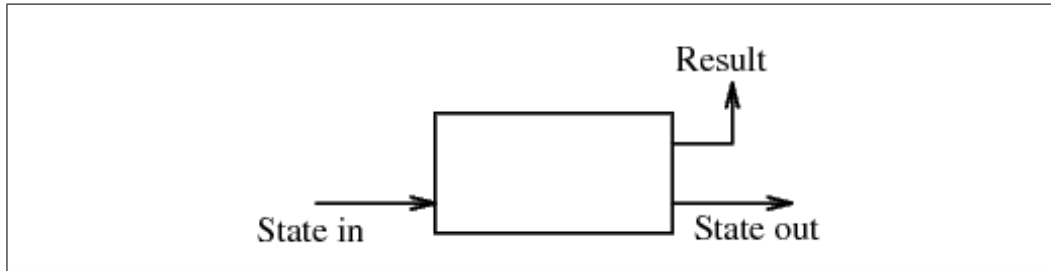
Figure 5.1: Diagram illustrating a state monad

[22]

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.

5.3 Error Transformer

ErrorT monad transformer can be used to add error handling to another monad. The ErrorT monad type constructor is ,

```
newtype ErrorT e m a
```

The first parameter e is the error type like the Either Monad constructor ,m is the inner monad,a is the type of the computation.

```
instance (Monad m, Error e) => Monad (ErrorT e m) where
    return a = ErrorT $ return (Right a)
    m >>= k  = ErrorT $ do
        a <- runErrorT m
        case a of
            Left  l -> return (Left l)
            Right r -> runErrorT (k r)
```

From the definition above , we can derive its bind strategy which is the most important thing for us to understand a monad.For two computation **m** and k,the bind strategy is that it will try to extract the result from the first computation which might return a error message ,the whole computation

28

will fail on the first computation which will attempt on the succeed of the first.computation.

The escape function **runErrorT** can help us to extract the inner level monad which is a computation with some kinds of monad logic and execute it.

```
getValidPassword :: ErrorT String IO String
getValidPassword = do s <- lift getLine
                      guard (isValid s)
                      return s
                            where isValid = (>5).length


askPassword :: ErrorT String IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- getValidPassword
                 lift $ putStrLn "your password is valid"

runProgram = runErrorT askpassword
```

The code above shows how a problem written by Error transformer **ErrorT** combines error handling as well as IO action.The runErrorT will extract the message which is either left value of right value.By combining **getValidPassword** together with lifted IO action,the logic that the string "your password is valid" will not return if user's input string is shorter than 5 has been implemented easily an elegantly because the bind operating itself implies failure values bypass the follow computation.

5.4   Combining Monads

Monad transformers create a new monad by extending an existing monad with additional operations, thereby allowing new programming language features to be de defined while preserving any language definition given for the original monad. [11]

As monad guarantees the correctness of the computation with the type restriction,namely we can not combine two computations one with "stateful" while other with IO logic into one building block,we might need addition functions like lift and liftIO together with monad transformer to fix the type problem .

As we can imagine that, if could combine every monad for most of monad if offer a monad transformer form.In this way,the action combine inside to notation would probably be no different from a real imperative language.When we combine all monads, the type restricted become slacked accepting a type of computation with side effect which is no difference from a real imperative language.

The only difference might be the sequence we encapsulate monad and monad transformers,that is we can only warp IO inside other monad transformers.

Chapter 6

Language Interpreter Design

6.1 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.

6.1.1 Parser

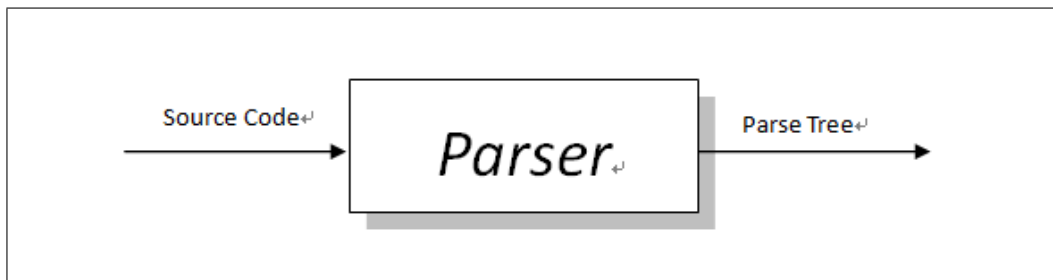the parser that the source code as input and generate a tree structure as output.

