



# COMP 181

---



Lecture 7  
*Table-driven Parsing*

October 2, 2005






## 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


2

3

## Last time

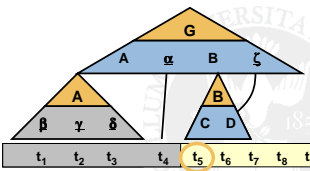
- Intro to parsing
- Top-down parsing
  - Non-termination – eliminating left recursion
  - Left factoring
  - FIRST and FOLLOW sets – the LL(1) property
  - Recursive descent parsers
- Project stage 1



4

## 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	$\mid \epsilon$

$t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \ t_8 \ t_9 \ \dots$  token stream  $\rightarrow$

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

$t_5 \in \text{FIRST}(C D)$   
 $t_5 \in \text{FIRST}(F)$   
 $t_5 \in \text{FIRST}(\epsilon) = \text{FOLLOW}(B)$


} disjoint?



5

## Today

- Building FIRST and FOLLOW sets
- Generating top-down parsers
- Start bottom-up parsing



6

## FIRST and FOLLOW sets

### FIRST( $\alpha$ )

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

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

### FOLLOW( $\alpha$ )

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

$\text{FOLLOW}(S) = \{\text{EOF}\}$ , where  $S$  is the start symbol



## Computing FIRST sets

### Idea:

Use FIRST sets of the right side of production

### Cases:

$$A \rightarrow B_1 B_2 B_3 B_4 B_5$$

- $\text{FIRST}(A) \cup = \text{FIRST}(B_1)$
- What if  $\epsilon$  in  $\text{FIRST}(B_1)$ ?  
 $\Rightarrow \text{FIRST}(A) \cup = \text{FIRST}(B_2)$  *repeat as needed*
- What if  $\epsilon$  in  $\text{FIRST}(B_i)$  for all  $i$ ?  
 $\Rightarrow \text{FIRST}(A) \cup = \{\epsilon\}$  *leave  $\{\epsilon\}$  for later*



## Computing FIRST Sets

```

for each  $x \in T$ ,  $\text{FIRST}(x) \leftarrow \{x\}$ 
for each  $A \in NT$ ,  $\text{FIRST}(A) \leftarrow \emptyset$ 
while (FIRST sets are still changing)
  for each  $p \in P$ , of the form  $A \rightarrow \beta$ ,
    if  $\beta$  is  $\epsilon$  then
       $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \{\epsilon\}$ 
    else if  $\beta$  is  $B_1 B_2 \dots B_k$ , then begin
       $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup (\text{FIRST}(B_1) - \{\epsilon\})$ 
      for  $i \leftarrow 1$  to  $k-1$  by 1 while  $\epsilon \in \text{FIRST}(B_i)$ 
         $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup (\text{FIRST}(B_{i+1}) - \{\epsilon\})$ 
      if  $i = k-1$  and  $\epsilon \in \text{FIRST}(B_k)$ 
        then  $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \{\epsilon\}$ 
    end
  
```

Fixpoint computation



NOTE:  $\epsilon \in \text{FIRST}(B)$  means  $B$  could be empty

## Example

#	Production rule	
1	goal $\rightarrow$ expr	$\text{FIRST}(1) = \text{FIRST}(2)$
2	expr $\rightarrow$ term expr2	$= \text{FIRST}(6)$
3	expr2 $\rightarrow$ + term expr2	$= \text{FIRST}(10) \cup \text{FIRST}(11)$
4	- term expr2	$= \{\text{number, identifier}\}$
5	$\epsilon$	$\text{FIRST}(3) = \{\epsilon\}$
6	term $\rightarrow$ factor term2	$\text{FIRST}(4) = \{-\}$
7	term2 $\rightarrow$ * factor term2	$\text{FIRST}(5) = \{\epsilon\}$
8	/ factor term2	$\text{FIRST}(7) = \{*\}$
9	$\epsilon$	$\text{FIRST}(8) = \{/ \}$
10	factor $\rightarrow$ number	$\text{FIRST}(9) = \{\epsilon\}$
11	identifier	



