Annie Lane and Michaela Kerem

COMP2022: Formal Languages and Logic

Semester 1, 2014

Assignment 2: Report

**Part I:**

The grammar G is not LL(1) because of left factoring and left recursion.

Left factoring occurs in:

L → I **;** | I

It is resolved by adding another variable M and modifying the rule for L:

L → I M

M → **;** L | ε

Left factoring also occurs in:

E2 → T Op2 E2 | T

It is resolved by adding variable K and modifying the rule for E2:

E2 → T K

K → Op2 E2 | ε

Left recursion occurs in:

E → E Op1 E2 | E2

It is resolved by adding variable R and modifying the rule for E:  
 E → E2 R

R → Op1 E2 R | ε

The new grammar G’ is:  
P →L

L →I M

M → **;** L | 

I →A | C | W

A →**id :=** E

C →**if** E **then** L O **endif**

O →**else** L | 

W →**while** E **do** L **end**

E → E2 R

R →Op1 E2 R | 

E2 →T K

K →Op2 E2 |  

T →**c** | **id**

Op1  →**<** |**=** | **!=**

Op2  →**+** | **-**

**Part II:**

To build FIRST set for a variable, the left hand side of the production rules are considered. The FIRST set of each terminal is itself. If the FIRST set of a variable contains ε, then in order to build the parse table, the FOLLOW set must also be calculated for that variable. This entails examining the occurrences of the variable on the right hand side of the production rules and determining which terminals can occur after the variable. Often, in order to build FOLLOW sets for a variable, other FOLLOW sets must be built.

The grammar is LL(1) because in the parse table (see Part III) no entry has multiple rules.

Below are the FIRST and FOLLOW sets for all of the variables in the grammar.

|  |  |  |
| --- | --- | --- |
| Variable | FIRST set | FOLLOW set |
| P | { id, if, while } | { $ } |
| L | { id, if, while } | { $, else, endif, end } |
| M | { ε, ; } | { $, else, endif, end } |
| I | { id, if, while } | { $, ;, else, endif, end } |
| A | { id } | { $, ;, else, endif, end } |
| C | { if } | { $, ;, else, endif, end } |
| O | { ε, else } | { endif } |
| W | { while } | { $, ;, else, endif, end } |
| E | { c, id } | {$, ;, else, endif, do, then, end} |
| R | { ε, <, =, != } | {$, ;, else, endif, do, then <, =, !=, c, id, end} |
| E2 | { c, id } | {$, ;, else, endif, do, then, -, +,<, =, != , c, id, end} |
| K | { ε, +, - } | { $, ;, else, endif, do, then, -, +,<, =, != , c, id, end} |
| T | { c, id } | {$, ;, else, endif, do, then, -, +,<, =, != , c, id, end} |
| Op1 | { <, =, != } | { c, id } |
| Op2 | { +, - } | { c, id } |

Building FOLLOW sets:

FOLLOW(P) = {$}

FOLLOW(L) = FOLLOW(P)U FOLLOW(M) FIRST(O)-{ ε} U FOLLOW(O) U {end}

FOLLOW(M) = FOLLOW(L)

FOLLOW(I) = FIRST(M) U FOLLOW(M)

FOLLOW(A) = FOLLOW(I)

FOLLOW(C) = FOLLOW(I)

FOLLOW(O) = {endif}

FOLLOW(W) = FOLLOW(I)

FOLLOW(E) = {do, then} U FOLLOW(A)

FOLLOW(R) = FIRST(R) U FIRST(E) U FOLLOW(R) U FOLLOW(E)

FOLLOW(E2) = FOLLOW(K) U FIRST(K) U FIRST(R) U FOLLOW(R)

FOLLOW(K) = FOLLOW(E2)

FOLLOW(T) = FIRST(K) U FOLLOW(K)

FOLLOW(Op1) = FIRST(E2)

FOLLOW(Op2) = FIRST(E2)

**Part III:**The parse table is built by using the FIRST and FOLLOW sets of the grammar. The table contains all of the variables in the left most columns and all of the terminals plus ‘$’ in the top most row. For each variable, the FIRST set is always used and the FOLLOW set is used if the FIRST set contains ε. The rule in the grammar yielding that terminal in the set is entered into the table.

*Example 1:*

Building the Parse Table along the Variable P.

Given:

FIRST(P) = { id, if, while }

To reach these terminals, the following rule must be used:

P →L

So in (P, id) (P, if) and (P, while) enter: P →L

*Example 2:*

Building the Parse table along Variable M.

Given:

FIRST(M) = { ε, ; }

Because ε is in the FIRST set, we will also need to look at the FOLLOW set:

FOLLOW(M) = { $, endif, else, end }

To reach ‘;’ use the following rule and enter it into (M, ;):

M->;L

To reach ‘$’, ‘endif’, and ‘end’ enter the rule into (M, $), (M, endif), (M, else) and (M,end):

M->ε

See Parse Table on next page.

