



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 4, 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

1	Introduction	7
1.1	Objective and Motivation	7
1.2	Challenges	7
1.3	Benefits of using Haskell	8
1.4	Development Methodology	8
1.5	Development Environment	9
1.6	Code Formats	9
2	Technologies	11
2.1	Parsing technologies	11
2.1.1	Formal Grammar	11
2.1.2	The Hierarchy of Grammars	11
2.1.3	Backus–Naur Form and Extended Backus–Naur Form .	13
2.2	Parser Generator Haskell Happy	14
2.3	Monadic Parsing using Parsec	14
2.4	Lexical analysis	15
2.4.1	The Lexer Generator Alex	15
2.5	Testing and Evaluation Technologies	15
3	Haskell	17
3.1	History	17
3.2	Haskell and Category Theory	18
3.3	Algebraic Data Type	19
3.4	Type Classes	20
3.5	Higher Order Functions	20
4	Monads	22
4.1	Overview	22
4.1.1	History	22
4.1.2	Side effect	22
4.2	Introduction to Monad	23

4.3	IO Monad	24
4.4	State Monad	25
4.5	List Comprehension and List Monad	26
4.5.1	List Comprehension	26
4.5.2	List Monad	27
4.6	Do Notation	27
4.7	Error Monad	28
4.8	Other Monads	29
5	Monad Transformers	30
5.1	Overview	30
5.2	State Transformer	30
5.3	Error Transformer	31
5.4	Combining Monads	32
6	Language Interpreter Design	33
6.1	Architecture	33
6.1.1	Parser	33
6.1.2	The Execution Engine	34
6.2	Imperative and Declarative language	34
6.3	Type System	35
6.3.1	Dynamic Typing and Static Typing	35
6.3.2	Strong Typing and Weak Typing	35
6.4	Problem and resolution in writing BNF/EBNF rules	36
6.4.1	Shift//Reduce Problem	36
6.4.2	Reduce//Reduce Problem	36
6.5	Syntax Design	36
6.5.1	The Main Structure	36
6.5.2	Statements	38
6.5.3	Generic Expression	39
6.5.4	Code Block	42
7	Language Implementation	45
7.1	Data Type	45
7.1.1	Tokenizer Type	45
7.1.2	Parser internal data type	46
7.1.3	Interpreter Internal Data Type	48
7.2	Module Evaluator	49
7.3	Statement Evaluator	49
7.4	Symbol Table and Parse Tree	51
7.5	Generic Expression Evaluator	51

7.6	Function Invocation	51
7.7	Main Function	51
8	Analysis, Conclusion and Future Work	52
8.1	Analysis	52
8.2	Conclusion	52
8.3	Future Work	53

List of Figures

3.1	How the high order function apply function f to list [1,2,3,4,5] with initial value z	21
4.1	Combining Two "Stateful" Monadic Functions	26
4.2	The First Monad Law illustration	27
5.1	Diagram illustrating a state monad	31
6.1	The Parser	33
6.2	The Parser	34
6.3	Module Syntax Diagram	37
6.4	Main Function Syntax Diagram	37
6.5	Function Syntax Diagram	37
6.6	Statement Syntax Diagram	38
6.7	Assign Statement Syntax Diagram	40
6.8	Expression Syntax Diagram	41
6.9	Expression Syntax Diagram	42
6.10	List Member Syntax Diagram	42
6.11	While Block Syntax Diagram	43
6.12	If Block Syntax Diagram	43
6.13	for Block Syntax Diagram	43
7.1	Diagram illustrating a parse tree of a generic expression	47
7.2	Diagram illustrating a parse tree of a list expression	48
7.3	Diagram illustrating a parse tree of a module	49
7.4	Diagram illustrating guard condition and function invocation .	50

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 Challenges

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

1.3 Benefits of using Haskell

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

A pure function is a function that accepts an input and produces an output. In Object-Oriented language, a program is constructed using classes and instances which encapsulate computations and states. Haskell program is constructed by functions as a 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]

1.4 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 phases: 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 use the iterative and incremental approach in my project rather than a completed agile development or RUP cycle for the reason that it's 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 guidelines in my project.

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

Eclipse with EclipseFP plug-in provides support for source code and package management in Eclipse. To keep track of 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 base 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 install in different platform include Windows, Linux and Mac OS X. I am using Ubuntu Linux 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" [?].

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. 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

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

$$S \mapsto aS \text{ and } S \mapsto ab$$

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 $\{aab, aaab, aaaab, \dots\}$.

2.1.2 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 Turing 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.

[?] 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 be the projection rule 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)	rule 7
((a+S))==S) && (S>S)	rule 3
((a+2))==S) && (S>S)	rule 1
((a+2))==S) && (S>S)	rule 3
((a+2))==S) && (S>S)	rule 1
((a+2))==S) && (a>S)	rule 3
((a+2))==S) && (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 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,

$$\langle symbol \rangle ::= _expression_$$

The left side of the equation can only be non-terminal thus enclosed with $\langle \rangle$. 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

[?]

Recursive rules of BNF like

$$1. \langle exp \rangle := \langle exp \rangle | sub$$

$$2. \langle exp \rangle := sub$$

that expressing a sequence of a particular syntactic element can be simplified using quantifier in EBNF as $\langle exp \rangle := sub^+$

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 a 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 to parser generator, monadic parsing has two major benefits

1. No need to learn additional parser generator grammar since parser combinator is written in the same language.
2. parser can be adjusted 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 parsed to tokens,

<code>s3</code>	identifier, variable name
<code>=</code>	operator
<code>s1</code>	identifier, variable name
<code>+</code>	operator
<code>"a string"</code>	a string constant

The regular expression $[a-zA-Z]1[a-zA-Z_0-9]^*$ can identify strings that begin with an 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 existing `lex` users, although it does depart from `lex` in a number of ways. [Alex User Guide]

2.5 Testing and Evaluation Technologies

There are two types of testing in Haskell. The type testing and unit testing. These correspond to two packages in Haskell, `QuickCheck` and `HUnit`.

