DUBLIN INSTITUTE of TECHNOLOGY

Institiúid Teicneolaíochta Bhaile Átha Cliath

Implmenting a interpreter for a scripting lanaguge using Haskell

INTERIM REPORT

Zhen Lao

zhen.lao@student.dit.ie

Supervisor: Richard Lawlor

2nd Reader: Cindy Liu

March 23, 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 ,we focus on

Keywords:programming language YUN,Haskell,Parsec,plan,CFG,EBNF

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

Grammar Design

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

Future Work and Project Plan

3.1 Future Work

I have done most of the research work of the project.the future work will be implementing the actual parser.There will be an initial implementation of part of the EBNF definition.The execution engine will be implemented at the same as the parser.Due to Haskell have a powerful abstraction mechanism and a suitable library (Parsec),the implementation will not be too difficult.

The major barrier will be implementing the execution engine since I have no knowledge about it. There will be more research in the later stage of this project on the interpreter part as well as the grammar and Haskell part.

3.2 Project Plan

Before November I have finish most of research on Chomsky's CFG grammar and Haskell.I have started to implement a prototype of the interpreter.
November 12 to November 26 Research on the execution engine of the interpreter.
November 12 to November 31 Development a prototype by following the documentation of parsec. The document "Write Yourself a Scheme in 48 Hours/Parsing" offer an example to implements a interpreter for Scheme using parsec library of Haskell.
November 31 to January 15 Implement a subset of EBNF specification of *yun*.Add the error checking to the interpreter.Meantime,as the code growing ,unit test will be added to guarantee the quality of existing code.
January 16 to February 15 Implement all the EBNF specification of *yun*
February 31 to March 15 Implement the IO command (library).Add more test code.
February 16 to March 16 Review the EBNF of *yun* programming language.Implement more library for it.Start the system testing.
March 17 to April 8 Prepare for the project fair.