Parse Table

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | id | if | while | ; | else | c | then | < | = | != | + | - | do | end | endif | $ |
| P | P->L | P->L | P->L |  |  |  |  |  |  |  |  |  |  |  |  |  |
| L | L->IM | L->IM | L->IM |  |  |  |  |  |  |  |  |  |  |  |  |  |
| M |  |  |  | M->;L | M->ε |  |  |  |  |  |  |  |  | M->ε | M->ε | M->ε |
| I | I->A | I->C | I->W |  |  |  |  |  |  |  |  |  |  |  |  |  |
| A | A->  id := E |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| C |  | C->  if E then L O endif |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| O |  |  |  |  | O->  else L |  |  |  |  |  |  |  |  |  | O->ε |  |
| W |  |  | W->  while E do L end |  |  |  |  |  |  |  |  |  |  |  |  |  |
| E | E->  E2 R |  |  |  |  | E->  E2 R |  |  |  |  |  |  |  |  |  |  |
| R | R->ε |  |  | R->ε | R->ε | R-> ε | R->ε | R->  Op1 E2 R | R->  Op1 E2 R | R->  Op1 E2 R | R->ε | R->ε | R->ε | K->ε | R->ε | R->ε |
| E2 | E2-> T K |  |  |  |  | E2-> T K |  |  |  |  |  |  |  |  |  |  |
| K | K-> ε |  |  | K-> ε |  | K-> ε | K-> ε | K->ε | K->ε | K->ε | K->  Op2 E2 | K->  Op2 E2 | K->ε | K->ε | K->ε | K-> ε |
| T | T->  id |  |  |  |  | T->  c |  |  |  |  |  |  |  |  |  |  |
| Op1 |  |  |  |  |  |  |  | Op1-> < | Op1-> = | Op1-> != |  |  |  |  |  |  |
| Op2 |  |  |  |  |  |  |  |  |  |  | Op2->  + | Op2-> - |  |  |  |  |

**Part IV:**

This includes an error recovery feature. After stating the error and what the parser expected, a warning is given. The warning indicates that the parser will attempt to continue to parse the input. In order to continue, the parser discards the current token, the source of the error. Based on the current variable on the top of the stack, a list of “acceptable” terminals for the grammar is created. The first token in the input is then compared to the “acceptable” terminals. If there is a match, then the parser proceeds. If the token does not match any of the “acceptable” terminals in the generated list, then the token is discarded from the input. This comparison process between input tokens and “acceptable” terminals continues until there is a match.

**LL\_1\_Parser.py**

#----------------------------Introduction-----------------------------------#

#Written by Annie Lane and Michaela Kerem

#For Assignment 2 for COMP2022: Formal Languages and Logic

#Semester 1, 2014

#LL(1) Parser for Grammar G'

#Reads input from a .txt file

#Prints line showing remaining input and stack for each step

#Determines if input is accepted or rejected

#Provides error messages when incorrect token found

#Includes error recovery feature

#This program is organized into the following sections

# Input Processing

# Derivation and Stack Printing

# Modifying the Stack

# Parse Table Functions

# Error Message Function

# Parser Function

# Intialize Parse Table

# Program

#----------------------------Input Processing-------------------------------#

#This also includes functions necessary to read input .txt files

#Read .txt file by name of file

#remove leading and following white space

#remove newline characters

#concatenate 'id :=' into 'id:=' so there is no space

#split along spaces to create separate tokens

#concatenate list in tokens with the input\_string

#add '$' to the end of the list to indicate end of input string

def read\_input\_file(file\_name):

file\_object = open(file\_name, 'r')

input\_string = []

temp\_string = []

temp\_string = file\_object.readlines()

for token in temp\_string:

token = token.strip()

token = token.replace('\n','')

token = token.split(' ')

input\_string = input\_string + token

file\_object.close()

input\_string.append('$')

return input\_string

#Requests a filename from user

def user\_input\_filename():

file\_name = input("Please enter the name of a file to parse: ")

if file\_name.lower() == "exit":

return "exit"

input\_string = read\_input\_file(file\_name.lower())

return input\_string

#---------------------Derivation and Stack Printing------------------------#

#Format the strings properly

#Print current state of stack and input string remaining

def print\_current\_state(stack,in\_string,buffer\_len):

input\_string = "".join(in\_string)

stack\_string = "".join(stack)

print(input\_string.ljust(buffer\_len) + stack\_string)

#Print header

def print\_header(buffer\_len):

print("INPUT STRING REMAINING".ljust(buffer\_len) + "STACK")

#----------------------Modifying the Stack---------------------------------#

#We will use lists as our stack

#We will call it: stack

#It will be intialized with '$'

#Function initializes the stack with '$'

#Call at the beginning (each time a new string is input to be considered)

def init\_stack():

stack = []

stack.insert(0,'$')

stack.insert(0,'P')

return stack

def pop\_stack(T,stack):

stack.remove(T)

return stack

#----------------------Parse Table Functions-------------------------------#

#Table\_Entry Class has 3 member variables

#variable is the variable read off the stack as a string

#Terminal is the input symbol read as a string

#Alpha is a list of of strings composed of Terminals and/or Variables

class Table\_Entry:

def \_\_init\_\_(self,var,term,a):

self.variable = var

self.terminal = term

self.alpha = a

#Member function prints the entry in desired format

def print\_entry(self):

print("P[" + self.variable + "," + self.terminal + "] yields " + self.variable + " -> " + ''.join(self.alpha))

