Bilkent University

Department of Computer Science

CS 315 - Programming Languages

Design and Implementation of a Parser for a Propositional Logic Programming Language

EZME

Project Report

Group 55

Emre Gürçay

Emre Başar

Erdem Adaçal

Course Instructor: Ertuğrul Kartal Tabak

Deadline: Nov 20, 2017

TABLE OF CONTENTS

1. DESCRIPTION

In this project, we as a group, demonstrated a programming language called “Ezme” with using similar syntax like C based programming languages but this language is used for predicates. In previous part we’ve demonstrated lex rules for our programming language which includes variables, comments, initialisation, assignment operation, logical expressions, while loops and function declerations. In this part of the project we’ve written parser using yacc according to our previous lex rules. We’ve faced some conflicts and for solving those conflicts, we’ve changed our lex. In addition to that as it is wanted we’ve added a structured data type for group of values, in other words arrays.

In this report first, BNF description of our language based on the terminal symbols returned by our lex implementation will be given. Then in the second part of this report, general description of our language and important non terminals descriptions will be given. Then descriptions of each nontrivial tokens will be given. As there is no conflict which remains unsolved, there will be no conflict explanation.

1.1 STRUCTURE OF THE LANGUAGE

In our language we wanted to have as many as identifier options thus while designing the language we determined the name such as while, if , else , etc. in the ‘ ‘ brackets like ‘if’ , ‘else’. This gave us the reduction of the reserved word count through the programming language. We added this flexibility because we thought that while coding we always want to use words such as while, input , print etc. Also we were influenced by C based languages because in C based languages it is very easy to undertstand the coding stuctrure and it is also very flexible.

While designing the language we also had many problems such as precedence and ambiguity. We wanted “ezme” to have precedence for the logic expressions but it made the logical structure of the language very hard. So we had help from our teachers’ example codes for eliminating those difficulties. Also implemeting precedence for the logical expression made yacc codes very complicates for us. Because of implementing precedence false we had 48 reduce/reduce conflicst. For eliminating those reduce/reduce conflicts we draw a parse tree thus we were able to see our conflict reasons. Also we had 3 shift/reduce conflictes. To eliminate those shift/reduce conflilcts we used the linux’ state diagram which shows the every state of the program and point where the conflicts occur. At the end of the parsing stage we were able have no conflicts and a working parser.

Also we had many problems with showing the line number of the errors because we were implementing the \n line at the lex stage wrong incrementing the lineno++ false. At first we thougth using yylineno could be useful but it was not like expected we were faced with many number of errors. At the end we solved that problem by looking at the calculator example that we did in the classroom.

2. BNF OF THE LANGUAGE

lines ::=<lines> <line>

line::= <start>

start:== MAIN <lcb> <statements> <rcb>

statements::= <statementMore>

statementMore::= <statement> | <statementMore> <statement>

statement::= <assign> | <ifElse> | <while> | <commentbegins> | <predicateCall> |

<createlist> | <message> | <predicate>

predicate::= <identifier> <bracket> <list> <bracket> <lcb> <statementMore> <output>

<logicalexp> <rcb>

predicatecall ::= <identifier> <bracket> <list> <bracket>

assign ::= <identifier> <equal> <true> | <identifier> <equal> <false> | <identifier> <equal>

<not> <logicalexp> | <identifier> <equal> <logicalassign> <logicalexp> | <identifier>

<equal> <predicatecall> | <identifier> <equal> <const> | <identifier> <equal> <input> |

<listelement> <equal> <logicalassign> <logicalexp> | <listelement> <equal> <true> |

<identifier> <equal> <false> | <const> <equal> <true> | <const> <equal> <false> | <const>

<equal> <logicalassign> <logicalexp> | <const> <equal> <predicatecall> | <const> <equal>

<not> <logicalexp>

ifElse::= <ifname> <bracket> <logicalexp> <bracket> <lcb> <statementmore> <rcb> |

<ifname> <bracket> <logicalexp> <bracket> <lcb> <statementmore> <rcb> <elsename>

<lcb> <statementmore> <rcb>

while ::= <whilename> <bracket> <logicalexp> <bracket> <lcb> <statementmore> <rcb>

input ::= ‘input’ ‘false’ | ‘input’ ‘true’

logicvalue ::= <term>

term::= ‘true’ | ‘false’

type::= <identifier> | <const> | <term> | <listelement>

listelement::= <identifier> <lcb> <number> <rcb>

list::= <type> | <list> <comma> <type>

createlist ::= <identifier> <lcb> <list> <rcb>

message::= ‘message’ <logicalexp>

logicalexp::= <ifandonlyifexp> <equals> <logicalexp> | <ifandonlyifexp>

ifandonlyifexp::= <impliesexp> <ifandif> <ifandonlyifexp> | <impliesexp>

<impliesexp>::= <orexp> <implies> <impliesexp> | <orexp>

<orexp> ::= <andexp> <or> <orexp> | <andexp>

<andexp> ::= <exp> <and> <andexp> | <exp>

<exp> ::= <type> | <bracket> <logicalexp> <bracket>

commentbegins::= <commentbegin> statements <commentfinish>

digit ::= [0-9]

char ::= [a-zA-Z]

identifier ::= <char> | <identifier> <digit> | <identifier> <char>

const ::= <lcb> <identifier>

number::= <digit> | <number> <digit>

whilename ::= ‘while’

elsename ::= ‘else’

comma ::= ‘,’

bracket ::= ‘^^’

lcb ::= ‘<<‘

rcb ::= ‘>>’

commentbegin::= ‘<<<‘

commentfinish::= ‘>>>’

ifname::= ‘if’