QuickCheck

For pure functions, a programmer can write a logical expression or functional specification to prove the code is correct rather than provide a specific 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* $\forall s. \text{length}(\text{take5 } s) = 5$ 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.

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[A set of postulates for the foundation of logic].

In 1936 Church isolated and published just the portion relevant to computation, what is now called the untyped lambda calculus[An unsolvable problem of elementary number theory].

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 Architecture (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. [Haskell 98 Language and Libraries: The Revised Report. December 2002.]

The first version of Haskell ("Haskell 1.0") was defined in 1990.["A history of Haskell: being lazy with class"]. The latest Haskell version describe in the Haskell report 2010,the implementation of the latest version Haskell Platform is released in Feb 2011 [haskell.org].

To summarize it ,Haskell has evolved continuously in recent years.

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 $f : A \rightarrow B$,

$$id_a \circ f = f \text{ and } f \circ id_a = f.$$

[?]

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 and 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 form 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 rewrite the hold function using a single fold

The all map/fusion is is equivalent to $foldr f e . map g = foldr ($

$xy \rightarrow f(gx)y)e$

therefore, the

$sum_of_square = foldr ($

$xy \rightarrow x^2 + y)0$

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,

```
data Bool = False | True
```

The two value constructors are separated by `|`, which allow us to construct a `False` value and `True` value, they are both belong to the `Bool` data type.

The constructor allows us to defined a constructor that holds multiple argument.

```
data Publication = Book Name Author ISBN Year
                | Paper Title Author Year Magazine
                | DVD Name Length Description
```

3.4 Type Classes

Type classes were introduced to the Haskell Committee by Wadler in a message sent to the fplangc mailing list dated 24 February 1988.[Monads for functional programming]

they define a set of functions that can have different implementations depending on the type of data they are given. Type classes may look like the objects of object-oriented programming, but they are truly quite different.[real world haskell]

3.5 Higher Order Functions

In Haskell, a function takes functions as parameter or returns functions is called high-order function. High order function is the most important features in Haskell because it allows functions to glue together.

Some of the useful high-order functions are,

Function Definition	Description
$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$	composition two function into one function
$foldl :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$	apply a computation to each member of a list with the initial value a
$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$	generate a new list from a existing list by applying a computation
$zipWith :: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$	generate a new list from two list by applying a computation
$filter :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$	filter the element of the list with a condition

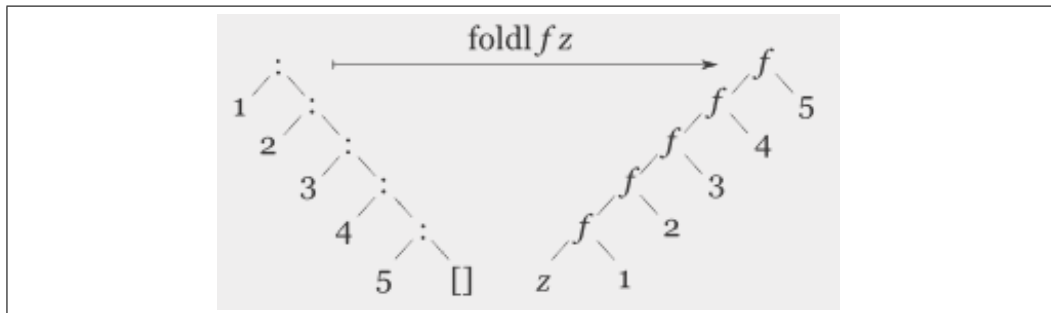


Figure 3.1: How the high order function apply function `f` to list `[1,2,3,4,5]` with initial value `z`

Folds can be used to implement any function where you traverse a list once, element by element, and then return something based on that. Whenever you want to traverse a list to return something, chances are you want a fold. [learn you a haskell]

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.[A History of Haskell:Being Lazy With Class]

In April 1997,the list comprehension is generalised into monad in the Haskell report 1.4.[A History of Haskell:Being Lazy With Class]

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.[Monads for functional programming]

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 lock 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]

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 Haskell have been used to model IO, state, logger, error as well as List.

4.2 Introduction to Monad

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

```
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 functions, one should be of type **m a** and another should be of type **a -> m b**.

Besides these two functions, Haskell also provides other monadic operators 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 operations are defined 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 learn 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 do not provide escape function.The IO monad 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) ))
```

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, it's 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 specifying 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 following facts of a computation with state, they are

- It is possible to construct a new state monadic function without initial state.
- When combining two states, 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 both the result of computation as well as the final state as an output of the computation.

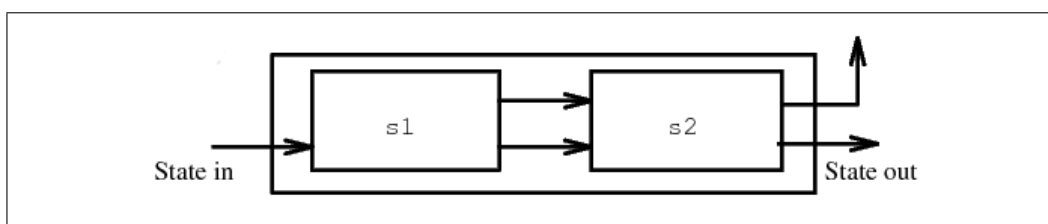


Figure 4.1: Combining Two "Stateful" Monadic Functions

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

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 \in , 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.