#Prints all entries in the Table

def print\_table(Table):

for example in Table:

example.print\_entry()

#------------------------Functions for Parser ---------------------------#

#Extract top symbol from stack

def get\_top\_stack(stack):

if stack != []:

return stack[0]

else:

return False

#Extract the current input symbol from input string

def get\_cur\_in\_sym(in\_string):

if in\_string != []:

return in\_string[0]

else:

return False

#True when T and I match to '$' (so the input is accepted)

def is\_accepted(T,I):

if T == '$' and I == '$':

return True

else:

return False

#compares top of stack to list of terminals to see if it is a terminal

#Otherwise, produces an error message

def is\_terminal(T,terminals):

for term in terminals:

if T == term:

return True

else:

return False

#will check if terminals match

def is\_matching\_terminal(T,I):

if T == I:

return True

else:

return False

#Removes I from the the in\_string

#"Consumes" it

def consume\_I(I,in\_string):

in\_string.remove(I)

return in\_string

def print\_error(T,I):

print("P[" + T + "," + I + "] does not have corresponding rule in Parse Table for this Grammar")

#Iterates through table in search of matching Terminal and Variable Pair

#Returns entry if match is found

def look\_in\_table(T,I,Table):

for entry in Table:

if T == entry.variable and I == entry.terminal:

#sentry.print\_entry()

return entry

#print\_error(T,I)

return False

#Checks if entry exists

def entry\_exists(entry):

if entry == False:

return False

else:

return True

#According to the entry in parse table, pushes alpha (in reverse) to the stack

def push\_alpha(entry,stack):

if entry.alpha == ['Epsilon']:

return stack

alpha\_reverse = list(entry.alpha)

alpha\_reverse.reverse()

for val in alpha\_reverse:

stack.insert(0,val)

return stack

#-------------------------Error Message Functions---------------------------#

#When input is not accepted

#Example: “expected a ‘;’ instead of ‘if’”

#Prints appropriate error message based on top of stack and current input token value

def error\_message(T, I, Table):

expected\_terminals = alternative\_terminals(T,Table)

print("")

print("Input is REJECTED")

print("Error: There is no entry in table for (" + T + ", " + I + "). Expected: '" + "' '".join(expected\_terminals) + "'")

print("")

#Creates a list from Parse Table of acceptable terminals from input

def alternative\_terminals(T,Table):

expected\_terminals = []

for entry in Table:

if entry.variable == T:

if entry.terminal != 'Epsilon' and entry.terminal != '$':

expected\_terminals.append(entry.terminal)

return expected\_terminals

#Uses the acceptable terminals to find matches so can continue parsing

def find\_good\_token(in\_string, expected\_terminals):

count = 0

stop = 'no'

clean\_string = list(in\_string)

for token in in\_string:

for possible in expected\_terminals:

if token == possible:

return clean\_string

clean\_string.remove(token)

count = count + 1

return clean\_string

#Combines find\_good\_token and alternative\_terminals into one neat function

def error\_recovery(in\_string, T, I, Table, buffer\_len):

print("Warning: See error above. Will attempt to continue by discarding bad input tokens")

print("")

expected\_terminals = alternative\_terminals(T,Table)

in\_string = find\_good\_token(in\_string, expected\_terminals)

if len(in\_string) > 0:

print\_header(buffer\_len)

else:

print("Could not recover from error")

print("")

return in\_string

#----------------------------------Parser Function-------------------------#

#This function parses the input string according to the parse table

#Implements psuedocode seen in lecture slides

def parse\_input(in\_string,Table):

count = 0

stack = init\_stack()

buffer\_len = len("".join(in\_string)) + 10

print\_header(buffer\_len)

#To prevent infinte loops

max\_count = 100

while len(in\_string) > 0 and count < max\_count:

T = get\_top\_stack(stack)

I = get\_cur\_in\_sym(in\_string)

print\_current\_state(stack,in\_string, buffer\_len)

if is\_accepted(T,I):

print("Input String is ACCEPTED")

break

elif T == I:

stack = pop\_stack(T,stack)

in\_string = consume\_I(I, in\_string)

elif is\_terminal(T, terminals):

if is\_matching\_terminal(T, I):

stack = pop\_stack(T, stack)

in\_string = consume\_I(I, in\_string)

else:

error\_message(T,I,Table)

in\_string = error\_recovery(in\_string, T, I, Table,buffer\_len)

else:

entry = look\_in\_table(T, I, Table)

if entry\_exists(entry):

stack = pop\_stack(T, stack)

stack = push\_alpha(entry,stack)

else:

error\_message(T,I,Table)

in\_string = error\_recovery(in\_string, T, I, Table, buffer\_len)

count = count + 1

#---------------------Initialize Parse Table-----------------------------#

#List of Table\_Entry Objects

Table = []

#Set all entries of the table

Table.append(Table\_Entry('P','if',['L']))

Table.append(Table\_Entry('P','while',['L']))

Table.append(Table\_Entry('P','id',['L']))

Table.append(Table\_Entry('L','if',['I','M']))

Table.append(Table\_Entry('L','while',['I','M']))

Table.append(Table\_Entry('L','id',['I','M']))