3. NON-TERMINALS

lines:/ nothing */*
|lines line
;

Definition: It allows our program to have multiple lines of code. /*nothing*/ is for

determining the line which is empty.

line:start
;

Definition: This provides our program to have only one unique start point (main function).

start: MAIN LCB statements RCB
;
Definition: Programs code must start with MAIN keyword. And whole code has to be

between brackets.

statements: statementMore
;
Definition: This provides our program to have only one block of code.

statementMore: statement
|statementMore statement
;
Definition: This provides that our program code can be consisted of one or more statements

of code.

statement: assign
|ifElse
|predicate
|while
|commentbegins
|predicatecall

|createlist
|message
;
Definition: This determines what statements are in our language, EZME.

predicate: IDENTIFIER BRACKETS list BRACKETS LCB statementMore OUTPUT logicalexp RCB
;
Definition: Predicate gives the opportunity to return a value according to the inputs it gets.

Inside a predicate the programmer could write statement as much as s(he) wants. Also there is no restriction for the input value of the predicate.

predicatecall: IDENTIFIER BRACKETS list BRACKETS
;
Definition: predicatecall is for assinning an identifier or an constant value to a output value of the predicate. The output is given according to inputs it gets.

assign: IDENTIFIER EQUAL TRUE
|IDENTIFIER EQUAL FALSE
|IDENTIFIER EQUAL NOT logicalexp
|IDENTIFIER EQUAL LOGICALASSIGN logicalexp
|IDENTIFIER EQUAL predicatecall
|IDENTIFIER EQUAL CONST
|IDENTIFIER EQUAL input
|listelement EQUAL LOGICALASSIGN logicalexp
|listelement EQUAL TRUE
|listelement EQUAL FALSE
|CONST EQUAL FALSE
|CONST EQUAL TRUE
|CONST EQUAL LOGICALASSIGN logicalexp
|CONST EQUAL predicatecall
|CONST EQUAL NOT logicalexp
;
Definition: This allows us to initialize a variable. It can be done at the time of declaration.

ifElse: IF BRACKETS logicalexp BRACKETS LCB statementMore RCB
|IF BRACKETS logicalexp BRACKETS LCB statementMore RCB ELSE LCB statementMore RCB
;

Definiton: This determines how If-Else statements will be written. For example,

if ^^ a&b ^^ << c = d >>

while: WHILE BRACKETS logicalexp BRACKETS LCB statementMore RCB
;
Definition: This determines how while statements will be written. Syntax is very similar to

if-else statements.

input: INPUT FALSE
|INPUT TRUE
;
Definition: Input gets value from the users. Users are able to enter true or false.

logicvalue: TRUE
|FALSE
;
Definition: This determines the values of logic values. They can either be true or false as this

is a logic language.

type: IDENTIFIER
|CONST
|logicvalue
|listelement
;
Definition: This determines the types that variables could be. For example consts or arrays.

listelement: IDENTIFIER LCB NUMBER RCB
;
Definition: This allows to lists spesific element to be reached from us. For example 5th

element of the list.

list: type
| list COMMA type
;
Definition: This is for determining list. List is consisted of multiple elements.

createlist: IDENTIFIER LCB list RCB
;
Definition: This provides a feature of creating list in our language.

message: MESSAGE logicalexp
;
Definition: This gives the result of an logical expression to users. Basicly it prints the result.

logicalexp: ifandonlyifexp EQUALS logicalexp
|ifandonlyifexp
;
Definition: General specification for logical expressions. This is the root of logical expression hierarchy for providing precedence.
ifandonlyifexp: impliesexp IFANDONLYIF ifandonlyifexp
|impliesexp
;
Definition: This is for allowing if and only if operation in our programming language for providing precedence hierarchy it can consist implies operation too.

impliesexp: orexp IMPLIES impliesexp
|orexp
;
Definition: This is for allowing implies operation in our language but for precedence it also can be consisted with 'OR' operation.

orexp: andexp OR orexp
|andexp
;
Definition: This is for allowing or operation in our language. But for precedence it also can be consisted with and operation.

andexp: exp AND andexp
|exp
;
Definition: This is for determining and operation in our language.

exp: type
|BRACKETS logicalexp BRACKETS
;
Definition: This is for determining logical expresssion.

commentbegins: COMMENTBEGINS statements COMMENTFINISH
;
Definition: This allows our programming language to have comments.

4. TERMINALS IN YACC FILE (TOKENS)

COMMA

Definition: Stands for “,”.

BRACKETS

Definition: Instead of “(“ and “)” this “^^” notation is used.

LCB

Definition: Instead of “{“ this “<<” is used.

RCB

Definition: Instead of “}“ this “>>” is used.

COMMENTBEGINS

Definition: Comments begins with “<<<”.

COMMENTFINISH

Definition: Comments endings are determined with “>>>”.

COMMENT

Definition: Single line comments starts with “.

NUMBER

Definition: Digit is used for defining numbers or defining variable names.

WHILE

Definition: while loops are created with while keyword

ELSE

Definition: if else statements else part is created with else keyword.

IF

Definition: if else statements if part is created with if keyword.

NOT

Definition: Stands for logical not.

IMPLIES

Definition: Stands for logical implies.

AND

Definition: Stands for logical and.

IFANDONLYIF

Definition: Stands for logical if and only if.

OR

Definition: Stands for logical or.

EQUALS

Definition: Is used for checking both sides equal or not

IDENTIFIER

Definition: An identifier is a name that identifies variables in this language.