## Computing FOLLOW sets

### Idea:

Push FOLLOW sets down, use FIRST where needed

### Cases:

$$A \rightarrow B_1 B_2 B_3 B_4 B_5$$

- $\text{FOLLOW}(B_k) = \text{FOLLOW}(A)$
- $\text{FOLLOW}(B_i) \cup = \text{FIRST}(B_{i+1})$
- What if  $\epsilon \in \text{FOLLOW}(B_i)$ ?  
 $\Rightarrow \text{FOLLOW}(B_{k-1}) \cup = \text{FOLLOW}(A)$  *extends to  $k-2$ , etc.*



## Computing FOLLOW Sets

```

 $\text{FOLLOW}(S) \leftarrow \{\text{EOF}\}$ 
for each  $A \in NT$ ,  $\text{FOLLOW}(A) \leftarrow \emptyset$ 
while (FOLLOW sets are still changing)
  for each  $p \in P$ , of the form  $A \rightarrow \beta_1 \beta_2 \dots \beta_k$ 
     $\text{FOLLOW}(\beta_k) \leftarrow \text{FOLLOW}(\beta_k) \cup \text{FOLLOW}(A)$ 
    TRAILER  $\leftarrow \text{FOLLOW}(A)$ 
    for  $i \leftarrow k$  down to 2
      if  $\epsilon \in \text{FIRST}(\beta_i)$  then
         $\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1}) \cup (\text{FIRST}(\beta_i) - \{\epsilon\})$ 
         $\cup$  TRAILER
      else
         $\text{FOLLOW}(\beta_{i-1}) \leftarrow \text{FOLLOW}(\beta_{i-1}) \cup \text{FIRST}(\beta_i)$ 
    TRAILER  $\leftarrow \emptyset$ 
  
```



## 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 → <u>number</u>
11	<u>identifier</u>

FOLLOW(goal) = { EOF }  
 FOLLOW(expr) = { EOF }  
 FOLLOW(expr2) = { EOF }  
 FOLLOW(term) U = FIRST(expr2)  
                   U = { +, -, ε }  
 FOLLOW(term2) U = FOLLOW(term)  
                   U = { +, -, EOF }  
 FOLLOW(factor) U = FIRST(term2)  
                   U = { \*, /, +, -, EOF }  
                   U = { \*, /, EOF }



## 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 → <u>number</u>
11	<u>identifier</u>

- 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 → <u>number</u>
11	<u>identifier</u>

- 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(#))
- Code...