Table.append(Table\_Entry('M',';',[';','L']))

Table.append(Table\_Entry('M','else',['Epsilon']))

Table.append(Table\_Entry('M','end',['Epsilon']))

Table.append(Table\_Entry('M','endif',['Epsilon']))

Table.append(Table\_Entry('M','$',['Epsilon']))

Table.append(Table\_Entry('I','if',['C']))

Table.append(Table\_Entry('I','while',['W']))

Table.append(Table\_Entry('I','id',['A']))

Table.append(Table\_Entry('A','id',['id',':=','E']))

Table.append(Table\_Entry('C','if',['if','E','then','L','O','endif']))

Table.append(Table\_Entry('O','else',['else','L']))

Table.append(Table\_Entry('O','endif',['Epsilon']))

Table.append(Table\_Entry('W','while',['while','E','do','L','end']))

Table.append(Table\_Entry('E','c',['E2','R']))

Table.append(Table\_Entry('E','id',['E2','R']))

Table.append(Table\_Entry('R',';',['Epsilon']))

Table.append(Table\_Entry('R','<',['Op1','E2','R']))

Table.append(Table\_Entry('R','=',['Op1','E2','R']))

Table.append(Table\_Entry('R','!=',['Op1','E2','R']))

Table.append(Table\_Entry('R','do',['Epsilon']))

Table.append(Table\_Entry('R','endif',['Epsilon']))

Table.append(Table\_Entry('R','then',['Epsilon']))

Table.append(Table\_Entry('R','$',['Epsilon']))

Table.append(Table\_Entry('R','then',['Epsilon']))

Table.append(Table\_Entry('R','else',['Epsilon']))

Table.append(Table\_Entry('R','-',['Epsilon']))

Table.append(Table\_Entry('R','+',['Epsilon']))

Table.append(Table\_Entry('R','c',['Epsilon']))

Table.append(Table\_Entry('R','id',['Epsilon']))

Table.append(Table\_Entry('R','end',['Epsilon']))

Table.append(Table\_Entry('E2','c',['T','K']))

Table.append(Table\_Entry('E2','id',['T','K']))

Table.append(Table\_Entry('K',';',['Epsilon']))

Table.append(Table\_Entry('K','<',['Epsilon']))

Table.append(Table\_Entry('K','=',['Epsilon']))

Table.append(Table\_Entry('K','!=',['Epsilon']))

Table.append(Table\_Entry('K','+',['Op2','E2']))

Table.append(Table\_Entry('K','-',['Op2','E2']))

Table.append(Table\_Entry('K','do',['Epsilon']))

Table.append(Table\_Entry('K','endif',['Epsilon']))

Table.append(Table\_Entry('K','then',['Epsilon']))

Table.append(Table\_Entry('K','else',['Epsilon']))

Table.append(Table\_Entry('K','c',['Epsilon']))

Table.append(Table\_Entry('K','id',['Epsilon']))

Table.append(Table\_Entry('K','$',['Epsilon']))

Table.append(Table\_Entry('K','end',['Epsilon']))

Table.append(Table\_Entry('T','c',['c']))

Table.append(Table\_Entry('T','id',['id']))

Table.append(Table\_Entry('Op1','<',['<']))

Table.append(Table\_Entry('Op1','=',['=']))

Table.append(Table\_Entry('Op1','!=',['!=']))

Table.append(Table\_Entry('Op2','+',['+']))

Table.append(Table\_Entry('Op2','-',['-']))

#These might not be necessary, but they are here at least for reference

#Basically, it helps to define our grammar

variables = ['P','L','M','I','A','C','O','W','E','R','E2','K','T','Op1','Op2']

terminals = ['id', 'if', 'while', ';', 'else', 'c', '<', '=', '!=', '+', '-','do','end','endif',':=','then']

#---------------------------------Program-----------------------------#

#Run program until "exit" is entered by the user to break the loop

while 1 < 2:

in\_string = user\_input\_filename()

if in\_string == "exit":

break

parse\_input(in\_string,Table)

print("Program terminating")

**Part V:**

If a rule is to be added or modified, then the appropriate “Table\_Entry” objects must be modified in the “Table.” A “Table\_Entry” is a custom class object which contains a variable, a terminal, and a list of variables and/or terminals that are put on the stack if the top of the stack is the “Table\_Entry” variable and the current input symbol is the terminal of the entry. For example, if we had an entry into the parse table of (S, ) with the rule S->T J, then it we would add the following line:

Table.append(Table\_Entry(‘S’,’\*’,[‘T’,’J’])

Also, the list “variables” and “terminals” must be updated to include any newly introduced or removed variables or terminals.

The Table is built in lines approximately 270 to 350 of the Python File. In this report, the Parse Table initialization code starts on page 11 and ends on page 13.

**Part VI:**

The parser is tested with various examples:

accept.txt

accept\_other\_terminals.txt

error\_missing\_comma.txt

error\_follow\_do.txt

accept\_semicolon.txt

recovery\_demo.txt

The parser needs to be tested with inputs including all of the terminals. It’s particularly important to test examples that are not expected to be accepted despite including only valid terminals. Also, demonstrating error recovery techniques is important.

accept.txt

if c < id then

id := c

else

while c < c + id do

id := id

end

endif

The input is expected to be accepted. The output below demonstrates that it is accepted.

