Lecture 6: General and Bottom-Up Parsing

Project #1 Notes

- Project involves generating an AST for Python dialect.
- Our tools provide extended BNF (BNF + regular-expression notations like '*', '+', and '?') both for context-free and lexical definitions.
- Tools also provide largely automatic AST building:
  - Tokens double as AST operators.
  - By default, each rule computes the list of all trees built by its right-hand side.
  - The '^' notation allows you to build a tree designating the operator.
  - Or, in an action, you can use '`$^(...)`' to build an AST node, and '`$*`' to denote the list of children's ASTs.
- We've also provided methods to print nodes.

Project #1 Notes (II)

- In my solution, a majority of grammar rules look like this:

```
attributeref: primary "."! identifier
              { $$ = $^(ATTRIBUTEREF, $*); }
    ;
```

  and all the printing, etc. is taken care of.

- Dummy tokens like ATTRIBUTEREF are first defined with

```
%token ATTRIBUTEREF "@attributeref"
```

- In a few cases, I can just write

```
expr1 : expr1 "or"^ expr1
```

  and the action is generated automatically.

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.

For example,

- $A : \alpha$ might describe the production `e:  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=$c_1 c_2 \ldots c_n$, we'll define parse(A, S) to return the length of a valid substring derivable from A.
- That is, parse(A, $c_1 c_2 \ldots c_n$) = k, where

$$\underbrace{c_1 c_2 \ldots c_k}_{A \stackrel{*}{\Longrightarrow}} c_{k+1} c_{k+2} \ldots c_n$$

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
```

- Assume that the grammar contains one production for the start symbol: `p:` $\gamma \dashv$.
- 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="`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, 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: 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").

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

Input String

```
- I + I ⊣
```

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

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			*k.* s: •, 3
d. s: •'-', 0			*l.* e: s •I, 3

4 ⊣	5
m. e: s I•, 3	*p.* p: e '⊣' •, 0
n. e: e '+' e•, 0	
o. p: e•'⊣', 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 `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 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 `parse(Y:` $\bullet\kappa$`, s, k)` to the nonterminal Y in the algorithm.