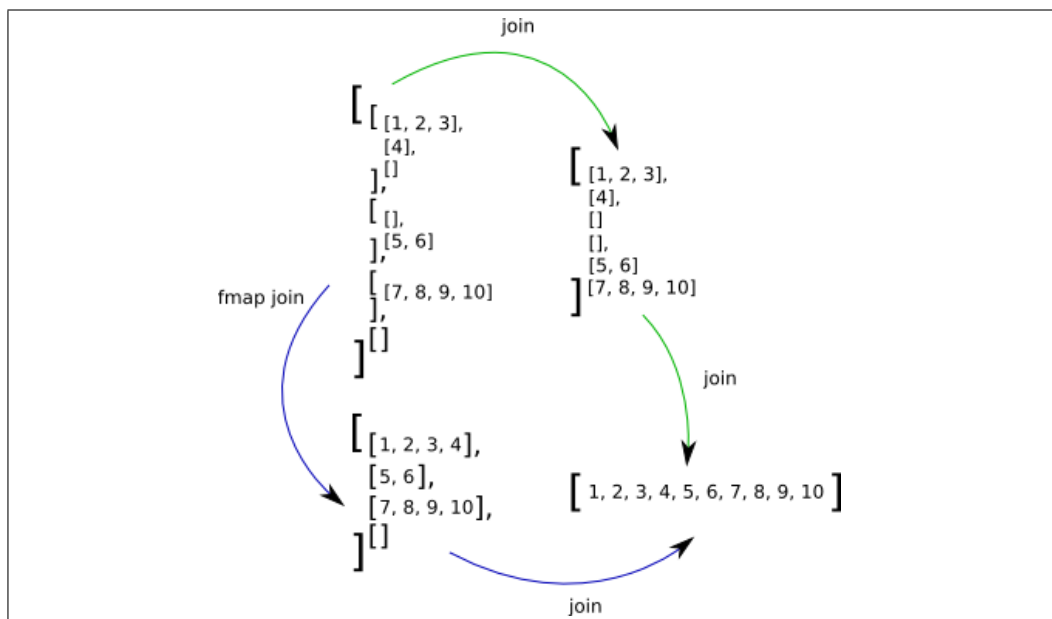


Figure 4.2: The First Monad Law illustration

[Haskell/Category theory]

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**.

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 unwrapping 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 for 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. [Monad Transformers Step by Step]

In real world application, it is often the case that a combination of different type of action is required to put into one function. Monad transformer provides 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 of type `s`, and delivers a value of type `a`. You can think of it as a box, like this

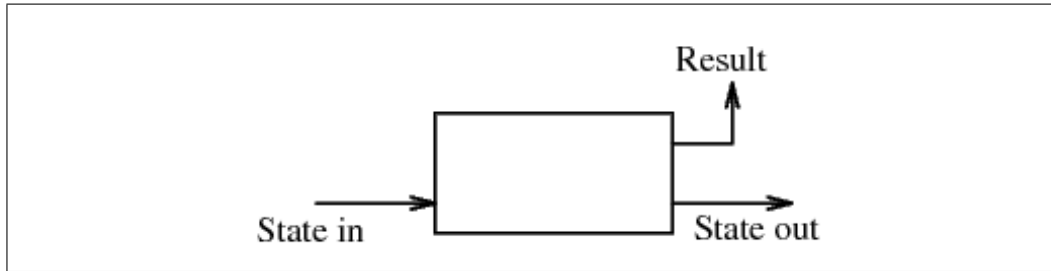


Figure 5.1: Diagram illustrating a state monad

[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.

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

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.

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.[Modular Compilers Based on Monad Transformers]

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.

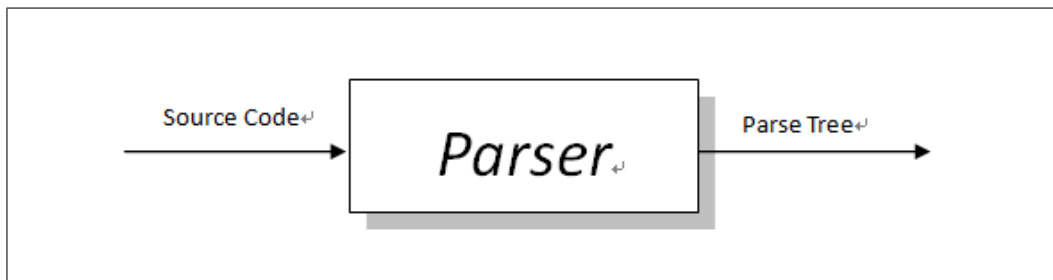


Figure 6.1: The Parser

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. [?]

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.