Output:

INPUT STRING REMAINING STACK

ifc<idthenid:=celsewhilec<c+iddoid:=idendendif$ P$

ifc<idthenid:=celsewhilec<c+iddoid:=idendendif$ L$

ifc<idthenid:=celsewhilec<c+iddoid:=idendendif$ IM$

ifc<idthenid:=celsewhilec<c+iddoid:=idendendif$ CM$

ifc<idthenid:=celsewhilec<c+iddoid:=idendendif$ ifEthenLOendifM$

c<idthenid:=celsewhilec<c+iddoid:=idendendif$ EthenLOendifM$

c<idthenid:=celsewhilec<c+iddoid:=idendendif$ E2RthenLOendifM$

c<idthenid:=celsewhilec<c+iddoid:=idendendif$ TKRthenLOendifM$

c<idthenid:=celsewhilec<c+iddoid:=idendendif$ cKRthenLOendifM$

<idthenid:=celsewhilec<c+iddoid:=idendendif$ KRthenLOendifM$

<idthenid:=celsewhilec<c+iddoid:=idendendif$ RthenLOendifM$

<idthenid:=celsewhilec<c+iddoid:=idendendif$ Op1E2RthenLOendifM$

<idthenid:=celsewhilec<c+iddoid:=idendendif$ <E2RthenLOendifM$

idthenid:=celsewhilec<c+iddoid:=idendendif$ E2RthenLOendifM$

idthenid:=celsewhilec<c+iddoid:=idendendif$ TKRthenLOendifM$

idthenid:=celsewhilec<c+iddoid:=idendendif$ idKRthenLOendifM$

thenid:=celsewhilec<c+iddoid:=idendendif$ KRthenLOendifM$

thenid:=celsewhilec<c+iddoid:=idendendif$ RthenLOendifM$

thenid:=celsewhilec<c+iddoid:=idendendif$ thenLOendifM$

id:=celsewhilec<c+iddoid:=idendendif$ LOendifM$

id:=celsewhilec<c+iddoid:=idendendif$ IMOendifM$

id:=celsewhilec<c+iddoid:=idendendif$ AMOendifM$

id:=celsewhilec<c+iddoid:=idendendif$ id:=EMOendifM$

:=celsewhilec<c+iddoid:=idendendif$ :=EMOendifM$

celsewhilec<c+iddoid:=idendendif$ EMOendifM$

celsewhilec<c+iddoid:=idendendif$ E2RMOendifM$

celsewhilec<c+iddoid:=idendendif$ TKRMOendifM$

celsewhilec<c+iddoid:=idendendif$ cKRMOendifM$

elsewhilec<c+iddoid:=idendendif$ KRMOendifM$

elsewhilec<c+iddoid:=idendendif$ RMOendifM$

elsewhilec<c+iddoid:=idendendif$ MOendifM$

elsewhilec<c+iddoid:=idendendif$ OendifM$

elsewhilec<c+iddoid:=idendendif$ elseLendifM$

whilec<c+iddoid:=idendendif$ LendifM$

whilec<c+iddoid:=idendendif$ IMendifM$

whilec<c+iddoid:=idendendif$ WMendifM$

whilec<c+iddoid:=idendendif$ whileEdoLendMendifM$

c<c+iddoid:=idendendif$ EdoLendMendifM$

c<c+iddoid:=idendendif$ E2RdoLendMendifM$

c<c+iddoid:=idendendif$ TKRdoLendMendifM$

c<c+iddoid:=idendendif$ cKRdoLendMendifM$

<c+iddoid:=idendendif$ KRdoLendMendifM$

<c+iddoid:=idendendif$ RdoLendMendifM$

<c+iddoid:=idendendif$ Op1E2RdoLendMendifM$

<c+iddoid:=idendendif$ <E2RdoLendMendifM$

c+iddoid:=idendendif$ E2RdoLendMendifM$

c+iddoid:=idendendif$ TKRdoLendMendifM$

c+iddoid:=idendendif$ cKRdoLendMendifM$

+iddoid:=idendendif$ KRdoLendMendifM$

+iddoid:=idendendif$ Op2E2RdoLendMendifM$

+iddoid:=idendendif$ +E2RdoLendMendifM$

iddoid:=idendendif$ E2RdoLendMendifM$

iddoid:=idendendif$ TKRdoLendMendifM$

iddoid:=idendendif$ idKRdoLendMendifM$

doid:=idendendif$ KRdoLendMendifM$

doid:=idendendif$ RdoLendMendifM$

doid:=idendendif$ doLendMendifM$

id:=idendendif$ LendMendifM$

id:=idendendif$ IMendMendifM$

id:=idendendif$ AMendMendifM$

id:=idendendif$ id:=EMendMendifM$

:=idendendif$ :=EMendMendifM$

idendendif$ EMendMendifM$

idendendif$ E2RMendMendifM$

idendendif$ TKRMendMendifM$

idendendif$ idKRMendMendifM$

endendif$ KRMendMendifM$

endendif$ RMendMendifM$

endendif$ MendMendifM$

endendif$ endMendifM$

endif$ MendifM$

endif$ endifM$

$ M$

$ $

Input String is ACCEPTED

accept\_other\_terminals.txt:

if c = id then

id := c

else

while c != c - id do

id := id

end

endif  
This input is expected to be accepted. It includes the 3 of the 4 terminals not included in accept.txt to ensure these terminals can be parsed.

