Lecture 18 - Nov. 17

Syntactic Analysis

Extended FIRST Set Computation
FOLLOW Set, START Set, Left Factoring
TDP: Terminating & Min. Backtracking

Announcements

- **Assignment 3** released
- **Project Milestone 2** submission due at **11:59pm on Tuesday, Nov. 22**
- **Project Report Template** to be walked over on Tuesday's class

Extended First Set

Q. How about FIRST(Expr' → Term' Factor)?

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

	Expr	Expr'	Term	Term'	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

$$\text{FIRST}(\beta_1\beta_2\ldots\beta_n) = \left\{ \text{FIRST}(\beta_1) \cup \text{FIRST}(\beta_2) \cup \ldots \text{FIRST}(\beta_k) \;\middle|\; \forall i : 1 \le i < k \bullet \epsilon \in \text{FIRST}(\beta_i) \wedge \epsilon \notin \text{FIRST}(\beta_k) \right\}$$

$A \to \beta_1 \beta_2 \ldots \beta_{k-1} \mid \beta_k \ldots \beta_n$

nullable / not nullable

$\epsilon \notin \text{FIRST}(\text{Term})$

$\text{FIRST}(\text{Term Expr}') = \text{FIRST}(\text{Term}) = \{(, n, n\}$

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term'*	→	x *Factor Term'*
1	*Expr*	→	*Term Expr'*	7		|	÷ *Factor Term'*
2	*Expr'*	→	+ *Term Expr'*	8		|	ϵ
3		|	- *Term Expr'*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term'*	11		|	name

first component that's not nullable ⇒ no need to collect FIRST further.

Is the FIRST Set Sufficient?

Expr′	→	+	*Term* *Term′*	(1)
	|	−	*Term* *Term′*	(2)
	|	ϵ		(3)

useful if we can know what symbols follows Expr

FIRST(+ Term Term′) = {+}

FIRST(- Term Term′) = {-}

FIRST(epsilon) = {ε}

Top-Down Parsing: Discovering Leftmost Derivations (2)

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack

backtrack ≜ pop focus.siblings; focus := focus.parent; focus.resetChildren
```

Parse: a + a * a

```
Expr   → Term Expr′
Expr′  → + Term Expr′
       | ϵ
Term   → Factor Term′
Term′  → * Factor Term′
       | ϵ
Factor → (Expr)
       | a
```

Rest: exercise

word: "a" "+"

doesn't match "+" but ε has no effect on concat ⇒ the next symbol may match

FOLLOW Set

terminals only

start variable

$$\text{FOLLOW}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$$

derived from v

a string that follows the derivation of v

Assumption: FIRST is already computed.

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

	Expr	Expr′	Term	Term′	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

FOLLOW(Expr) ∩ FIRST(Expr′) contains ϵ

	Expr	Expr′	Term	Term′	Factor
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

FOLLOW(Expr)

FOLLOW Set: Algorithm

$$\textsc{Follow}(v) = \{w \mid w, x, y \in \Sigma^* \wedge v \overset{*}{\Rightarrow} x \wedge S \overset{*}{\Rightarrow} xwy\}$$

