COMP 181

Lecture 7

More parsing

September 26, 2006

Prelude

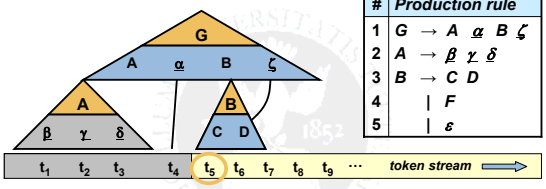
- What is this structure?
  Ryugyong Hotel, North Korea
- Facts
  - 105 floors, 1083 ft
  - 3000 rooms, 3.9 million sq. ft.
  - Started in 1987, halted 1992
  - DPRK: it doesn't exist

Where are we

- **Last time**: Top-down parsing
  - Non-termination – eliminating left recursion
  - Using FIRST and FOLLOW sets
  - The LL(1) property
  - Recursive descent parsers
- **Today**:
  - Building FIRST and FOLLOW sets
  - Generating top-down parsers
  - Start bottom-up parsing

Top-down parsing

- Build parse tree top down



#	Production rule
1	$G \rightarrow A\ \alpha\ B\ \zeta$
2	$A \rightarrow \beta\ \gamma\ \delta$
3	$B \rightarrow C\ D$
4	$\mid F$
5	$\mid \varepsilon$

Is "CD"? Consider all possible strings derivable from "CD" What is the set of tokens that can appear at start?

$t_5 \in$ FIRST(C D)
$t_5 \in$ FIRST(F)
} disjoint?
$t_5 \in$ FIRST(ε) = FOLLOW(B)

Left factoring

- **Problem**
  - What if my grammar is not LL(1)?
  - May be able to fix it, with transformations
- Example:

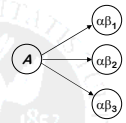
#	Production rule
1	$A \rightarrow \alpha\ \beta_1$
2	$\mid \alpha\ \beta_2$
3	$\mid \alpha\ \beta_3$

#	Production rule
1	$A \rightarrow \alpha\ Z$
2	$Z \rightarrow \beta_1$
3	$\mid \beta_2$
4	$\mid \beta_3$

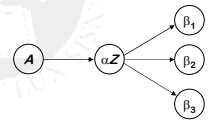
Left factoring

- Graphically

#	Production rule
1	$A \rightarrow \alpha\ \beta_1$
2	$\quad\ \mid \alpha\ \beta_2$
3	$\quad\ \mid \alpha\ \beta_3$



#	Production rule
1	$A \rightarrow \alpha\ Z$
2	$Z \rightarrow \beta_1$
3	$\quad \mid \beta_2$
	$\quad \mid \beta_3$



Expression example

#	Production rule
1	*factor* → identifier
2	| identifier [*expr*]
3	| identifier (*expr*)

First+(1) = {identifier}
First+(2) = {identifier}
First+(3) = {identifier}

After left factoring:

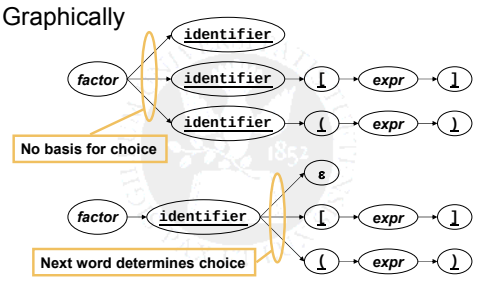
#	Production rule
1	*factor* → identifier *post*
2	*post* → [*expr*]
3	| (*expr*)
4	| ε

First+(1) = {identifier}
First+(2) = { [}
First+(3) = { (}
First+(4) = ?
= Follow(*post*)
= {operators}

➡ **In this form, it has LL(1) property**

Left factoring

- Graphically



Left factoring

- *Question*

  Using left factoring and left recursion elimination, can we turn an arbitrary CFG to a form where it meets the LL(1) condition?