Figure 6.1: 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. [2]

6.1.2 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.
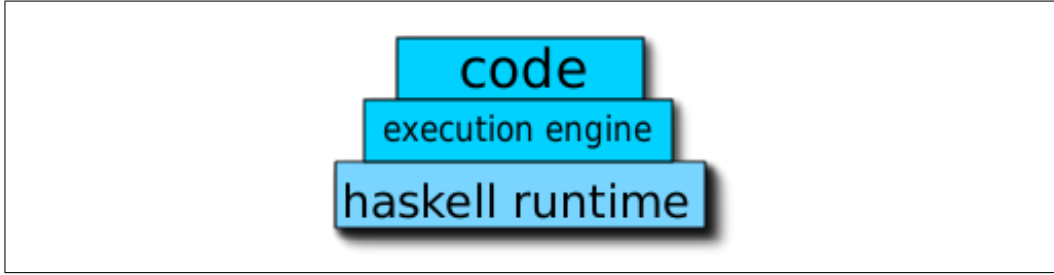

Figure 6.2: The Parser

6.2 Imperative and Declarative language

Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow [26] 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

6.3 Type System

6.3.1 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.

6.3.2 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.

6.4 Problem and resolution in writing BN-F/EBNF rules

For a parser , there are four operation it will do when encounters a terminal/nonterminal 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. [1]

6.4.1 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.

6.4.2 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.

6.5 Syntax Design

6.5.1 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.
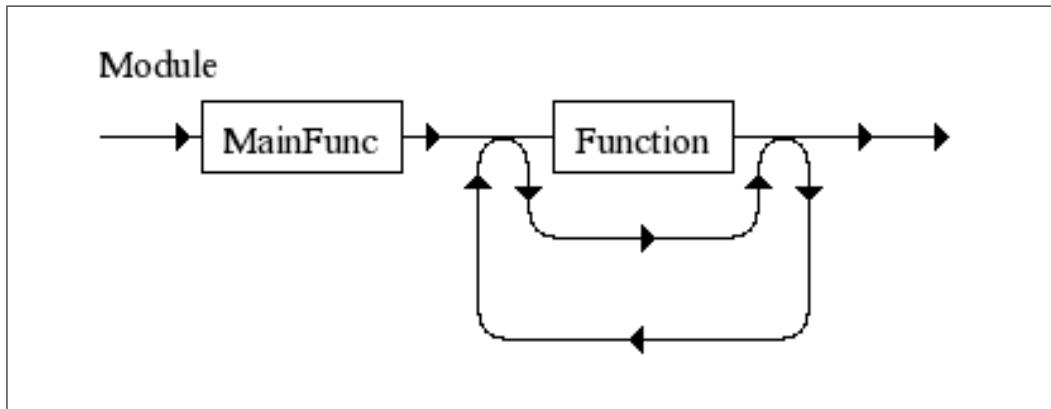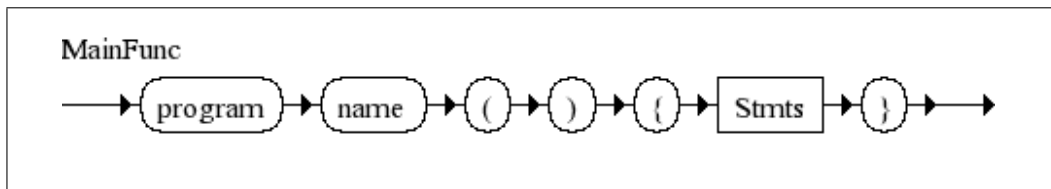
Figure 6.3: Module Syntax Diagram

Figure 6.4: Main Function Syntax Diagram
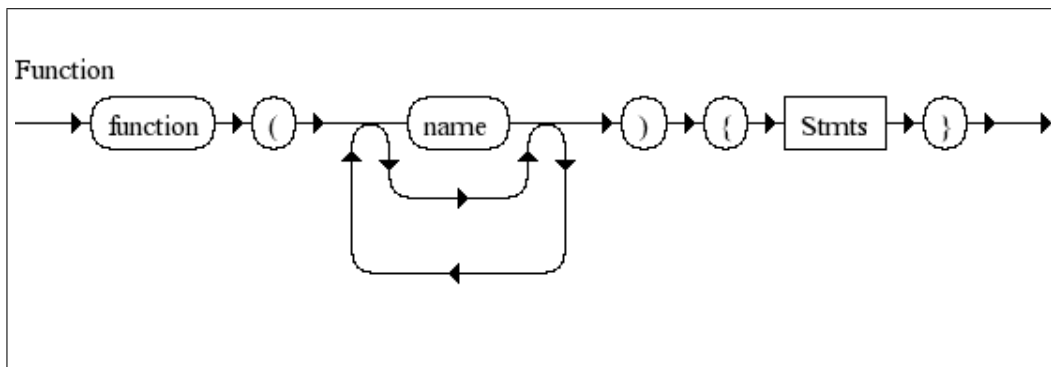

