Lecture 7: General and Bottom-Up Parsing

Parsing So Far

- We have seen that recursive-descent parsing it a simple and straightforward way to convert a grammar to a program that parses source using that grammar.
- However, because one has to predict which production to take without having seen the source tokens to be produced, it needs workarounds, as we've seen.
- In particular, must eliminate *left-recursion* and perform *left factoring* to make sure that branches are unique.
- So let's see what happens when we put off the decision about what production to use until after we've examined the text to be produced.
- This entails processing the children of a node in the parse tree *before* deciding on the production for that node; we determine the parse tree *from the bottom up.*

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,. . .)*.
- 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 ($\alpha, \beta, \ldots$).
- Subscripts on lower-case greek letters indicate individual symbols within them, so $\alpha = \alpha_1\alpha_n \ldots \alpha_n$ and each α_i is a single terminal or nonterminal.

So $A ::= \alpha$ might describe the production `e ::= e '+' t`,

. . . and $B \Rightarrow \alpha A\gamma \Rightarrow \alpha\beta\gamma$ 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 parse(A, S) to return the length of a valid prefix of S derivable from A.
- That is, parse(A, $c_1c_2\dots c_n$) = k, where

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

Abstract body of parse(A,S)

- Can formulate top-down parsing analogously to NFAs.

```
parse (A, S):
   """Assuming A is a nonterminal and S = c₁c₂...cₙ is a string, return
      integer k such that A can derive the prefix string c₁...cₖ of S."""
   Choose production 'A: α₁α₂···αₘ' for A (nondeterministically)
   k = 0
   for x in α₁, α₂, ···, αₘ:
       if x is a terminal:
          if x == cₖ₊₁:
              k += 1
          else:
              GIVE UP
       else:
          k += parse (x, cₖ₊₁···cₙ)
   return k
```

- Let the start symbol be p with exactly one production: p ::= γ $\dashv$.
- We'll say that a call to parse returns a value if *some* set of choices for productions (the blue step) would not give up (just like NFA).
- Then if parse(p, S) returns a value, S must be in the language.

Example

Consider parsing S="ID*ID⊣" with a grammar from last time:

```
p ::= e '⊣'
e ::= t
    | e '/' t
    | e '*' t
t ::= ID
```

Example

Consider parsing S="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 k3 += 1;
                  return 1    (= k3; added to k2)
            return 1 (and add to k1)
      Check S[2] == S[k1+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="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, 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_i.

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 or CUP).

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: α • β, s, k):
   """Assumes A: αβ is a production in the grammar,
      0 <= s <= k <= n, and α can produce the string c_{s+1}···c_k.
      Returns integer j such that β can produce c_{k+1}···c_j."""
```

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

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

Earley's Algorithm: II

```
parse (A ::= α • β, s, k):
   """Assumes A ::= αβ is a production in the grammar,
      0 <= s <= k <= n, and α can produce the string c_{s+1} ··· c_k.
      Returns integer j such that β can produce c_{k+1} ··· c_j."""
   if β is empty:
      return k
   Assume β has the form xδ
   if x is a terminal:
      if x == c_{k+1}:
          return parse(A ::= αx • δ, s, k+1)
      else:
          GIVE UP
   else:
      Choose production 'x ::= κ' for x (nondeterministically)
      j = parse(x ::= •κ, k, k)
      return parse (A ::= αx • δ, s, j)
```

- Now do all possible choices that result in such a way as to avoid redundant work ("nondeterministic memoization").
- That is, if `parse` is called with the same three arguments as a previous call, just use the result(s) of the previous call.

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 ($\texttt{A} ::= \alpha \bullet \beta$, 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: [$\texttt{A} ::= \alpha \bullet \beta$, s], which are called *items*.
- The columns, therefore, are *item sets*.

Example

Grammar

