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

March 25, 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

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

The objective of this project is to develop an week-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 array
- polymorphic array

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

1.2 Introduction to 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.

1.3 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 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.

A formal grammar of this type consists of:

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

[**?**]

From previous 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 $\{aab, aaab, aaaab, \cdots\}$.

The Chomsky hierarchy consists of the 4 levels:

- Type-0 grammars. It is a unrestricted grammars that include all.
- Type-1 grammars.
- Type-2 grammars.
- Type-3 grammars.

2.1.2 Backus–Naur Form and Extended Backus–Naur Form

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

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

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

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

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

[**?**]

Recrusive rules of BNF like

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

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

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

2.2 Parser Generator Haskell Happy

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

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

2.3 Monadic Parsing using Parsec

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

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

2.4 Lexical analysis

Before parsing,the lexical 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.

2.5 Language definition

Production rule in EBNF

Chapter 3

Monad in 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

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

3.2 Monadic Function

3.3 Using Monad Operator to Combine Monadic Function

3.4 Type system in Haskell

3.5 Do Notation - Reinventing a imperatilanguageve language

Chapter 4

Monad Transformer

Chapter 5

Language Design

—

Chapter 6

Language Implementation

Bibliography