Figure 6.5: Function Syntax Diagram

Possible program code may looks like follow,

```
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
```

6.5.2 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.
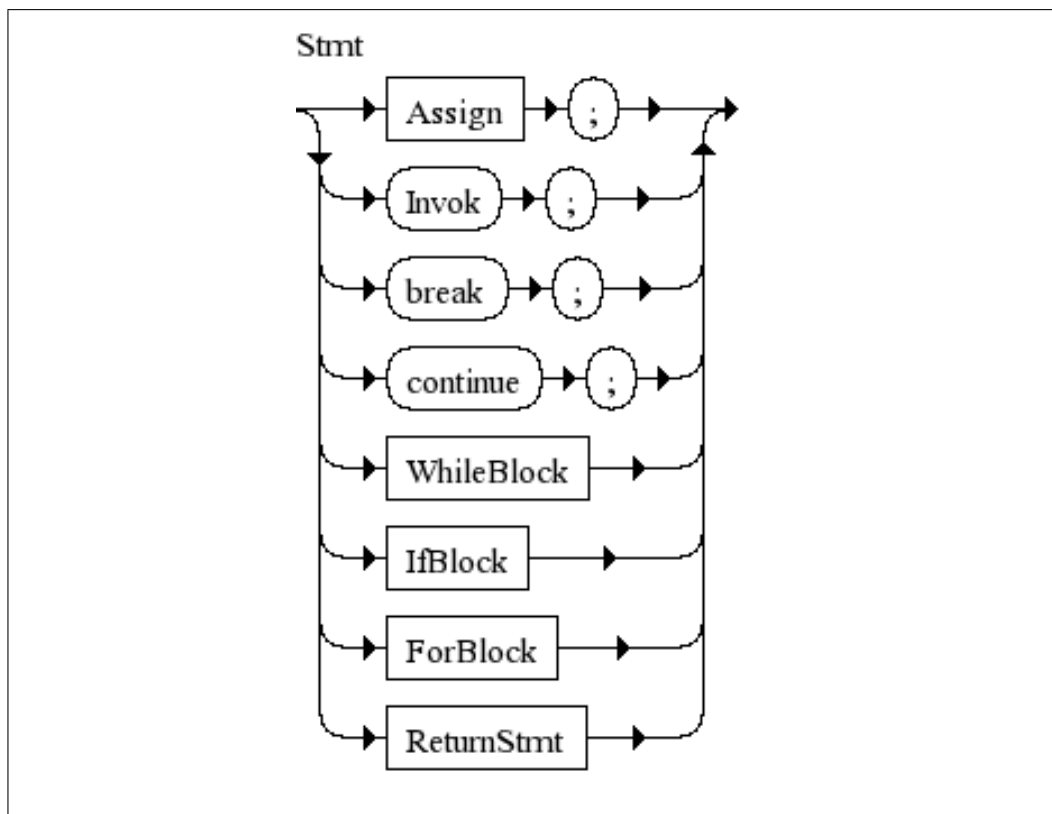

Figure 6.6: Statement Syntax Diagram

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.

6.5.3 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.
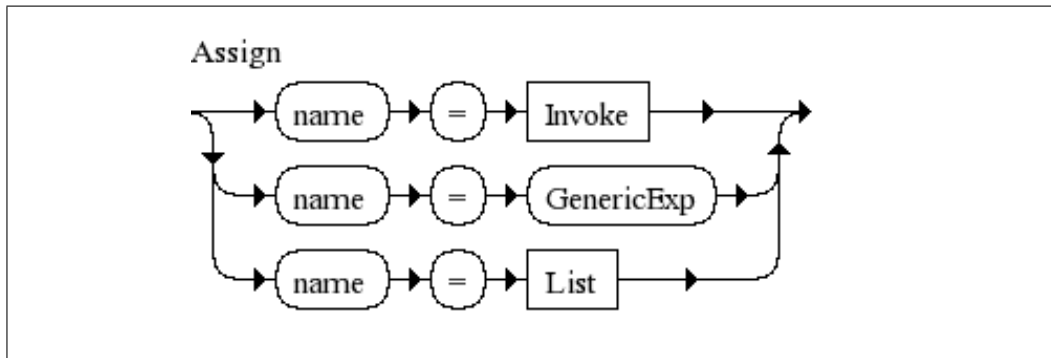
Valid expression could be ,