## Code

```

push the start symbol, S, 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
      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



## Bottom-up parsing

- Assemble tree bottom up

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

- "Frontier" is at the top of the subtrees
- Builds rightmost derivation

Tufts University Computer Science 19

## Bottom-up Parsing

Sentential form: terminals and non-terminals

A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$$

← bottom-up

- To replace  $\gamma_i$  with  $\gamma_{i-1}$  match some *rhs*  $\beta$  against  $\gamma_i$  then replace  $\beta$  with its corresponding *lhs*, A. (assuming the production  $A \rightarrow \beta$ )
- Replacement of  $\beta$  with A is called a **reduction**

Tufts University Computer Science 20

## Finding Reductions

Consider the simple grammar

1	Goal	$\rightarrow$	$a A B e$
2	A	$\rightarrow$	$A b c$
3		$\mid$	$b$
4	B	$\rightarrow$	$d$

Sentential Form	Next Reduction Prod'n	Pos'n
abcde	3	2
aAbcde	2	4
aAde	4	3
aABe	1	4
Goal	—	—

And the input string abcde

The trick is scanning the input and finding the next reduction  
The mechanism for doing this must be efficient

Tufts University Computer Science 21

## Finding Reductions (Handles)

- Problem:
  - Find a substring of frontier that matches the right-hand side of a production
  - Informally, we call this substring  $\beta$  a **handle**
- Formally,
  - A **handle** of a right-sentential form  $\gamma$  is a pair  $\langle A \rightarrow \beta, k \rangle$
  - $A \rightarrow \beta \in P$  and  $k$  is the position in  $\gamma$  of  $\beta$ 's rightmost sym.
  - If  $\langle A \rightarrow \beta, k \rangle$  is a handle, then replacing  $\beta$  at  $k$  with A produces the right sentential form from which  $\gamma$  is derived in the rightmost derivation.

Tufts University Computer Science 22

## Handles

A **handle** of a right-sentential form  $\gamma$  is a pair  $\langle A \rightarrow \beta, k \rangle$  where  $A \rightarrow \beta \in P$  and  $k$  is the position in  $\gamma$  of  $\beta$ 's rightmost symbol.

$\gamma = Z X t_7 Y t_{10}$   
 $\beta = X d Y e$   
 $k = 5$

$A \rightarrow X d Y e$

Tufts University Computer Science 23

## Finding Reductions (Handles)

- Critical Insight (Theorem?)  
If G is unambiguous, then every right-sentential form has a **unique handle**.
- Sketch of Proof:
  - 1 G is unambiguous  $\Rightarrow$  rightmost derivation is unique
  - 2  $\Rightarrow$  a unique production  $A \rightarrow \beta$  applied to derive  $\gamma_i$  from  $\gamma_{i-1}$
  - 3  $\Rightarrow$  a unique position  $k$  at which  $A \rightarrow \beta$  is applied
  - 4  $\Rightarrow$  a unique handle  $\langle A \rightarrow \beta, k \rangle$

Tufts University Computer Science 24

## Example

#	Production rule	Prod'n	Sentential Form	Handle
1	goal $\rightarrow$ expr	—	Goal	—
2	expr $\rightarrow$ expr + term	1	Expr	1,1
3	expr - term	3	Expr - Term	3,3
4	term	5	Expr - Term * Factor	5,5
5	term $\rightarrow$ term * factor	9	Expr - Term * <id,y>	9,5
6	term / factor	7	Expr - Factor * <id,y>	7,3
7	factor	8	Expr - <num,z> * <id,y>	8,3
8	factor $\rightarrow$ number	4	Term - <num,z> * <id,y>	4,1
9	identifier	7	Factor - <num,z> * <id,y>	7,1
10	( expr )	9	<id,y> - <num,z> * <id,y>	9,1

The expression grammar

Handles for rightmost derivation of  $x - 2 * y$



## Handle pruning

- Discovering a handle & reducing it to the appropriate left-hand side is called **handle pruning**
- Handle pruning forms the basis for a bottom-up parsing method
- To construct a rightmost derivation  
 $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$
- Apply the following simple algorithm for  $i \leftarrow n$  to 1 by  $-1$   
 Find the handle  $\langle A_i \rightarrow \beta_i, k_i \rangle$  in  $\gamma_i$   
 Replace  $\beta_i$  with  $A_i$  to generate  $\gamma_{i-1}$



## Handle-pruning

- One implementation technique is the **shift-reduce parser**

```

push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)
  if the top of the stack is a handle  $A \rightarrow \beta$ 
    then // -- reduce  $\beta$  to  $A$ 
      pop  $|\beta|$  symbols off the stack
      push  $A$  onto the stack
    else if (token = EOF)
      then // -- shift
        push token
        token ← next_token()
    else // -- need to shift, but out of input
      report an error
    