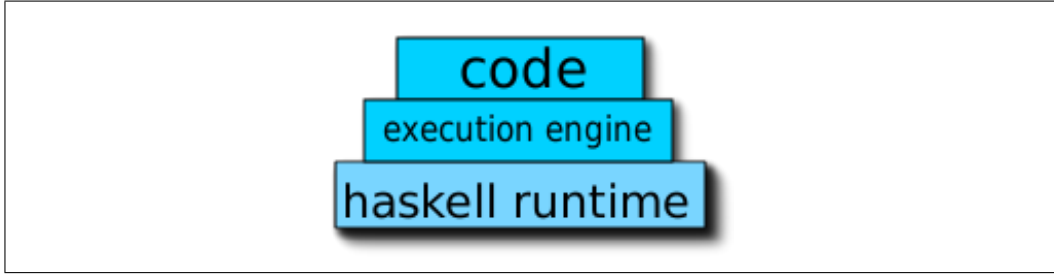


Figure 6.2: The Parser

6.2 Imperative and Declarative language

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

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 compiler will check 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
<pre>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 */</pre>	<pre>a=1; /* does not need to specified a type for this variable */ a ="a string"; /* it valid to change the type of the variable */</pre>

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
<pre>a=123; /* a is a number */ b="123" /* b is a string */ c=a+b /* return type error */</pre>	<pre>a = 123; b = "123"; c = a+b; /* either a will be convert to a string or b will be convert to a number */</pre>

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 BNF/EBNF rules

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

- Shift - push token onto stack
- Reduce - remove handle from stack and push on corresponding non-terminal
- 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]

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 several functions. The main function is an entry point; it may invoke other functions.

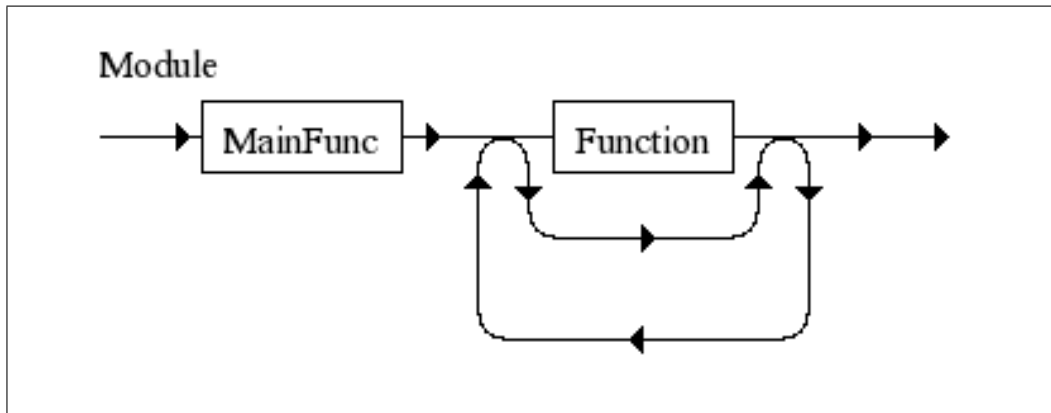


Figure 6.3: Module Syntax Diagram

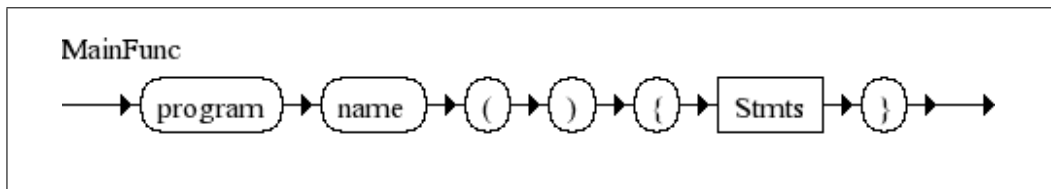


Figure 6.4: Main Function Syntax Diagram

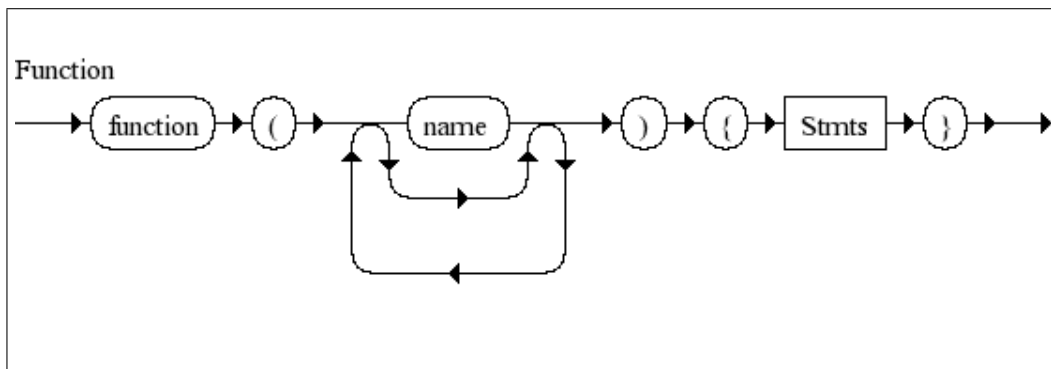


Figure 6.5: Function Syntax Diagram

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 , WhileBlock, IfBlock, ForBlock are recursively defined by statements as well. This allow nested for loop ,nested while loop and etc.