Figure 6.7: Assign Statement Syntax Diagram

```
1  "string"
2  1
3  -10
4  1.1
5  True
6  False
7  True
8  False  // primitives
9
10 1 + "a string"
11 1 * 2 + 1
12 a && b
13 a || b
14 ! a
15 a[2] //get the third element of a list
16 a[1,2]  // get the element in a multi dimension list
17
18 (a+b)+c // operation between variables
19 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.
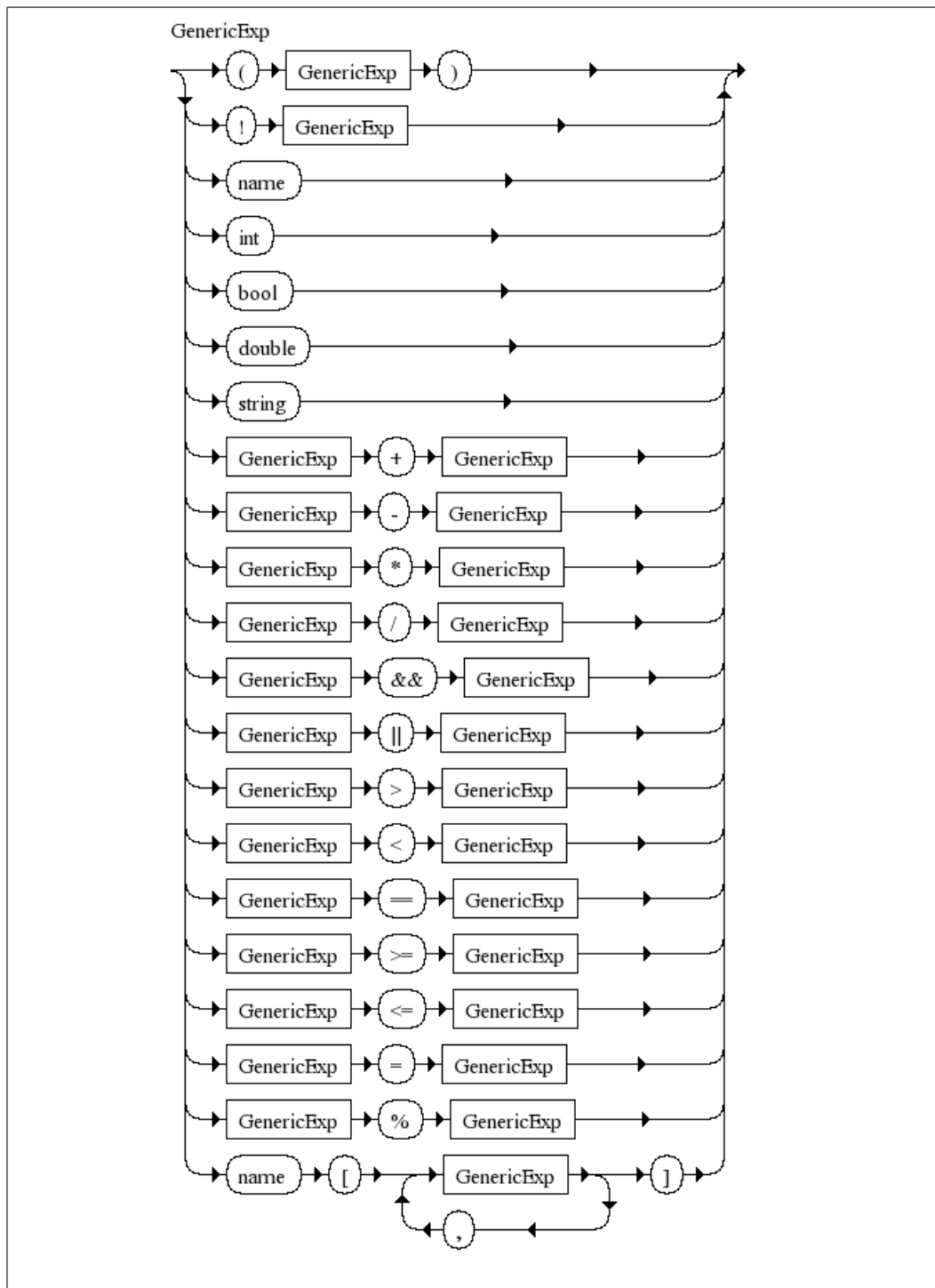
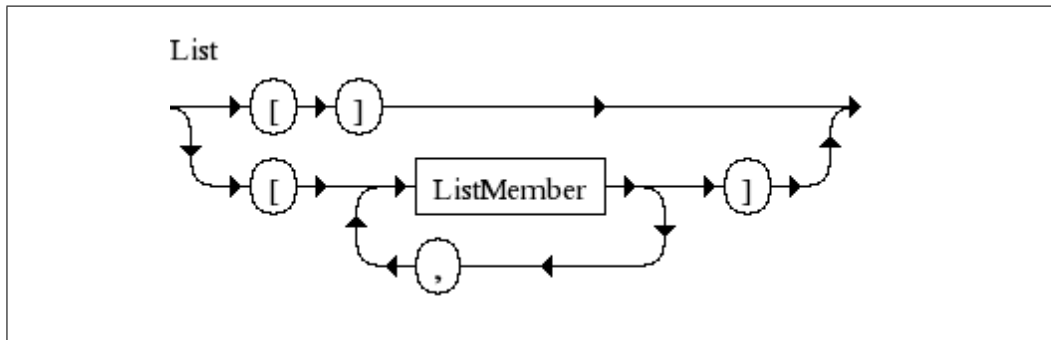**Figure 6.8:** Expression Syntax Diagram
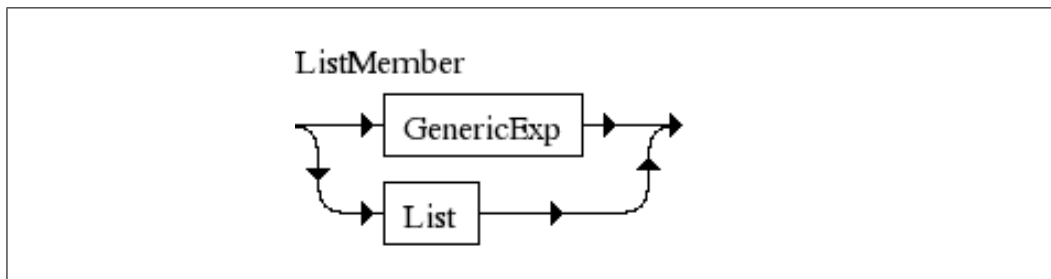
Figure 6.9: Expression Syntax Diagram

Figure 6.10: 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
```

