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

March 30, 2011

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

Abstract

In this thesis,

Keywords:programming language YUN

Declaration

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

Signed ______________________

Zhen Lao

Acknowledgements

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

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective and Motivation

The objective of this project is to develop a weak-type interpreted language using Haskell.This language is able to support the following feature,

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

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

1.2 Benefits of using Haskell

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

1.2.1 Pure Function

A pure function is a function that accept an input and generate and an output.

1.3 Development Methodology

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

Chapter 2

Compiler and Interpreter Technologies

2.1 Parsing technologies

2.1.1 Formal Grammar

Mathematically, formal grammar consists of:

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

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

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

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

In addition,we are able to write all the production rule from the given abstract language, the language like

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 Turning machine.
- Type-1 grammars (context-sensitive grammar).if all rules are of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ where $\alpha\, \beta\, \gamma$ are terminal symbols and A is non-terminal symbol.
- Type-2 grammars (context-free grammar).
- Type-3 grammars (regular grammar).

2.1.3 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,

$$< \mathit{symbol} >::= \_\_\mathit{expression}\_\_$$

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

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

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

- + : which means that something can appear one or more times

[?]

Recrusive rules of BNF like

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

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

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

2.2 Parser Generator Haskell Happy

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

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

2.3 Monadic Parsing using Parsec

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

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

2.4 Lexical analysis

Before parsing,the lexical analsier will scan the source code and generate a sequence of token.

2.4.1 Regular Expression and token

Tokens are defined by using regular expression,

2.4.2 The Lexer Generator Alex

In this project, the Alex Haskell Lexer generator is apply in generating token streams.

each token can be defined using regular expression.

Chapter 3

Monads and Haskell

3.1 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 $g \circ f : dom\ f \to cod\ g$, satisfying the following associative law:
   For any arrow $f : A \to B, g : B \to C, and\ h : C \to D$(with A,B,C and D not necessarily distinct),
   $$h \circ (g \circ f) = (h \circ g) \circ f$$
5. for each object A, an identify arrow $id_a : A \to A$ satisfying the following identity law:
   For any arrow $f : A \to B$,
   $$id_a \circ f = f\ and\ f \circ id_a = f.$$

[?]

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

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

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

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

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

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

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

therefore, the

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

3.2 Monadic Function

3.3 Monads

In Haskell,monad is used an abstract data type constructor to represent multiple kinds of computation such as a computation that will do IO action,or a computation that has state.Those computations are in-pure because that manipulate the outside world.In Haskell.Mathematically, monads are governed by set of laws that should hold for the monadic operations [A Gentle Introduction to Haskell, Version 98]. There are two basic law in monads ,they are bind return .The Monad class is defined as follow:

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

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

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

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

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

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

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

Monadic Characteristic

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

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

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

3.3.1 IO Monad

Haskell use IO monad to limit the IO sequence.One significant different between IO monad an other monad is that it does not provide escape function like

```
IO String ->String
```

3.3.2 State Monad

The State Monad is defined as follow

```
newtype State s a = State {runState :: s -> (a, s)}
```

3.3.3 List Monad

Haskell try to use list monad to represent a calculation that return multiple result.

3.4 Using Monad Operator to Combine Monadic Function

3.5 Type system in Haskell

3.6 Do Notation - Reinventing a imperatilan-guageve language

Chapter 4

Monad Transformer

Chapter 5

Language Interpreter Design

5.1 Type System

5.1.1 Dynamic Typing and Static Typing

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

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

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

5.1.2 Strong Typing and Weak Typing

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

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

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

5.2 Problem and resolution in writing BNF/EBNF rules

Shift//Reduce Problem

Reduce//Reduce Problem

Chapter 6

Language Implementation

Bibliography