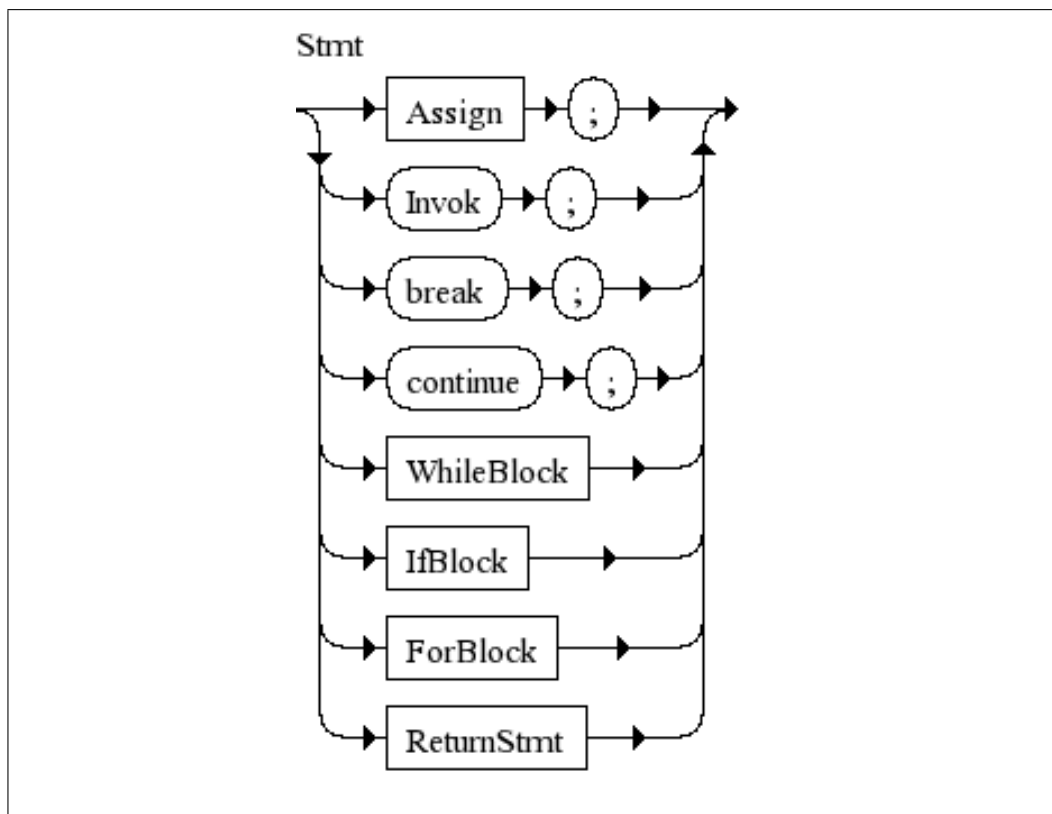


Figure 6.6: Statement Syntax Diagram

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 ,

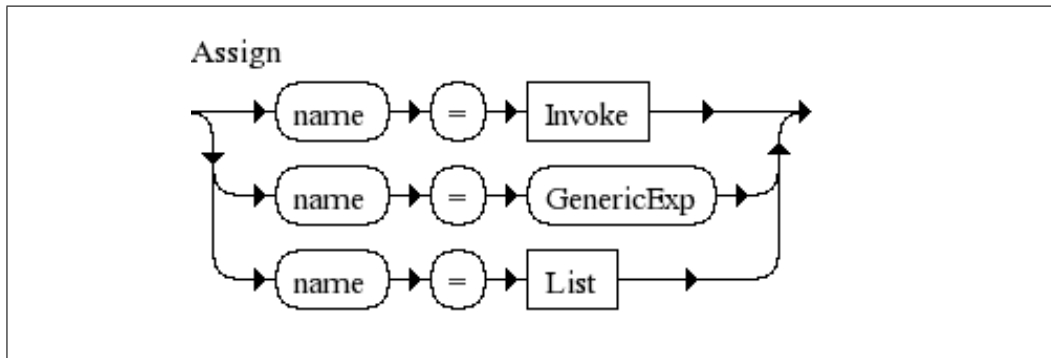


Figure 6.7: Assign Statement Syntax Diagram

```

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.

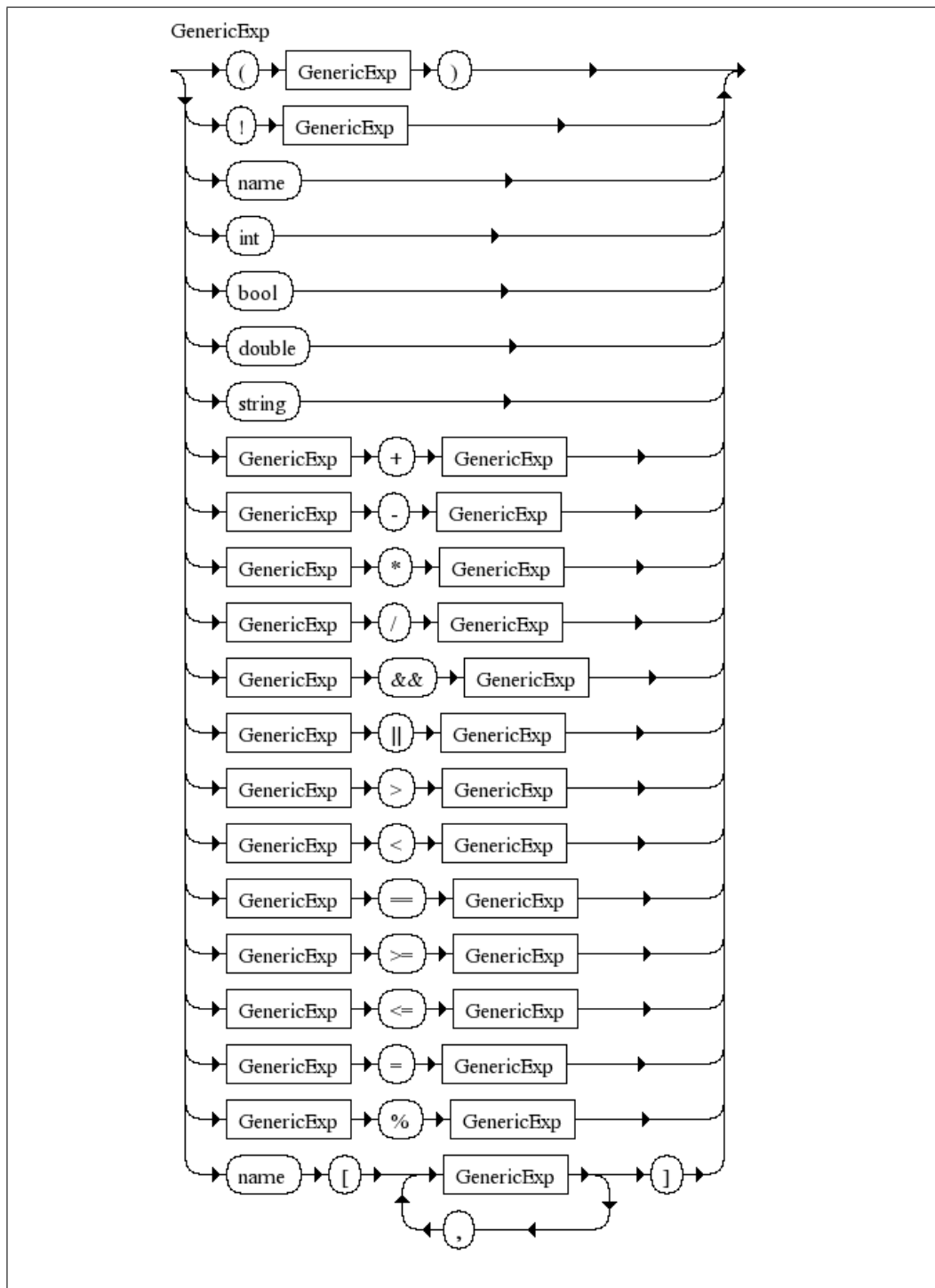


Figure 6.8: Expression Syntax Diagram

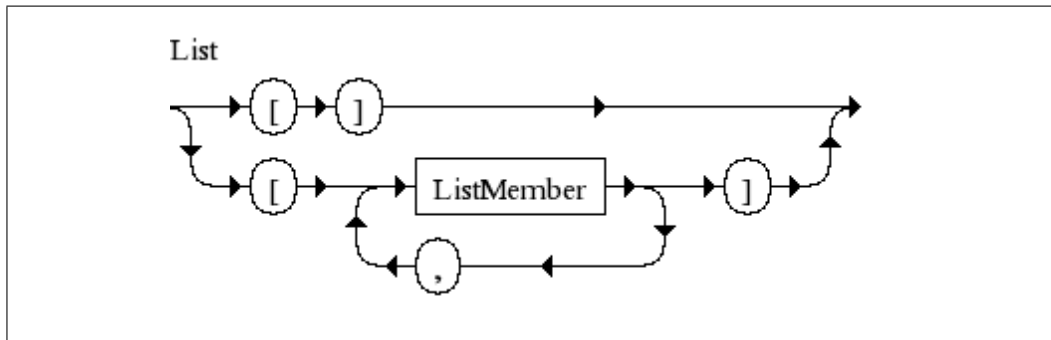


Figure 6.9: Expression Syntax Diagram