6.5.4 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.
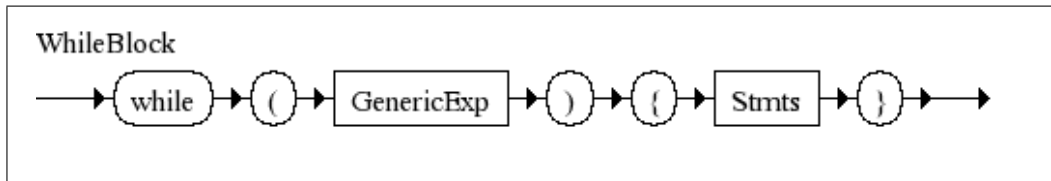
Figure 6.11: While Block Syntax Diagram
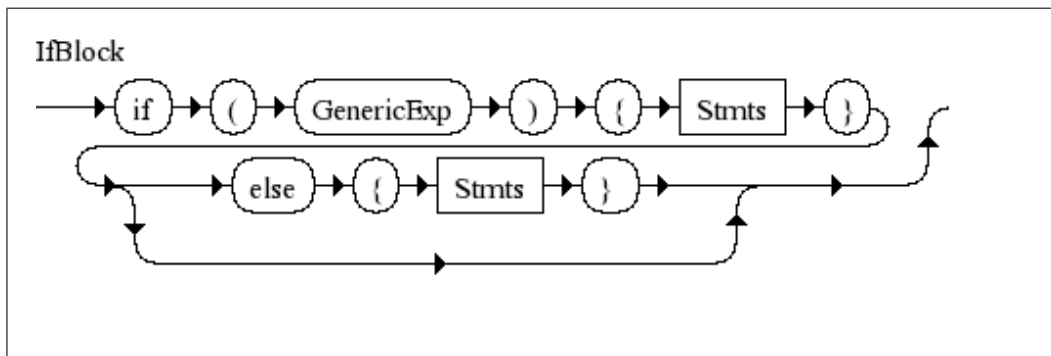
Figure 6.12: If Block Syntax Diagram

Figure 6.13: 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 7

Language Implementation

7.1 Data Type

7.1.1 Tokenizer Type

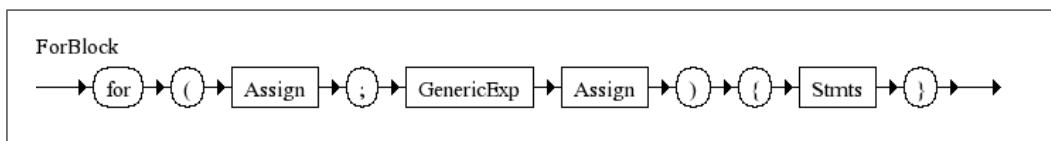All tokens are define using the Haskell Lexer Generator Alex.Tokens will be recognized and converted to Haskell data types.For example , a valid variable is comprised with one or more alphabetic characters and digit and the first character must be an alphabetic characters. This rule can be defined as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

```
1  $digit = 0-9                          -- digits
2  $lowerCase = [a-z]
3  $alpha = [a-zA-Z]                      -- alphabetic characters
4
5  tokens :-
6    $alpha [$alpha $digit \_ \' \.]* { \s -> TName s }
7  -- data definition
8  data Token =
9          TName String |
10         TBool Bool  |
11         TInt Int  |
12         ...  -- 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.