```



## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	id = num * id	none	shift
\$ id	= num * id		

- Shift until the top of the stack is the right end of a handle
- Find the left end of the handle & reduce



## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	id = num * id	none	shift
\$ id	= num * id	9,1	red. 9
\$ Factor	= num * id	7,1	red. 7
\$ Term	= num * id	4,1	red. 4
\$ Expr	= num * id		

- Shift until the top of the stack is the right end of a handle
- Find the left end of the handle & reduce



## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	id = num * id	none	shift
\$ id	= num * id	9,1	red. 9
\$ Factor	= num * id	7,1	red. 7
\$ Term	= num * id	4,1	red. 4
\$ Expr	= num * id	none	shift
\$ Expr =	num = id	none	shift
\$ Expr = num	= id		

- Shift until the top of the stack is the right end of a handle
- Find the left end of the handle & reduce



## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	$id = num * id$	none	shift
\$ $id$	$= num * id$	9,1	red. 9
\$ $Factor$	$= num * id$	7,1	red. 7
\$ $Term$	$= num * id$	4,1	red. 4
\$ $Expr$	$= num * id$	none	shift
\$ $Expr =$	$num * id$	none	shift
\$ $Expr = num$	$= id$	8,3	red. 8
\$ $Expr = Factor$	$= id$	7,3	red. 7
\$ $Expr = Term$	$= id$		

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

Tufts University Computer Science 31

## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	$id = num * id$	none	shift
\$ $id$	$= num * id$	9,1	red. 9
\$ $Factor$	$= num * id$	7,1	red. 7
\$ $Term$	$= num * id$	4,1	red. 4
\$ $Expr$	$= num * id$	none	shift
\$ $Expr =$	$num * id$	none	shift
\$ $Expr = num$	$= id$	8,3	red. 8
\$ $Expr = Factor$	$= id$	7,3	red. 7
\$ $Expr = Term$	$= id$	none	shift
\$ $Expr = Term =$	$id$	none	shift
\$ $Expr = Term = id$			

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

Tufts University Computer Science 32

## Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	$id = num * id$	none	shift
\$ $id$	$= num * id$	9,1	red. 9
\$ $Factor$	$= num * id$	7,1	red. 7
\$ $Term$	$= num * id$	4,1	red. 4
\$ $Expr$	$= num * id$	none	shift
\$ $Expr =$	$num * id$	none	shift
\$ $Expr = num$	$= id$	8,3	red. 8
\$ $Expr = Factor$	$= id$	7,3	red. 7
\$ $Expr = Term$	$= id$	none	shift
\$ $Expr = Term =$	$id$	none	shift
\$ $Expr = Term = id$		9,5	red. 9
\$ $Expr = Term = Factor$		5,5	red. 5
\$ $Expr = Term$		3,3	red. 3
\$ $Expr$		1,1	red. 1
\$ $Goal$		none	accept

5 shifts +  
9 reduces +  
1 accept

1. Shift until the top of the stack is the right end of a handle  
2. Find the left end of the handle & reduce

Tufts University Computer Science 33

## Example

Stack	Input	Action
\$	$id = num * id$	shift
\$ $id$	$= num * id$	red. 9
\$ $Factor$	$= num * id$	red. 7
\$ $Term$	$= num * id$	red. 4
\$ $Expr$	$= num * id$	shift
\$ $Expr =$	$num * id$	shift
\$ $Expr = num$	$= id$	red. 8
\$ $Expr = Factor$	$= id$	red. 7
\$ $Expr = Term$	$= id$	shift
\$ $Expr = Term =$	$id$	shift
\$ $Expr = Term = id$		red. 9
\$ $Expr = Term = Factor$		red. 5
\$ $Expr = Term$		red. 3
\$ $Expr$		red. 1
\$ $Goal$		accept

Tufts University Computer Science 34

## Shift-reduce Parsing

- Shift reduce parsers are easily built and easily understood
- A shift-reduce parser has just four actions
  - Shift** — next word is shifted onto the stack
  - Reduce** — right end of handle is at top of stack  
Locate left end of handle within the stack  
Pop handle off stack & push appropriate *lhs*
  - Accept** — stop parsing & report success
  - Error** — call an error reporting/recovery routine
- Implementation
  - Accept & Error are simple
  - Shift is just a push and a call to the scanner
  - Reduce takes *rhs* pops & 1 push
  - If handle-finding requires state, put it in the stack  $\Rightarrow 2x$  work

Handle finding is key

- handle is on stack
- finite set of handles  $\Rightarrow$  use a DFA!

Tufts University Computer Science 35

## More about Handles

To be a handle, a substring of a sentential form  $\gamma$  must:

- match the right hand side  $\beta$  of some rule  $A \rightarrow \beta$
- be some rightmost derivation from the goal symbol that produces the sentential form  $\gamma$  with  $A \rightarrow \beta$  as the last production applied

Simply looking for right hand sides that match strings is not good enough

**Critical Question:** How can we know when we have found a handle without generating lots of different derivations?

- Answer:** we use look ahead in the grammar along with tables produced as the result of analyzing the grammar.
- LR(1) parsers build a DFA that runs over the stack & finds them

Tufts University Computer Science 36

## LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar

*Informal definition:*

A grammar is LR(1) if, given a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

We can

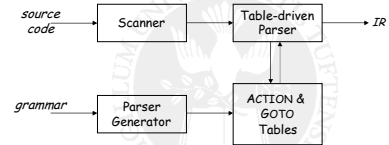
1. isolate the handle of each right-sentential form  $\gamma_i$ , and
2. determine the production by which to reduce,

by scanning  $\gamma_i$  from left-to-right, going at most 1 symbol beyond the right end of the handle of  $\gamma_i$



## LR(1) Parsers

- A table-driven LR(1) parser looks like



- Tables can be built by hand  
However, this is a perfect task to automate



## Next time...

- LR parsing tables
- Could help with programming project
- Done with parsing
- Homework?