- *Answer*

  Given a CFG that does not meet LL(1) condition, it is ***undecidable*** whether or not an LL(1) grammar exists

- *Example*

  $\{a^n\, 0\, b^n \mid n \geq 1\} \cup \{a^n\, 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

Limits of LL(1)

- No LL(1) grammar for this language:

  $\{a^n\, 0\, b^n \mid n \geq 1\} \cup \{a^n\, 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

#	Production rule
1	$G \rightarrow \underline{a}\ A\ \underline{b}$
2	$\quad \mid a\ B\ \underline{bb}$
3	$A \rightarrow \underline{a}\ A\ \underline{b}$
4	$\quad \mid \underline{0}$
5	$B \rightarrow \underline{a}\ B\ \underline{bb}$
6	$\quad \mid \underline{1}$

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group

Recursive descent parsing

- Massage grammar to have LL(1) condition
  - Remove left recursion
  - Left factor, where possible
- Build FIRST (and FOLLOW) sets
- Define a procedure for each non-terminal
  - Implement a case for each right-hand side
  - Call procedures as needed for non-terminals
  - Add extra code, as needed
- Can we automate this process?

FIRST and FOLLOW sets

The right-hand side of a production

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) *iff* $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentence.

FOLLOW(G) = {EOF}, where G is the start symbol

Computing FIRST sets

- ***Idea***:
  Use FIRST sets of the right side of production

  $$A \rightarrow B_1\ B_2\ B_3 \ldots$$

- Cases:
  - FIRST(A→B) = FIRST(B_1)
    - What does FIRST(B_1) mean?
    - Union of FIRST($B_1 \rightarrow \gamma$) for all γ
  - What if ε in FIRST(B_1)?
    - ⇒ FIRST(A→B) += FIRST(B_2) *repeat as needed*
  - What if ε in FIRST(B_i) for all i?
    - ⇒ FIRST(A→B) += {ε} *leave {ε} for later*

Why += ?

Algorithm

- For one production: p = A → β

```
if (β is a terminal t)
        FIRST(p) = {t}
else if (β == ε)
        FIRST(p) = {ε}
else
        Given  β = B1 B2 B3 … Bk
        i = 0
        do {    i = i + 1;
                FIRST(p) += FIRST(Bi)  -  {ε}
        } while ( ε  in FIRST(Bi)  &&  i < k)

        if (ε  in FIRST(Bi) && i == k)      FIRST(p) += {ε}
```

Why do we need to remove ε from FIRST(B_i)?

Algorithm

- For one production:
  - Given **A** → B_1 B_2 B_3 B_4 B_5
  - Compute FIRST(**A**→**B**) using FIRST(**B**)
  - How do we get FIRST(**B**)?
- What kind of algorithm does this suggest?
  - Recursive? Like a depth-first search of the productions
- **Problem**:
  - What about recursion in the grammar?
  - **A** → **B** and **B** → **A**

Algorithm

- **Solution**
  - Start with FIRST(B) empty
  - Compute FIRST(A) using empty FIRST(B)
  - Now go back and compute FIRST(B)
    - What if it's no longer empty?
    - Then we recompute FIRST(A)
    - What if new FIRST(A) is different from old FIRST(A)?
    - Then we recompute FIRST(B) again…
- When do we stop?
  - When no more changes occur – called ***convergence***
  - FIRST(A) and FIRST(B) both satisfy equations
- This is another ***fixpoint*** algorithm

Algorithm

- Using fixpoints:

```
forall p    FIRST(p) = {}

while (FIRST sets are changing)
        pick a random p
        compute FIRST(p)
```

- Can we be smarter?
  - Yes, visit in special order
  - Reverse post-order depth first search
    Visit all children (all right-hand sides) before visiting the left-hand side, whenever possible

Example

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier

FIRST(3) = { + }
FIRST(4) = { - }
FIRST(5) = { ε }

FIRST(7) = { * }
FIRST(8) = { / }
FIRST(9) = { ε }

FIRST(1) = ?

