How to Create a Programming Language

LECTURE 6: RECURSIVE DECENT PARSING

1

LECTURE OUTLINE

- **RECURSIVE DECENT PARSING**
  - **GRAMMAR MUST BE NON-LEFT RECURSIVE**
  - **NO COMMON PREFIXES**
  - **EASY TO WRITE**
    - **ONE METHOD FOR EACH 'NON-TERMINAL'**
    - **STACK BASED**
    - **BASED ON INCOMING TOKEN, DECIDES**
      - **MATCH THE TOKEN AGAINST EXPECTED TOKEN**
      - **PICK EXPANSION FOR CURRENT NON-TOKEN**

2

WHAT ABOUT SRP?

- TURNS HOW TO BE DIFFICULT/IMPOSSIBLE TO CREATE AN 'LL' GRAMMAR FOR REVERSE POLISH EXPRESSIONS
- SAME HOLDS FOR POLISH EXPRESSIONS
- LOOK AT A MORE STANDARD INFIX EXPRESSION GRAMMAR

3

SIMPLE INFIX NOTATION EXPRESSIONS (SINE)

- TYPICAL EXPRESSIONS

```
15 + 9 * 7
21 * (7 – 3) / 2
a = b + c * 15 – d;
```


4

WHAT MIGHT THE GRAMMAR LOOK LIKE? PASS 0

S => E Etail
E => n Etail
Etail => op E | ∈

Notes:
Very simplistic. All operators equal precedence.
No parenthesis
Already non-left-recursive

5

SIMPLE PSEUDO-CODE IMPLEMENTATION

```
start() {
  tok = nextTok()
  if (tok.type == $$) {
    // epsilon
    exit
  } else if (tok.type == 'n') {
    expr()
    exprTail()
  }
}
expr() {
  match(n)   // reads next token in tok
  exprTail()
}
exprTail() {
   if (tok.type == op) {
     match(op)   // reads next token in tok
     expr()
   } else {
     // (epsilon)
     return
   }
}
```


6

WHAT WOULD A GRAMMAR LOOK LIKE? PASS 1

Pass 1:

S => E
E => number
E => identifier
E => E + E
E => E – E
E => E * E
E => E / E

Just add separate productions for each operator
Added identifiers

So: What are the drawbacks?

7

GRAMMAR PASS 1

Problems:
1. No precedence
2. Missing assignment statement syntax
3. Left recursive. ← This prevents writing a recursive decent parser

So let's not worry about assignment statement
Focus on items 1 and 3

8

GRAMMAR PASS 1

1. No precedence

So to create 'precedence' in a grammar, need multiple different 'non-terminals'

E => expression
T => term - part of additive sub expressions
F => factor. - part of multiplicative sub expressions

9

GRAMMAR PASS 2

S => E
E => E + T
E => E – T
E => T
T => T * F // These expansions make * and / bind tighter than + and -
T => T / F //
T => F
F => id
F => number
F => (E) // This allows us to prioritize additive expressions above multiplicative
// ones. I.e. a * (b + c)

10

GRAMMAR PASS 3

Next: Remove Left recursion
This occurs because the first non-terminal in a production matches the non-terminal being expanded.
Have to rewrite the grammar.

S => E
E => T Etail
Etail => + T Etail | - T Etail | ∈
T => F Ttail
Ttail => * F Ttail | / F Ttail | ∈
F => id | number | (E)
F => number
F => (E)

11

GRAMMAR PASS 3

Next: Remove Left recursion

S => E | ∈
E => T Etail
Etail => + T Etail | - T Etail | ∈
T => F Ttail
Ttail => * F Ttail | / F Ttail | ∈
F => id | number | (E)
F => number
F => (E)

Notes:

For all productions, there is no non-terminal on the left that matches the non-terminal being expanded

Since every productions starts with a unique terminal or a non-terminal that leads to a unique terminal, we can 'predict' which of several productions we need to use

12

IMPLEMENTATION

Lexical Analyzer recognizes:

- id (1 letter variable name)
- num (1 digit number)
- op (+, -, *, /, =)
- lp ('(')
- rp (')')

Returns a tuple (<type>, <char or value>)

13

IMPLEMENTATION

```
import lex
import sys
global currentTok
currentTok = ()

# S => E | e
# E => T Etail
# Etail => + T Etail | - T Etail | e
# T => F Ttail
# Ttail => * F Ttail | / F Ttail | e
# F => id
# F => number
# F => (E)

def parse(fileName):
  global currentTok

  lex.initLex(fileName)
  currentTok = lex.nextTok()
  res = start()
  if res:
    print("Parse succeeded!")
```

```
def start():
  global currentTok

  if currentTok[0] == '$$':
    return True
  return expr()

def expr():
  global currentTok

  if currentTok[0] in ['id', 'num', 'lp']:
    print("E => T Etail")
    term()
    etail()
    return True
  else:
    print('Error. Bad token')
    return False

def term():
  global currentTok

  if currentTok[0] in ['id', 'num', 'lp']:
    print("T => F Ttail")
    factor()
    ttail()
  else:
    print('Error. Bad token')
    return False
```


14

IMPLEMENTATION

```
def etail():
  global currentTok

  if currentTok[1] in ['+', '-']:
    operator = currentTok[1]
    print("Etail => + T Etail | - T Etail")
    currentTok = lex.nextTok()
    term()
    etail()
  else:
    print("Etail => e")

def ttail():
  global currentTok
  if currentTok[1] in ['*', '/']:
    operator = currentTok[1]
    currentTok = lex.nextTok()
    print("Ttail => * F Ttail | / F Ttail")
    term()
    etail()
  else:
    print("Ttail => e")
```

```
def factor():
  global currentTok
  if currentTok[0] == 'id':
    # Symbol table check/add
    print("F => id")
    currentTok = lex.nextTok()
  elif currentTok[0] == 'num':
    print("F => num")
    currentTok = lex.nextTok()
  else:
    print("F => (E)")
    currentTok = lex.nextTok()
    expr()
    # Should be 'rp'
    if currentTok[0] == 'rp':
      currentTok = lex.nextTok()
    else:
      print("Error. Bad token")
      return False
  return True

if len(sys.argv) < 1:
  print('Usage: rdparser.py <filename>')
  sys.exit()

parse(sys.argv[1])
```


15

QUESTIONS

16