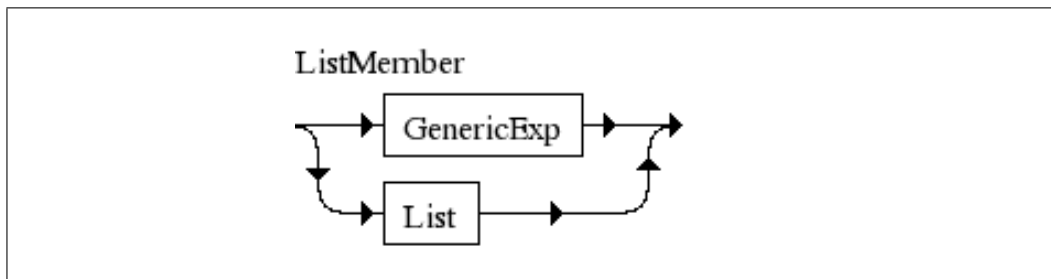


Figure 6.10: List Member Syntax Diagram

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.

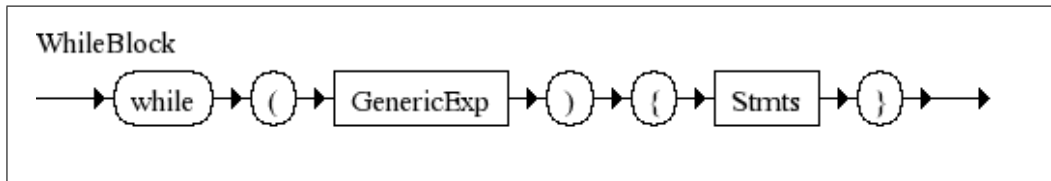


Figure 6.11: While Block Syntax Diagram

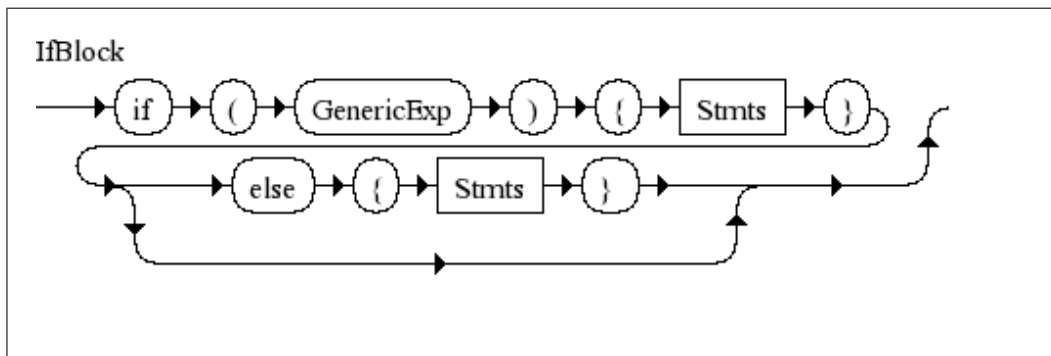


Figure 6.12: If Block Syntax Diagram

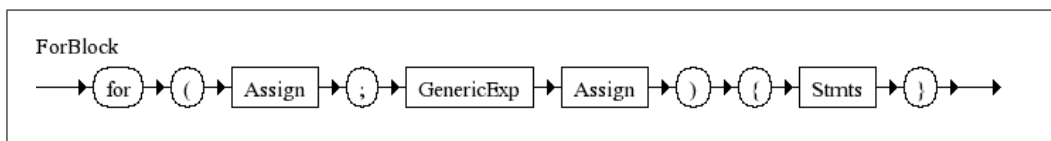


Figure 6.13: for Block Syntax Diagram

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,TOCB]