```
1  program myProgram ()
2  {
3          var1 = 1;
4          result = var1 + "a string";
5  }
```

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

```
1  lexer "program myProgram() { var1 = 1; result = var1 +
        False; return 0; }"
2  [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]
```

7.1.2   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

```
1  ( a +1 )   /=   3
2  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 ]
```
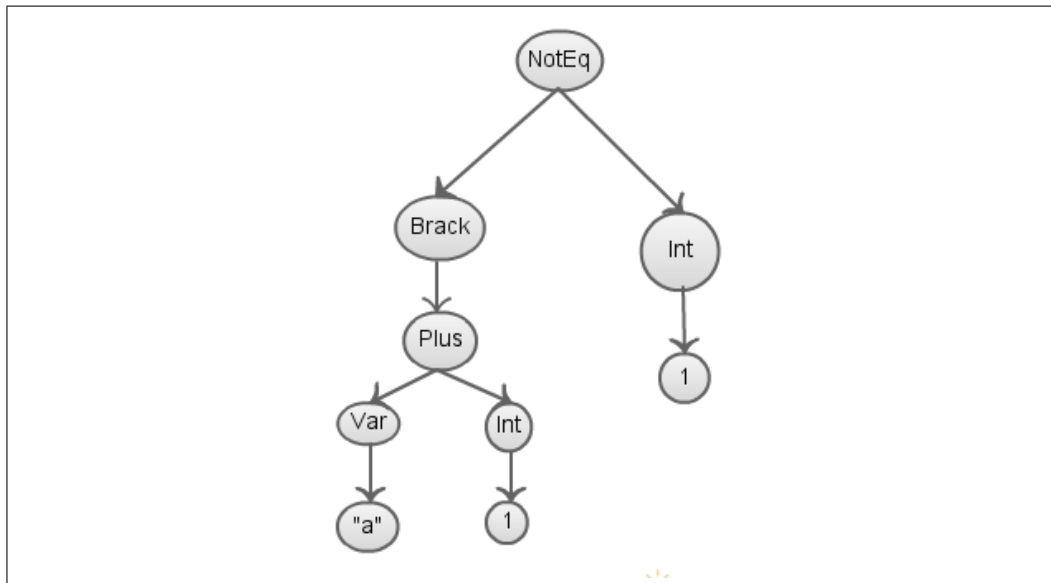
Figure 7.1: Diagram illustrating a parse tree of a generic expression

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)]]
```
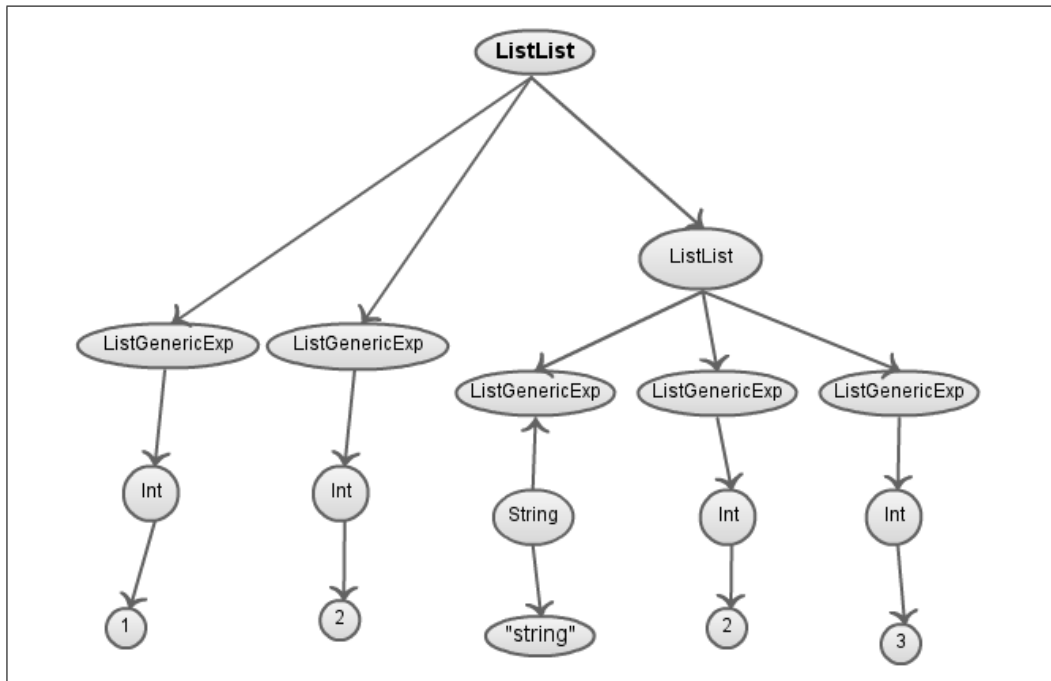
Figure 7.2: Diagram illustrating a parse tree of a list expression

7.1.3 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.