FIRST(1) = FIRST(2)
= FIRST(6)
= FIRST(10) ∪ FIRST(11)
= { number, identifier }

Computing FOLLOW sets

- ***Idea***:
  Push FOLLOW sets down, use FIRST where needed
- Cases:

$$A \rightarrow B_1 \; B_2 \; B_3 \; B_4 \; \ldots B_k$$

  - What is FOLLOW(B_i)?
    - FOLLOW(B_i) = FIRST(B_{i+1})
  - What about FOLLOW(B_k)?
    - FOLLOW(B_k) = FOLLOW(A)
  - What if ε ∈ FOLLOW(B_k)?
    ⇒ FOLLOW(B_{k-1}) ∪= FOLLOW(A) *extends to k-2, etc.*

Example

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier

FOLLOW(*goal*) = { EOF }

FOLLOW(*expr*) = FOLLOW(*goal*) = { EOF }

FOLLOW(*expr2*) = FOLLOW(*expr*) = { EOF }

FOLLOW(*term*) = ?

FOLLOW(*term*) += FIRST(*expr2*)
+= { +, -, ε }
+= { +, -, FOLLOW(*expr*)}
+= { +, -, EOF }

Example

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier

FOLLOW(*term2*) += FOLLOW(*term*)

FOLLOW(*factor*) = ?

FOLLOW(*factor*) += FIRST(*term2*)
+= { *, / , ε }
+= { *, / , FOLLOW(*term*)}
+= { *, / , +, -, EOF }

Computing FOLLOW Sets

FOLLOW(G) ← {EOF}
for each A ∈ NT, FOLLOW(A) ← Ø

while (FOLLOW sets are still changing)
  for each p ∈ P, of the form $A \rightarrow \beta_1 \beta_2 \ldots \beta_k$
    $FOLLOW(\beta_k) \leftarrow FOLLOW(\beta_k) \cup FOLLOW(A)$
    TRAILER ← FOLLOW(A)
    for i ← k down to 2
      if $\varepsilon \in FIRST(\beta_i)$ then
        $FOLLOW(\beta_{i-1}) \leftarrow FOLLOW(\beta_{i-1}) \cup \{FIRST(\beta_i) - \{\varepsilon\}\} \cup TRAILER$
      else
        $FOLLOW(\beta_{i-1}) \leftarrow FOLLOW(\beta_{i-1}) \cup FIRST(\beta_i)$
        TRAILER ← Ø

Generating a top-down parser

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier

- Two pieces:
  - Select the right RHS
  - Satisfy each part
- First piece:
  - FIRST+() for each rule
  - Mapping:
    NT × Σ → *rule#*
    Look familiar?

Generating a top-down parser

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → `number`
11	| `identifier`

- Second piece
  - Keep track of progress
  - Like a depth-first search
  - Use a stack
- **Idea**:
  - Push *Goal* on stack
  - Pop stack:
    - Match terminal symbol, *or*
    - Apply NT mapping, push RHS on stack

Table-driven approach

- Encode mapping in a table
  - Row for each non-terminal
  - Column for each terminal symbol

  Table[NT, symbol] = rule#

  if symbol ∈ FIRST+(NT -> rhs(#))

	+,-	*, /	id, num
expr2	***term expr2***	error	error
term2	error	***factor term2***	error
factor	error	error	***(do nothing)***

Code

```
push the start symbol, G, onto Stack
top ← top of Stack
loop forever
  if top = EOF and token = EOF then break & report success
  if top is a terminal then
     if top matches token then
        pop Stack                    // recognized top
        token ← next_token()
  else                               // top is a non-terminal
     if TABLE[top,token] is A→ B1B2…Bk then
        pop Stack                    // get rid of A
        push Bk, Bk-1, …, B1         // in that order
  top ← top of Stack
```

Missing else's for error conditions

Parsing

- Where are we?
  - Top-down parsers
  - LL(1) property
  - Automatic, table-driven parsers
- *Next*: bottom-up parsers
  - Why?
  - More powerful
  - Widely used – yacc, bison, JavaCUP

Next time

- Bottom up parsing

Left factoring

- Algorithm:

  ∀ *A* ∈ *NT*,
  find the longest prefix α *that occurs in two or more right-hand sides of A*
  if $\alpha \neq \varepsilon$ *then replace all of the A productions,*
  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$,
  with
  $A \rightarrow \alpha Z \mid \gamma$
  $Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
  where Z is a new element of NT

  Repeat until no common prefixes remain