```

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 ]

```

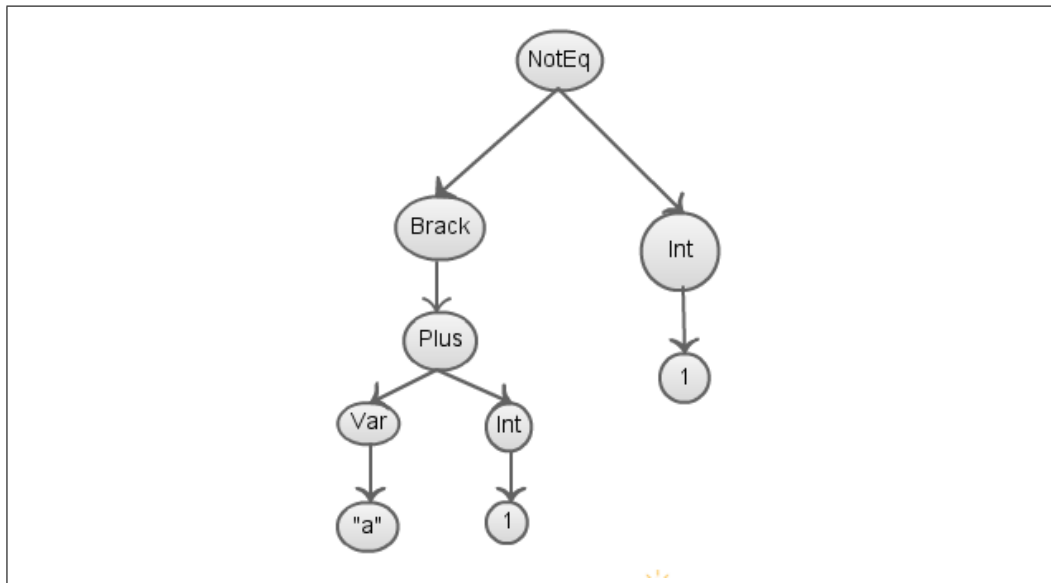



Figure 7.1: Diagram illustrating a parse tree of a generic expression

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)]]
```

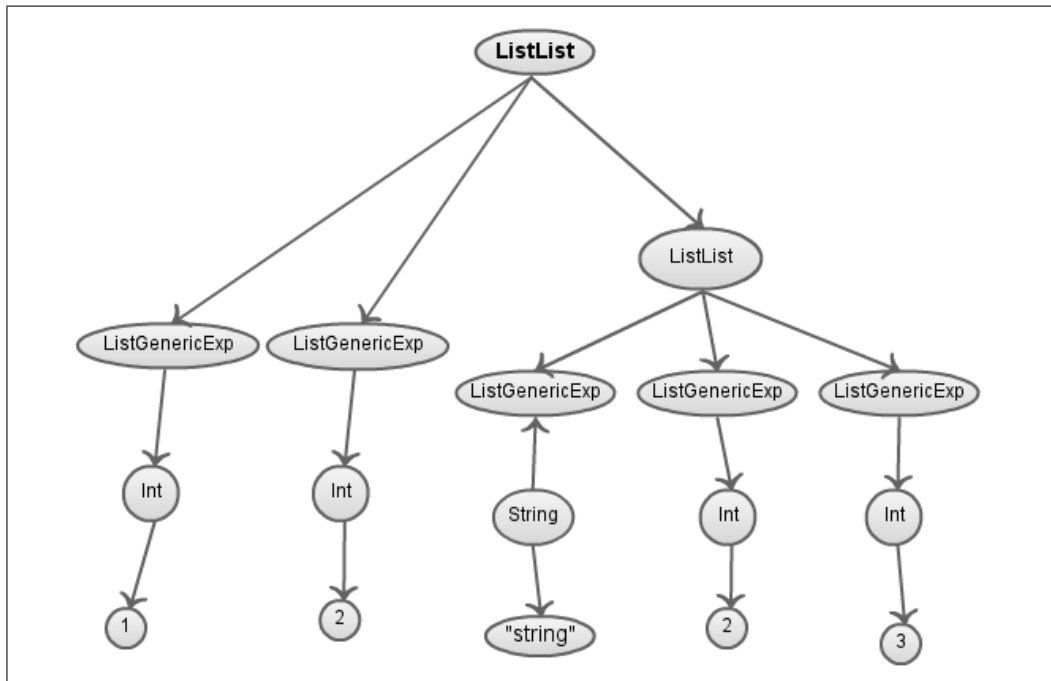


Figure 7.2: Diagram illustrating a parse tree of a list expression

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 an interpreter corresponding to it. The sub-interpreter will return for example an integer to represent the result of a generic expression. The recursive data type is used for 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.