```
p ::= e '⊣'
e ::= s I  |  e '+' e
s ::= '-' |
```

Input String

```
- I + I ⊣
```

Chart. Headings are values of k and c_{k+1} (raised symbols). Item labels (*a–f*) trace the "ancestry" of each item. (Have shortened '::=' to ':' for compactness.)

0 (−)	1 (I)	2 (+)	3 (I)
a. `p: •e '⊣', 0`	*d.* `s: '-'•, 0`	*c.* `e: s I•, 0`	*b.* `e: e '+' •e, 0`
b. `e: •e '+' e, 0`	*c.* `e: s•I, 0`	*b.* `e: e •'+' e, 0`	*e.* `e: •s I, 3`
c. `e: •s I, 0`			*f.* `s: •, 3`
d. `s: •'-', 0`			*e.* `e: s •I, 3`

4 (⊣)	5
e. `e: s I•, 3`	*a.* `p: e '⊣' •, 0`
b. `e: e '+' e•, 0`	
a. `p: e•'⊣', 0`	

Example, completed

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

0 −	1 I	2 +	3 I
a. p: •e '⊣', 0	*d.* s: '−'•, 0	*c.* e: s I•, 0	*b.* e: e '+' •e, 0
b. e: •e '+' e, 0	*c.* e: s•I, 0	*b.* e: e •'+' e, 0	*e.* e: •s I, 3
c. e: •s I, 0		*a.* p: e • '⊣', 0	*f.* s: •, 3
d. s: •'−', 0			*e.* e: s •I, 3
g. s: •, 0			*i.* s: • '−', 3
h. e: s • I, 0			*j.* e: • e '+' e, 3

4 ⊣	5
e. e: s I•, 3	*a.* p: e '⊣' •, 0
b. e: e '+' e•, 0	
a. p: e•'⊣', 0	
j. e: e • '+' e, 3	

Ambiguous Example

Grammar

```
p ::= e '⊣'
e ::= I  |  e '+' e
```

Input String

```
I + I + I⊣
```

Chart. Only useful items shown.

0 (I)	1 (+)	2 (I)	3 (+)
a. p: •e '⊣', 0	*c.* e: I •, 0	*b.* e: e '+'•e, 0	*d.* e: I •, 2
b. e: •e '+' e, 0	*b.* e: e •'+' e, 0	*d.* e: •I, 2	*b.* e: e '+' e •, 0
c. e: •I, 0		*e.* e: •e '+' e, 2	*e.* e: e •'+' e, 2
			b. e: e •'+' e, 0

4 (I)	5 (⊣)	6
b. e: e '+' •e, 0	*f.* e: I •, 4	*a.* p: e ⊣•, 0
e. e: e '+' •e, 2	*b.* e: e '+' e •, 0	
f. p: •I, 4	*e.* e: e '+' e •, 2	
	a. p: e •⊣, 0	

Adding Semantic Actions

- Using syntax-directed translation to get semantic values is pretty much like recursive descent.
- The call `parse(A:` $\alpha \bullet \beta$`, s, k)` can return, in addition to j, the semantic value of the `A` that matches symbols $c_{s+1} \cdots c_j$.
- The value is computed during calls of the form `parse(A:` $\alpha' \bullet$`, s, k)` (i.e., where the β part is empty). For terminal symbols, value is provided by the lexer.

Adding Semantic Actions (II)

- On a chart, when we see an item `A:` $\alpha\bullet, s$ in column k, it tells us to
  - Perform the semantic action corresponding to the production `A ::=` α, getting a semantic value v for the left-hand side `A`.
  - For each item `B:` $\beta \bullet A\gamma$, t in column s of the chart, when adding the item `B:` $\beta A \bullet \gamma$, t to column k, also attach value v to that instance of A in the new item.
  - For all items derived from `B:` $\beta \bullet A\gamma$, t as its dot is shifted, also attach v to the same instance of A.

This step is what provides the values of nonterminals needed to compute v values (in Bison notation: `$1`, `$2`, etc.; in CUP notation, labels such as `e1` and `e2` in the rule $e ::= e:e1\ '+'\ e:e2$).

Example with Semantic Values

Grammar

```
p : e:a '⊣'          {: RESULT = a; :}
e : t:b              {: RESULT = b; :}
e : e:a '+' t:b      {: RESULT = a + b; :}
t : I:a              {: RESULT = a; :}
t : t:a '*' I:b      {: RESULT = a * b; :}
```

Input String (I's are numerals).

1 + 3 * 2 ⊣

Chart. Only useful items shown. Semantic values are subscripts; red items show where they are computed.

0 (I_1)	1 (+)	2 (I_3)	3 (*)
a. p: •e '⊣', 0	*d.* t_1: I_1 •, 0	*b.* e: e_1 '+'•t, 0	*e.* t_3: I_3 •, 2
b. e: •e '+' t, 0	*c.* e_1: t_1 •, 0	*e.* t: •I, 2	*f.* t: t_3 •'*' I, 2
c. e: •t, 0	*b.* e: e_1 •'+' t, 0	*f.* t: •t '*' I, 2	
d. t: •I, 0			

4 (I_2)	5 (⊣)	6
f. t: t_3 '*' •I, 2	*f.* t_6: t_3 '*' I_2 •, 2	*a.* p_7: e_7 ⊣•, 0
	b. e_7: e_1 '+' t_6 •, 0	
	a. p: e_7 •⊣, 0	

Handling Ambiguity in Semantics (Sketch)

- Ambiguity really only important here when computing semantic actions.
- Rather than being satisfied with a single path through the chart, we look at *all* paths.
- The call `parse(A:` $\alpha \bullet \beta$`, s, k)` can return a *set* of semantic values.
- Accordingly, we attach sets of semantic values to nonterminals.