7.2   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.
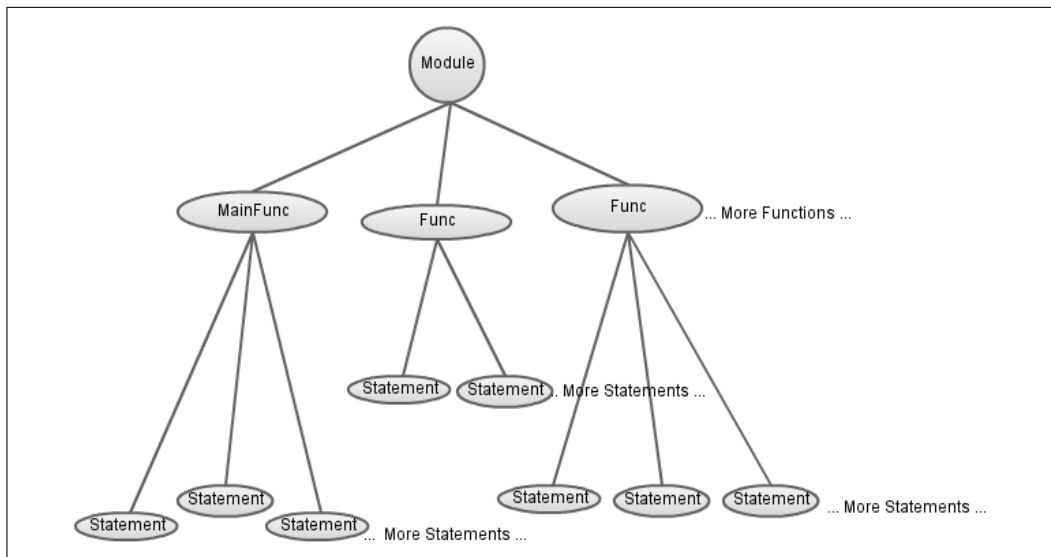

Figure 7.3: Diagram illustrating a parse tree of a module

7.3   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.
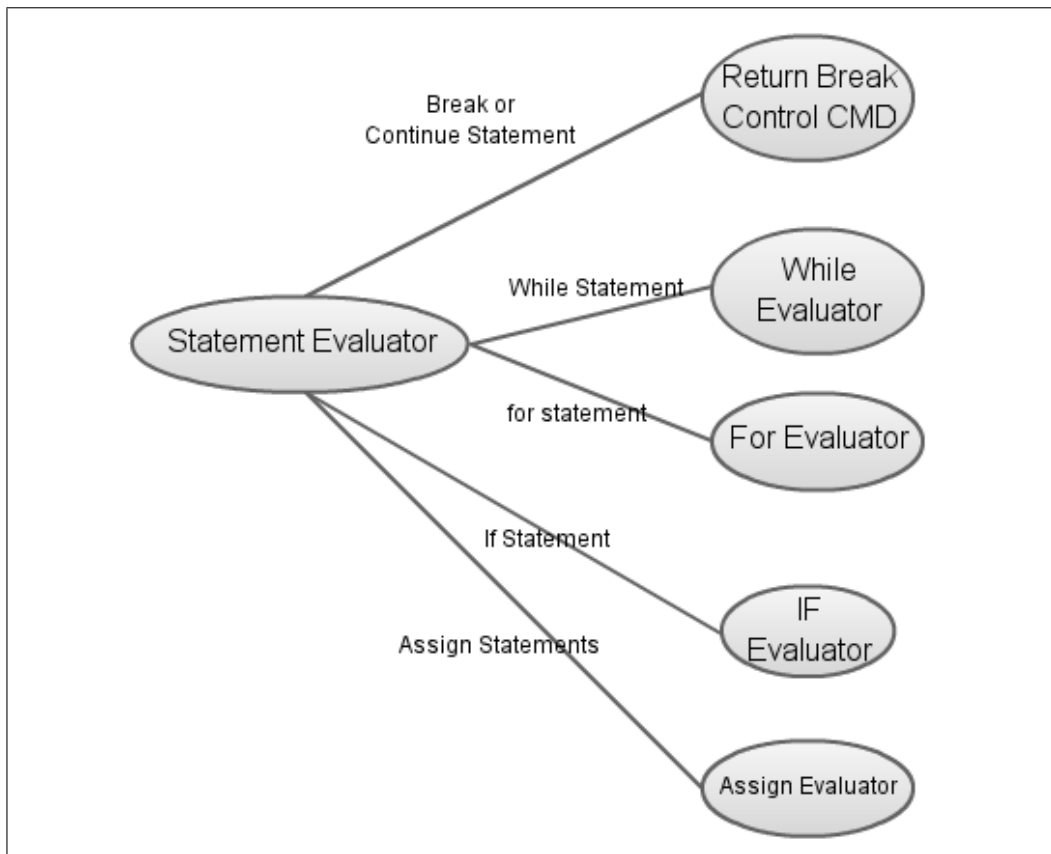