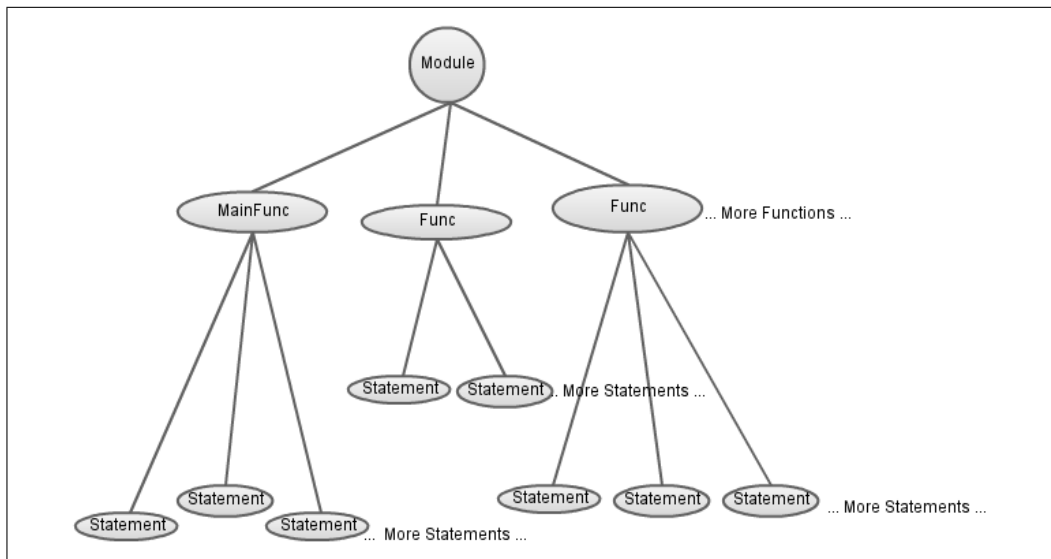


Figure 7.3: Diagram illustrating a parse tree of a module

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 pattern 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.

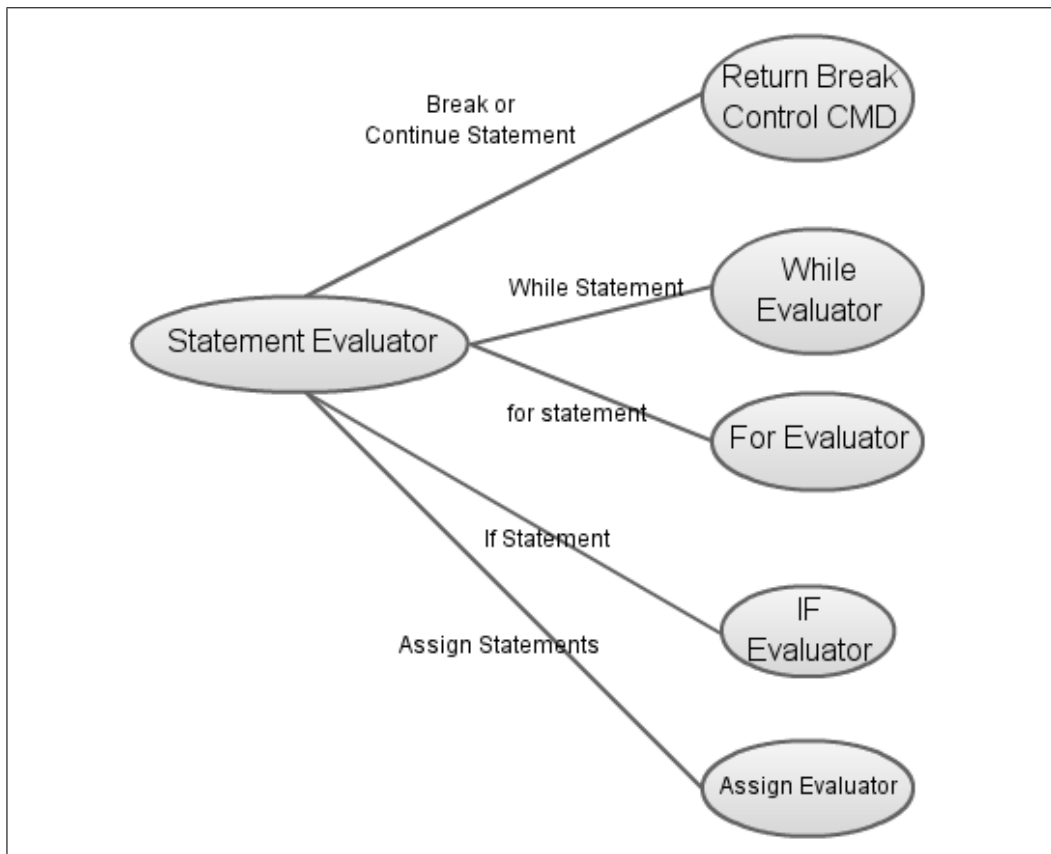


Figure 7.4: Diagram illustrating guard condition and function invocation

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, then 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 or 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 phase I can have correct implementation.

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

There are great amounts of disputes on pros and cons between different programming languages. Those disputes, to a large extent, have ignored the fact that programming languages are created for computer scientists to resolve problems.

8.3 Future Work

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

With Haskell's powerful support for abstraction, support like monads, high-order functions and function composition, implementing a language is not so painful. Though the current project is only focused on a generic purpose language, other people have implemented lots of Domain-specific languages 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 problems that is easy to design and tedious to implement and Haskell is the best candidate to do this job.