INPUT STRING REMAINING STACK

ifc=idthenid:=celsewhilec!=c-iddoid:=idendendif$ P$

ifc=idthenid:=celsewhilec!=c-iddoid:=idendendif$ L$

ifc=idthenid:=celsewhilec!=c-iddoid:=idendendif$ IM$

ifc=idthenid:=celsewhilec!=c-iddoid:=idendendif$ CM$

ifc=idthenid:=celsewhilec!=c-iddoid:=idendendif$ ifEthenLOendifM$

c=idthenid:=celsewhilec!=c-iddoid:=idendendif$ EthenLOendifM$

c=idthenid:=celsewhilec!=c-iddoid:=idendendif$ E2RthenLOendifM$

c=idthenid:=celsewhilec!=c-iddoid:=idendendif$ TKRthenLOendifM$

c=idthenid:=celsewhilec!=c-iddoid:=idendendif$ cKRthenLOendifM$

=idthenid:=celsewhilec!=c-iddoid:=idendendif$ KRthenLOendifM$

=idthenid:=celsewhilec!=c-iddoid:=idendendif$ RthenLOendifM$

=idthenid:=celsewhilec!=c-iddoid:=idendendif$ Op1E2RthenLOendifM$

=idthenid:=celsewhilec!=c-iddoid:=idendendif$ =E2RthenLOendifM$

idthenid:=celsewhilec!=c-iddoid:=idendendif$ E2RthenLOendifM$

idthenid:=celsewhilec!=c-iddoid:=idendendif$ TKRthenLOendifM$

idthenid:=celsewhilec!=c-iddoid:=idendendif$ idKRthenLOendifM$

thenid:=celsewhilec!=c-iddoid:=idendendif$ KRthenLOendifM$

thenid:=celsewhilec!=c-iddoid:=idendendif$ RthenLOendifM$

thenid:=celsewhilec!=c-iddoid:=idendendif$ thenLOendifM$

id:=celsewhilec!=c-iddoid:=idendendif$ LOendifM$

id:=celsewhilec!=c-iddoid:=idendendif$ IMOendifM$

id:=celsewhilec!=c-iddoid:=idendendif$ AMOendifM$

id:=celsewhilec!=c-iddoid:=idendendif$ id:=EMOendifM$

:=celsewhilec!=c-iddoid:=idendendif$ :=EMOendifM$

celsewhilec!=c-iddoid:=idendendif$ EMOendifM$

celsewhilec!=c-iddoid:=idendendif$ E2RMOendifM$

celsewhilec!=c-iddoid:=idendendif$ TKRMOendifM$

celsewhilec!=c-iddoid:=idendendif$ cKRMOendifM$

elsewhilec!=c-iddoid:=idendendif$ KRMOendifM$

elsewhilec!=c-iddoid:=idendendif$ RMOendifM$

elsewhilec!=c-iddoid:=idendendif$ MOendifM$

elsewhilec!=c-iddoid:=idendendif$ OendifM$

elsewhilec!=c-iddoid:=idendendif$ elseLendifM$

whilec!=c-iddoid:=idendendif$ LendifM$

whilec!=c-iddoid:=idendendif$ IMendifM$

whilec!=c-iddoid:=idendendif$ WMendifM$

whilec!=c-iddoid:=idendendif$ whileEdoLendMendifM$

c!=c-iddoid:=idendendif$ EdoLendMendifM$

c!=c-iddoid:=idendendif$ E2RdoLendMendifM$

c!=c-iddoid:=idendendif$ TKRdoLendMendifM$

c!=c-iddoid:=idendendif$ cKRdoLendMendifM$

!=c-iddoid:=idendendif$ KRdoLendMendifM$

!=c-iddoid:=idendendif$ RdoLendMendifM$

!=c-iddoid:=idendendif$ Op1E2RdoLendMendifM$

!=c-iddoid:=idendendif$ !=E2RdoLendMendifM$

c-iddoid:=idendendif$ E2RdoLendMendifM$

c-iddoid:=idendendif$ TKRdoLendMendifM$

c-iddoid:=idendendif$ cKRdoLendMendifM$

-iddoid:=idendendif$ KRdoLendMendifM$

-iddoid:=idendendif$ Op2E2RdoLendMendifM$

-iddoid:=idendendif$ -E2RdoLendMendifM$

iddoid:=idendendif$ E2RdoLendMendifM$

iddoid:=idendendif$ TKRdoLendMendifM$

iddoid:=idendendif$ idKRdoLendMendifM$

doid:=idendendif$ KRdoLendMendifM$

doid:=idendendif$ RdoLendMendifM$

doid:=idendendif$ doLendMendifM$

id:=idendendif$ LendMendifM$

id:=idendendif$ IMendMendifM$

id:=idendendif$ AMendMendifM$

id:=idendendif$ id:=EMendMendifM$

:=idendendif$ :=EMendMendifM$

idendendif$ EMendMendifM$

idendendif$ E2RMendMendifM$

idendendif$ TKRMendMendifM$

idendendif$ idKRMendMendifM$

endendif$ KRMendMendifM$

endendif$ RMendMendifM$

endendif$ MendMendifM$

endendif$ endMendifM$

endif$ MendifM$

endif$ endifM$

$ M$

$ $

Input String is ACCEPTED

error\_missing\_semicolon.txt:

This input is expected to be rejected. The output does indicate that the input is rejected. This is because a ‘;’ was expected between the c’s in order to correctly parse the string.