```
ALGORITHM: GetFollow
   INPUT: CFG G = (V, Σ, R, S)
   OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})        (map)
PROCEDURE:
   for A ∈ V: FOLLOW(A) := ∅
   FOLLOW(S) := {eof}                       (start variable)
   lastFollow := ∅
   while (lastFollow ≠ FOLLOW):
      lastFollow := FOLLOW
      for A → β1 β2 ... βk ∈ R:             (when considering this rule, Follow may need to be updated for ...)
         trailer := FOLLOW(A)
         for i: k .. 1                      (→ go from right to left entries)
            if βi ∈ V then
               FOLLOW(βi) := FOLLOW(βi) ∪ trailer
               if ε ∈ FIRST(βi)             (is βi nullable?)
                  then trailer := trailer ∪ (FIRST(βi) − ε)   (if βi is nullable, accumulate its FIRST for the Follow of βi-1)
                  else trailer := FIRST(βi)                   (if βi is not nullable, then don't include its FIRST.)
            else
               trailer := FIRST(βi)
```

$$A \longrightarrow \beta_1\ \beta_2\ \cdots\ \beta_{k-1}\ \beta_k$$

Follow?

FOLLOW(β_k) = ? Follow(A)

When $\varepsilon \in$ FIRST(β_k)

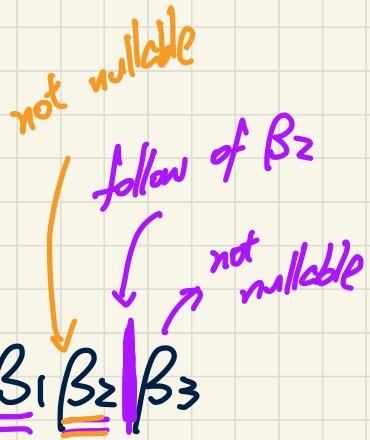
FOLLOW(β_{k-1}) = ?

When $\varepsilon \notin$ FIRST(β_k)

FOLLOW(β_{k-1}) = ?

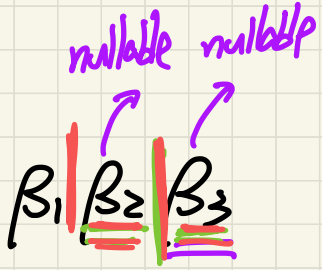
Computing the FOLLOW Sets: Trailers

$A \rightarrow \beta_1\beta_2\beta_3$



Case 1: $\varepsilon \notin$ FIRST(β_3), $\varepsilon \notin$ FIRST(β_2)

+ FOLLOW(β_3) = Follow(A)
+ FOLLOW(β_2) = FIRST(β_3) ∪ ~~Follow(β_3)~~ ?
+ FOLLOW(β_1) = FIRST(β_2).



Case 2: $\varepsilon \in$ FIRST(β_3), $\varepsilon \in$ FIRST(β_2)

+ FOLLOW(β_3) = Follow(A)
+ FOLLOW(β_2) = FIRST(β_3) ∪ Follow(β_3)
+ FOLLOW(β_1) = FIRST(β_2) ∪ FIRST(β_3) ∪ Follow(β_3)

trailer

Right-Recursive CFG:

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

G, F, E, T, T′

```
ALGORITHM: GetFollow
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: FOLLOW : V ⟶ ℙ(T ∪ {eof})
PROCEDURE:
  for A ∈ V: FOLLOW(A) := ∅
  FOLLOW(S) := {eof}
  lastFollow := ∅
  while (lastFollow ≠ FOLLOW):
    lastFollow := FOLLOW
    for A → β1 β2 ... βk ∈ R:
      trailer := FOLLOW(A)
      for i: k .. 1:
        if βi ∈ V then
          FOLLOW(βi) := FOLLOW(βi) ∪ trailer
          if ϵ ∈ FIRST(βi)
            then trailer := trailer ∪ (FIRST(βi) − ϵ)
            else trailer := FIRST(βi)
        else
          trailer := FIRST(βi)
```

FOLLOW Set: Tracing

First choose rules whose LHS is processed.
Then rules whose RHS ends with a terminal.

	Expr	Expr′	Term	Term′	Factor
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

Goal	Expr	Expr′	Term	Term′	Factor
eof	eof,)	eof,)	+, -, eof	)	*, +, :

↳ FOLLOW(Expr′) if RHS nullable

Backtrack-Free Grammar

A ***backtrack-free grammar*** has each of its productions $A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ satisfying:

$$\forall i, j : 1 \le i, j \le n \wedge i \neq j \bullet \textbf{START}(\gamma_i) \cap \textbf{START}(\gamma_j) = \varnothing$$

between any two productions rules, we can always unambiguously choose one.

$$\textbf{START}(A \to \beta) = \begin{cases} \textbf{FIRST}(\beta) & \text{if } \epsilon \notin \textbf{FIRST}(\beta) \\ \textbf{FIRST}(\beta) \cup \textbf{FOLLOW}(A) & \text{otherwise} \end{cases}$$

β is not nullable

β is nullable

$\textbf{FIRST}(\beta)$ is the extended version where β may be $\beta_1 \beta_2 \ldots \beta_n$

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

Top-Down Parsing: Algorithm with lookahead

```
ALGORITHM: TDParse
  INPUT: CFG G = (V, Σ, R, S)
  OUTPUT: Root of a Parse Tree or Syntax Error
PROCEDURE:
  root := a new node for the start symbol S
  focus := root
  initialize an empty stack trace
  trace.push(null)
  word := NextWord()
  while (true):
    if focus ∈ V then
      if ∃ unvisited rule focus → β1 β2 ... βn ∈ R ∧ word ∈ START(β) then
        create β1, β2 ... βn as children of focus
        trace.push(βn βn−1 ... β2)
        focus := β1
      else
        if focus = S then report syntax error
        else backtrack
    elseif word matches focus then
      word := NextWord()
      focus := trace.pop()
    elseif word = EOF ∧ focus = null then return root
    else backtrack
```

always choose the prod. w/ the whole start symbol matches

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name