Figure 7.4: Diagram illustrating guard condition and function invocation

7.4　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
```

7.5　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.

7.6　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.

7.7　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 8

Analysis,Conclusion and Future Work

8.1　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.

8.2　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.

Compared to imperative programming paradigm,its obvious that the functional programming paradigm is not von Neumann related.For a von Neumann style programming, to construct a program,we have to write command to continuously modify the state of a memory cell to achieve ,for instance ,changing a pixel's color on the screen.Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed,do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages. [10]

8.3   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 and primitives needed to be added into the language.For example,Haskore is a clean and elegant notation for music information and structure.In this framework, musical objects consist of primitive notions such as notes and rests, operations to transform musical objects such as transpose and tempo-scaling, and operations to combine musical objects to form more complex ones, such as concurrent and sequential composition. [15]

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. [7]

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.

Therefore, I would like to put this topic in future work part.As form Haskell,creating new grammar with the powerful tool it already provided is straight forward and convenient.

Bibliography

[1] P. W Abrahams. A final solution to the dangling else of algol 60 and related languages. *Communications of the ACM*, 9(9):679–682, 1966.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: principles, techniques, and tools. *Reading, MA,*, 1986.

[3] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3).

[4] A. Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 33(2), 1932.

[5] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[6] C. Dornan and I. Jones. *Alex User Guide*. 2003.

[7] A. Gill. A haskell hosted DSL for writing transformation systems. In *Domain-Specific Languages*, pages 285–309, 2009.

[8] A. Gill and S. Marlow. Happy: The parser generator for haskell. *University of Glasgow*, 1995.

[9] Martin Grabmüller. Monad transformers step by step. 2006. Draft paper.

[10] D. Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.

[11] W. L Harrison and S. N Kamin. Modular compilers based on monad transformers. In *iccl*, page 122, 1998.

[12] D. Herington. *HUnit 1.0 User's Guide, 2002*.

[13] P. Hudak and J. H Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5), 1992.

[14] P. Hudak, J. Hughes, S. P Jones, and P. Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.

[15] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music -. *Journal of Functional Programming*, 6(03):465–484, 1996.

[16] J. Hughes. Why functional programming matters. *The computer journal*, 32(2):98, 1989.

[17] G. Hutton. *Programming in Haskell*. Cambridge Univ Pr, 2007.

[18] M. Jones. Functional programming with overloading and higher-order polymorphism. *Advanced Functional Programming*, 1995.

[19] S. L.P Jones and P. Wadler. Imperative functional programming. 1993.

[20] S. P Jones and L. Augustsson. Haskell 98 language and libraries-the revised report. december 2002. *URL http://www.haskell.org/onlinereport.*

[21] S. P Jones and L. Simon. Wearing the hair shirt: a retrospective on haskell. *Invited talk at POPL*, 206, 2003.

[22] J. Launchbury and S. L Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994.

[23] D. Leijen. Parsec, a fast combinator parser, 2001.

[24] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. *Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2001-35*, 2001.

[25] Miran Lipovaca. *Learn You a Haskell for Great Good!* 2011.

[26] J. W Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE*, volume 94, 1994.

[27] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, 2008.

[28] B. C Pierce. *Basic category theory for computer scientists*. The MIT Press, 1991.

[29] P. Wadler. Monads for functional programming. *Advanced Functional Programming*, 1995.

Appendix A

Appendix

A.1   Environment setup

```
1 sudo apt−get install haskell−platform
2 # install haskell−platform
3 sudo cabal install happy
4 sudo cabal install alex
5
6 cd PATH_TO_PROJECT
7 #change path to project directory
8
9 happy −i grammar/Grammar.y
10 alex grammar/Lexer.x
11 ghc −−make −i"test" −i"src" −i"grammar" "src/Main.hs"
       −outputdir out −o yun
12 #building the project
13
14 rm grammar/∗.o
15 rm grammar/∗.hi
16 rm src/∗.o
17 rm /src/∗.hi
18 rm yun
19 #clean up the output folder
```