INPUT STRING REMAINING STACK

id:=id+cc:=c+id$ P$

id:=id+cc:=c+id$ L$

id:=id+cc:=c+id$ IM$

id:=id+cc:=c+id$ AM$

id:=id+cc:=c+id$ id:=EM$

:=id+cc:=c+id$ :=EM$

id+cc:=c+id$ EM$

id+cc:=c+id$ E2RM$

id+cc:=c+id$ TKRM$

id+cc:=c+id$ idKRM$

+cc:=c+id$ KRM$

+cc:=c+id$ Op2E2RM$

+cc:=c+id$ +E2RM$

cc:=c+id$ E2RM$

cc:=c+id$ TKRM$

cc:=c+id$ cKRM$

c:=c+id$ KRM$

c:=c+id$ RM$

c:=c+id$ M$

Input is REJECTED

Error: There is no entry in table for (M, c). Expected: ';' 'else' 'end' 'endif'

Warning: See error above. Will attempt to continue by discarding bad input tokens

Could not recover from error

error\_follow\_do.txt

Expect rejection because there is nothing between “do” and “end”. The parser expects an operation to occur after the do.

INPUT STRING REMAINING STACK

ifc+c=idthenwhiledoend$ P$

ifc+c=idthenwhiledoend$ L$

ifc+c=idthenwhiledoend$ IM$

ifc+c=idthenwhiledoend$ CM$

ifc+c=idthenwhiledoend$ ifEthenLOendifM$

c+c=idthenwhiledoend$ EthenLOendifM$

c+c=idthenwhiledoend$ E2RthenLOendifM$

c+c=idthenwhiledoend$ TKRthenLOendifM$

c+c=idthenwhiledoend$ cKRthenLOendifM$

+c=idthenwhiledoend$ KRthenLOendifM$

+c=idthenwhiledoend$ Op2E2RthenLOendifM$

+c=idthenwhiledoend$ +E2RthenLOendifM$

c=idthenwhiledoend$ E2RthenLOendifM$

c=idthenwhiledoend$ TKRthenLOendifM$

c=idthenwhiledoend$ cKRthenLOendifM$

=idthenwhiledoend$ KRthenLOendifM$

=idthenwhiledoend$ RthenLOendifM$

=idthenwhiledoend$ Op1E2RthenLOendifM$

=idthenwhiledoend$ =E2RthenLOendifM$

idthenwhiledoend$ E2RthenLOendifM$

idthenwhiledoend$ TKRthenLOendifM$

idthenwhiledoend$ idKRthenLOendifM$

thenwhiledoend$ KRthenLOendifM$

thenwhiledoend$ RthenLOendifM$

thenwhiledoend$ thenLOendifM$

whiledoend$ LOendifM$

whiledoend$ IMOendifM$

whiledoend$ WMOendifM$

whiledoend$ whileEdoLendMOendifM$

doend$ EdoLendMOendifM$

Input is REJECTED

Error: There is no entry in table for (E, do). Expected: 'c' 'id'

Warning: See error above. Will attempt to continue by discarding bad input tokens

Could not recover from error

accept\_semicolon.txt

Shows that input in “error\_missing\_semicolon.txt” can be resolved by adding a semicolon between the spaces. This is expected to be accepted.

INPUT STRING REMAINING STACK

id:=id+c;id:=c+id$ P$

id:=id+c;id:=c+id$ L$

id:=id+c;id:=c+id$ IM$

id:=id+c;id:=c+id$ AM$

id:=id+c;id:=c+id$ id:=EM$

:=id+c;id:=c+id$ :=EM$

id+c;id:=c+id$ EM$

id+c;id:=c+id$ E2RM$

id+c;id:=c+id$ TKRM$

id+c;id:=c+id$ idKRM$

+c;id:=c+id$ KRM$

+c;id:=c+id$ Op2E2RM$

+c;id:=c+id$ +E2RM$

c;id:=c+id$ E2RM$

c;id:=c+id$ TKRM$

c;id:=c+id$ cKRM$

;id:=c+id$ KRM$

;id:=c+id$ RM$

;id:=c+id$ M$

;id:=c+id$ ;L$

id:=c+id$ L$

id:=c+id$ IM$

id:=c+id$ AM$

id:=c+id$ id:=EM$

:=c+id$ :=EM$

c+id$ EM$

c+id$ E2RM$

c+id$ TKRM$

c+id$ cKRM$

+id$ KRM$

+id$ Op2E2RM$

+id$ +E2RM$

id$ E2RM$

id$ TKRM$

id$ idKRM$

$ KRM$

$ RM$

$ M$

$ $

Input String is ACCEPTED

recovery\_demo.txt

Expect rejection because of random ‘+‘. However, the error recovery implemented is able to ignore this erroneous character after issuing a warning and then continues to parse the rest of the input. As a result, the input is conditionally accepted

