

Lecture 9: General and Bottom-Up Parsing

A Little Notation

Here and in lectures to follow, we'll often have to refer to general productions or derivations. In these, we'll use various alphabets to mean various things:

- Capital roman letters are nonterminals (A, B, \dots).
- Lower-case roman letters are terminals (or tokens, characters, etc.)
- Lower-case greek letters are sequences of zero or more terminal and nonterminal symbols, such as appear in sentential forms or on the right sides of productions (α, β, \dots).
- Subscripts on lower-case greek letters indicate individual symbols within them, so $\alpha = \alpha_1 \alpha_2 \dots \alpha_n$ and each α_i is a single terminal or nonterminal.

For example,

- $A : \alpha$ might describe the production $e : e \rightarrow e '+' t$,
- $B \Rightarrow \alpha A \gamma \Rightarrow \alpha \beta \gamma$ might describe the derivation steps $e \Rightarrow e '+' t \Rightarrow e '+' ID$ (α is $e '+'$; A is t ; B is e ; and γ is empty.)

Fixing Recursive Descent

- First, let's define an impractical but simple implementation of a top-down parsing routine.
- For nonterminal A and string $S=c_1c_2\dots c_n$, we'll define $\text{parse}(A, S)$ to return the length of a valid substring derivable from A .
- That is, $\text{parse}(A, c_1c_2\dots c_n) = k$, where

$$\underbrace{c_1c_2\dots c_k}_{A \xRightarrow{*}} c_{k+1}c_{k+2}\dots c_n$$

Abstract body of parse(A,S)

- Can formulate top-down parsing analogously to NFAs.

```
parse (A, S):
```

```
    """Assuming A is a nonterminal and S =  $c_1c_2\dots c_n$  is a string, return integer  $k$  such that A can derive the prefix string  $c_1\dots c_k$  of S."""
```

```
    Choose production 'A:  $\alpha_1\alpha_2\dots\alpha_m$ ' for A (nondeterministically)
```

```
    k = 0
```

```
    for x in  $\alpha_1, \alpha_2, \dots, \alpha_m$ :
```

```
        if x is a terminal:
```

```
            if x ==  $c_{k+1}$ :
```

```
                k += 1
```

```
            else:
```

```
                GIVE UP
```

```
        else:
```

```
            k += parse (x,  $c_{k+1}\dots c_n$ )
```

```
    return k
```

- Assume that the grammar contains one production for the start symbol: $p: \gamma \rightarrow$.
- We'll say that a call to parse returns a value if *some* set of choices for productions (the blue step) would return a value (just like NFA).
- Then if parse(p, S) returns a value, S must be in the language.

Example

Consider parsing $S = \text{"ID*ID-"}$ with a grammar from last time:

```
p : e '-'  
e : t  
  | e '/' t  
  | e '*' t  
t : ID
```

Example

Consider parsing $S = \text{"ID*ID+}"$ with a grammar from last time:

```
p : e '+'
e : t
  | e '/' t
  | e '*' t
t : ID
```

A failing path through the program:

```
parse(p, S):
  Choose p : e '+':
    parse(e, S):
      Choose e : t:
        parse(t, S):
          choose t : ID:
            check S[1] == ID; OK, so  $k_3 += 1$ ;
            return 1 (=  $k_3$ ; added to  $k_2$ )
          return 1 (and add to  $k_1$ )
        Check S[2] == S[ $k_1+1$ ] == '+': GIVE UP (S[2] == '*')
```

k_i means "the variable k in the call to parse that is nested i deep." Outermost k is k_1 .

Example

Consider parsing $S = \text{"ID*ID-|"}$ with a grammar from last time:

```
p : e '−'|
e : t
  | e '/' t
  | e '*' t
t : ID
```

A successful path through the program:

```
parse(p, S):
  Choose p : e '−':
    parse(e, S):
      Choose e : e '*' t:
        parse(e, S):
          choose e : t:
            parse(t, S):
              choose t : ID:
                check S[1] == ID; OK, so return 1
              return 1 (so k2 += 1)
            check S[k2] == '*'; OK, k2 += 1
          parse(t, S3): # S3 == "ID −|"
            choose t : ID:
              check S3[k3+1] == S3[1] == ID; OK
              k3 += 1; return 1 (so k2 += 1)
            return 3
          Check S[k1+1] == S[4] == '−': OK
          k1 += 1; return 4
```

k_i means "the variable k in the call to parse that is nested i deep." Outermost k is k_1 . Likewise for S .

Making a Deterministic Algorithm

- If we had an infinite supply of processors, could just spawn new ones at each "Choose" line.
- Some would give up, some loop forever, but on correct programs, at least one processor would get through.
- To do this for real (say with one processor), need to keep track of all possibilities systematically.
- This is the idea behind Earley's algorithm:
 - Handles any context-free grammar.
 - Finds all parses of any string.
 - Can recognize or reject strings in $O(N^3)$ time for ambiguous grammars, $O(N^2)$ time for "nondeterministic grammars", or $O(N)$ time for deterministic grammars (such as accepted by Bison).

