CS 345

Parsing, Lexical Analysis, and Tools

William Cook

Parsing techniques

- Top-Down
  - Begin with start symbol, derive parse tree
  - Match derived non-terminals with sentence
  - Use input to select from multiple options
- Bottom Up
  - Examine sentence, applying reductions that match
  - Keep reducing until start symbol is derived
  - Collects a set of tokens before deciding which production to use

Top-Down Parsing

- Recursive Descent
  - Interpret productions as functions, nonterminals as calls
  - Must *predict* which production will match
    - looks ahead at a few tokens to make choice
  - Handles EBNF naturally
  - Has trouble with *left-recursive, ambiguous* grammars
    - left recursion is production of form E ::= E …
- Also called LL(k)
  - *scan* input **L**eft to right
  - use **L**eft edge to *select productions*
  - use **k** symbols of look-ahead for *prediction*

Recursive Descent LL(1) Example

- Example

  E ::= E + E | E – E | T   note: left recursion

  T ::= N | (E)

  N ::= { 0 | 1 | … | 9 }   { … } means *repeated*

- Problems:
  - Can’t tell at beginning whether to use E + E or E - E
    - would require arbitrary look-ahead
    - But it doesn’t matter because they both begin with T
  - Left recursion in E will never terminate…

Recursive Descent LL(1) Example

- Example

  E ::= T [+ E | – E]   […] means *optional*

  T ::= N | (E)

  N ::= { 0 | 1 | … | 9 }

- Solution
  - Combine equivalent forms in original production:

    E ::= E + E | E – E | T
  - There are algorithms for reorganizing grammars
    - cf. Greibach normal form (out of scope of this course)

LL Parsing Example

E•23+7

T•23+7

N•23+7

23•+7

23+•7

23+E•7

23+T•7

23+N•7

23+7•

E ::= T [+ E | – E]

T ::= N | (E)

N ::= { 0 | 1 | … | 9 }

• = Current location

Preduction

indent = function call

Intuition: Growing the parse tree from root down towards terminals.

Recursive Descent LL(1) Psuedocode

```
procedure E()              // E ::= T  [ + E  |  – E  ]
  a = T();
  if next token is "+" then b = E(); return add(a, b)
  if next token is "-"  then b = E(); return subtract(a, b)
  else return a

procedure T()    //   T ::= N  |  ( E )
  if next token is "(" then
    a = E();  check next token is ")";  return a;
  else      return N();

procedure N()     // N ::= { 0 | 1 | ... | 9 }
    while next token is digit do...
```

Bottom-Up Parsing

- Shift-Reduce
  - Examine sentence, applying reductions that match
  - Keep reducing until start symbol is derived
- Technique
  - Analyze grammar for all possible reductions
  - Create a large parsing table (never done by hand)
- Also called LR(k)
  - *scan* input **L**eft to right
  - use **R**ight edge to *select productions*
  - usually only **k=1** symbols of *look-ahead* needed

LR Parsing Example

•23+7

2•3+7

D•3+7

N•3+7

N3•+7

ND•+7

N•+7

T•+7

E•+7

…

E+•7

E+7•

E+D•

E+N•

E+T•

E+E•

E

E ::= E + E | E – E | T

T ::= N | (E)

N ::= N D | D

D ::= 0 | 1 | … | 9

• = Current location

Shift step

Reduce step

Intuition: Growing the parse tree from terminals up towards root.

Conficts

- Problem
  - Sometimes multiple actions apply
    - Shift another token / Reduce by rule R
    - Reduce by rule A / Reduce by rule B
  - Flagged as a *conflict* when parsing table is built
- Resolving conflicts
  - Rewrite the grammar
  - Use a default strategy
    - Shift-reduce: Prefer shifting
    - Reduce-reduce: Use first rule in written grammar
  - Use a token-dependent strategy
    - There's a nice way to do this

Confict Example

E*E•+	→ E*E+• (shift)	
	→ E•+ (reduce)	
E+E•+	→ E+E+• (shift)	
	→ E•+ (reduce)	

What does each resolution direction do?

Where have we seen this problem before?

Directives

- Precedence
  - Establish a token order: * binds tighter than +
    - Doesn't need to be given for all tokens
    - If unordered tokens conflict, use default strategy
- Associativity
  - *Left-associative:* favor reduce
  - *Right-associative*: favor shift
  - *Non-associative:* raise error
    - Flags “inherently confusing” expressions
    - Consider: a – b – c

Parser Generators

- Parser Generators
  - Input is a form of BNF grammar
    - Include "actions" to be performed as rules are recognized
  - Output is a parser
- Examples
  - ANTLR, JavaCC
    - generate recursive descent parsers
  - Yacc (many versions: CUP for Java)
    - generates bottom-up (shift-reduce) parsers

ANTLR Example

```
grammar Exp;

add  returns [double value]
    :    m1=prim                  {$value =  $m1.value;}
         ( '+' m2=prim               {$value += $m2.value;}
         | '-' m2=prim            {$value -= $m2.value;}
         )*;
prim returns [double value]
    :    n=Number        {$value = Double.parseDouble($n.text);}
    |    '(' e=add ')' {$value = $e.value;}
     ;
Number    :    ('0'..'9')+ ('.' ('0'..'9')+)?  ;
WS           :   (' ' | '\t' | '\r'| '\n') {$channel=HIDDEN;} ;
```

ANTLR Example creating AST

```
grammar Exp;

add  returns [Exp value]
   :   m1=prim           {$value =  $m1.value;}
       ( '+' m2=prim)*  {$value = new Add($value, $m2.value);}
    ;
prim returns [Exp value]
   :   n=Number       {double x = Double.parseDouble($n.text);
                        $value = new Num(x);}
   |   '(' e=add ')'      {$value = $e.value;}
    ;
Number    :    ('0'..'9')+ ('.' ('0'..'9')+)?  ;
WS           :   (' ' | '\t' | '\r'| '\n') {$channel=HIDDEN;} ;
```

Simplified AST without closures

```
interface Exp { int interp(); }
class Num implements Exp {
    int n;
    public Num(int n) { this.n = n; }
    public int interp() { return n; }
}
class Add implements Exp {
    Exp l, r;
    public Add (Exp l, r) { this.l = l; this.r = r; }
    public int interp() { return l.interp() + r.interp(); }
}
```