INPUT STRING REMAINING STACK

ifc=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ P$

ifc=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ L$

ifc=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ IM$

ifc=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ CM$

ifc=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ ifEthenLOendifM$

c=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ EthenLOendifM$

c=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ E2RthenLOendifM$

c=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ TKRthenLOendifM$

c=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ cKRthenLOendifM$

=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ KRthenLOendifM$

=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ RthenLOendifM$

=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ Op1E2RthenLOendifM$

=idthenid:=celsewhilec!=c-+iddoid:=idendendif$ =E2RthenLOendifM$

idthenid:=celsewhilec!=c-+iddoid:=idendendif$ E2RthenLOendifM$

idthenid:=celsewhilec!=c-+iddoid:=idendendif$ TKRthenLOendifM$

idthenid:=celsewhilec!=c-+iddoid:=idendendif$ idKRthenLOendifM$

thenid:=celsewhilec!=c-+iddoid:=idendendif$ KRthenLOendifM$

thenid:=celsewhilec!=c-+iddoid:=idendendif$ RthenLOendifM$

thenid:=celsewhilec!=c-+iddoid:=idendendif$ thenLOendifM$

id:=celsewhilec!=c-+iddoid:=idendendif$ LOendifM$

id:=celsewhilec!=c-+iddoid:=idendendif$ IMOendifM$

id:=celsewhilec!=c-+iddoid:=idendendif$ AMOendifM$

id:=celsewhilec!=c-+iddoid:=idendendif$ id:=EMOendifM$

:=celsewhilec!=c-+iddoid:=idendendif$ :=EMOendifM$

celsewhilec!=c-+iddoid:=idendendif$ EMOendifM$

celsewhilec!=c-+iddoid:=idendendif$ E2RMOendifM$

celsewhilec!=c-+iddoid:=idendendif$ TKRMOendifM$

celsewhilec!=c-+iddoid:=idendendif$ cKRMOendifM$

elsewhilec!=c-+iddoid:=idendendif$ KRMOendifM$

elsewhilec!=c-+iddoid:=idendendif$ RMOendifM$

elsewhilec!=c-+iddoid:=idendendif$ MOendifM$

elsewhilec!=c-+iddoid:=idendendif$ OendifM$

elsewhilec!=c-+iddoid:=idendendif$ elseLendifM$

whilec!=c-+iddoid:=idendendif$ LendifM$

whilec!=c-+iddoid:=idendendif$ IMendifM$

whilec!=c-+iddoid:=idendendif$ WMendifM$

whilec!=c-+iddoid:=idendendif$ whileEdoLendMendifM$

c!=c-+iddoid:=idendendif$ EdoLendMendifM$

c!=c-+iddoid:=idendendif$ E2RdoLendMendifM$

c!=c-+iddoid:=idendendif$ TKRdoLendMendifM$

c!=c-+iddoid:=idendendif$ cKRdoLendMendifM$

!=c-+iddoid:=idendendif$ KRdoLendMendifM$

!=c-+iddoid:=idendendif$ RdoLendMendifM$

!=c-+iddoid:=idendendif$ Op1E2RdoLendMendifM$

!=c-+iddoid:=idendendif$ !=E2RdoLendMendifM$

c-+iddoid:=idendendif$ E2RdoLendMendifM$

c-+iddoid:=idendendif$ TKRdoLendMendifM$

c-+iddoid:=idendendif$ cKRdoLendMendifM$

-+iddoid:=idendendif$ KRdoLendMendifM$

-+iddoid:=idendendif$ Op2E2RdoLendMendifM$

-+iddoid:=idendendif$ -E2RdoLendMendifM$

+iddoid:=idendendif$ E2RdoLendMendifM$

Input is REJECTED

Error: There is no entry in table for (E2, +). Expected: 'c' 'id'

Warning: See error above. Will attempt to continue by discarding bad input tokens

INPUT STRING REMAINING STACK

iddoid:=idendendif$ E2RdoLendMendifM$

iddoid:=idendendif$ TKRdoLendMendifM$

iddoid:=idendendif$ idKRdoLendMendifM$

doid:=idendendif$ KRdoLendMendifM$

doid:=idendendif$ RdoLendMendifM$

doid:=idendendif$ doLendMendifM$

id:=idendendif$ LendMendifM$

id:=idendendif$ IMendMendifM$

id:=idendendif$ AMendMendifM$

id:=idendendif$ id:=EMendMendifM$

:=idendendif$ :=EMendMendifM$

idendendif$ EMendMendifM$

idendendif$ E2RMendMendifM$

idendendif$ TKRMendMendifM$

idendendif$ idKRMendMendifM$

endendif$ KRMendMendifM$

endendif$ RMendMendifM$

endendif$ MendMendifM$

endendif$ endMendifM$

endif$ MendifM$

endif$ endifM$

$ M$

$ $

Input String is ACCEPTED