Earley's Algorithm: I

- First, reformulate to use recursion instead of looping. Assume the string $S = c_1 \cdots c_n$ is fixed.
- Redefine **parse**:

parse (A: $\alpha \bullet \beta$, s, k):

""Assumes A: $\alpha\beta$ is a production in the grammar,
 $0 \leq s \leq k \leq n$, and α can produce the string $c_{s+1} \cdots c_k$.
Returns integer j such that β can produce $c_{k+1} \cdots c_j$.""

- Or diagrammatically, **parse** returns an integer **j** such that:

$$c_1 \cdots c_s \underbrace{c_{s+1} \cdots c_k}_{\alpha \xRightarrow{*}} \underbrace{c_{k+1} \cdots c_j}_{\beta \xRightarrow{*}} c_{j+1} \cdots c_n$$

Earley's Algorithm: II

parse (A: $\alpha \bullet \beta$, s, k):

```
    """Assumes A:  $\alpha\beta$  is a production in the grammar,  
    0 <= s <= k <= n, and  $\alpha$  can produce the string  $c_{s+1}\cdots c_k$ .  
    Returns integer j such that  $\beta$  can produce  $c_{k+1}\cdots c_j$ ."""
```

```
    if  $\beta$  is empty:
```

```
        return k
```

```
    Assume  $\beta$  has the form  $x\delta$ 
```

```
    if  $x$  is a terminal:
```

```
        if  $x == c_{k+1}$ :
```

```
            return parse(A:  $\alpha x \bullet \delta$ , s, k+1)
```

```
        else:
```

```
            GIVE UP
```

```
    else:
```

```
        Choose production ' $x: \kappa$ ' for  $x$  (nondeterministically)
```

```
        j = parse(x:  $\bullet\kappa$ , k, k)
```

```
        return parse (A:  $\alpha x \bullet \delta$ , s, j)
```

- Now do all possible choices that result in such a way as to avoid redundant work ("nondeterministic memoization").

Chart Parsing

- Idea is to build up a table (known as a *chart*) of all calls to parse that have been made.
- Only one entry in chart for each distinct triple of arguments ($A: \alpha \bullet \beta, s, k$).
- We'll organize table in columns numbered by the k parameter, so that column k represents all calls that are looking at c_{k+1} in the input.
- Each column contains entries with the other two parameters: $[A: \alpha \bullet \beta, s]$, which are called *items*.
- The columns, therefore, are *item sets*.

Example

Grammar

$p : e \text{ '}' \dashv \text{'}$
 $e : s \ I \mid e \text{ '}' + \text{' } e$
 $s : \text{'}' - \text{' } \mid$

Input String

$- \ I \ + \ I \ \dashv$

Chart. Headings are values of k and c_{k+1} (raised symbols).

	0	-	1	I	2	+	3	I
a. p:	$\bullet e \text{ '}' \dashv \text{'}, 0$							
b. e:	$\bullet e \text{ '}' + \text{' } e, 0$							
c. e:	$\bullet s \ I, 0$							
d. s:	$\bullet \text{'}' - \text{'}, 0$							
e. s:		$\bullet \text{'}' - \text{'}, 0$						
f. e:		$s \bullet I, 0$						
g. e:			$s \ I \bullet, 0$					
h. e:				$e \bullet \text{'}' + \text{' } e, 0$				
i. e:					$e \text{ '}' + \text{' } \bullet e, 0$			
j. e:						$\bullet s \ I, 3$		
k. s:						$\bullet, 3$		
l. e:							$s \bullet I, 3$	
	4	\dashv	5					
m. e:	$s \ I \bullet, 3$							
n. e:	$e \text{ '}' + \text{' } e \bullet, 0$							
o. p:	$e \bullet \text{'}' \dashv \text{'}, 0$							
p. p:					$e \text{ '}' \dashv \text{' } \bullet, 0$			

Example, completed

- Last slide showed only those items that survive and get used. Algorithm actually computes dead ends as well (unlettered, in red).

0	-	1	I	2	+	3	I
a.p: ● e '−', 0	e.s: '−'●, 0	g.e: s I●, 0	i.e: e '+' ● e, 0				
b.e: ● e '+' e, 0	f.e: s● I, 0	h.e: e ● '+' e, 0	j.e: ● s I, 3				
c.e: ● s I, 0		p: e ● '−', 0	k.s: ●, 3				
d.s: ● '−', 0			l.e: s ● I, 3				
s: ●, 0			s: ● '−', 3				
e: s ● I, 0			e: ● e '+' e, 3				
4	+	5					
m.e: s I●, 3		p.p: e '−' ●, 0					
n.e: e '+' e●, 0							
o.p: e● '−', 0							
e: e ● '+' e, 3							

Adding Semantic Actions

- Pretty much like recursive descent. The call $\text{parse}(A: \alpha \bullet \beta, s, k)$ can return, in addition to j , the semantic value of the A that matches characters $c_{s+1} \cdots c_j$.
- This value is actually computed during calls of the form $\text{parse}(A: \alpha' \bullet, s, k)$ (i.e., where the β part is empty).
- Assume that we have attached these values to the nonterminals in α , so that they are available when computing the value for A .

Ambiguity

- Ambiguity only important here when computing semantic actions.
- Rather than being satisfied with a single path through the chart, we look at *all* paths.
- And we attach the *set* of possible results of $\text{parse}(Y: \bullet\kappa, s, k)$ to the nonterminal Y in the algorithm.