



# COMP 181

---



Lecture 8  
*Shift-reduce parsing*

Oct 5, 2005

## Prelude

- Starbucks question  
How much did Starbucks corporation spend last year on health care for employees?
- Answer:  
More than it spent on coffee

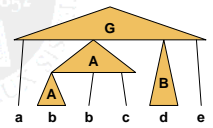

## Shift-reduce parsing

- Grammar:
- Is "abcde" in L(G)?
- Yes

Prove by "reverse" derivation:

#	Production rule
1	$G \rightarrow a A B e$
2	$A \rightarrow A b c$
3	$\mid b$
4	$B \rightarrow d$

Rule	Sentential form
-	abcde
3	aAbcde
2	aAde
4	aABe
1	G






## Choosing reductions

- Idea:
  - Search for right sides of productions, reduce
  - Does this work?
- Not always:
 



Rule	Sentential form
-	abcde
3	aAbcde
2	aAde
?	...now what?
- Problem:
 

*"aAde" is not part of any sentential form*

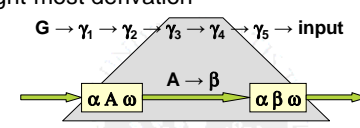
## Handles



- Informally
  - A substring of the derivation that matches a RHS
  - Replace it with the left-hand non-terminal
- Formally
  - Let  $\gamma$  be a right-sentential form
  - A handle of  $\gamma$  is:
    - a production  $A \rightarrow \beta$
    - a position in  $\gamma$  where  $\beta$  can be found
  - Such that
    - replacing  $\beta$  with  $A$  in  $\gamma$  produces the previous sentential form in the right-most derivation

## Why does this help?

- Right-most derivation
 

$G \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_5 \rightarrow \text{input}$ 

- $\omega$  contains only terminal symbols
- $A$  is the right-most non-terminal in  $\gamma_3$
- Unambiguous grammar
  - Right-most derivation is unique
  - At each step, handle is unique

## Example

Reduce	Rule	Sentential form	Handle
↓	-	abcde	$A \rightarrow b$ at pos 2
	3	aAbcde	$A \rightarrow Abc$ at pos 2
	2	aAde	$B \rightarrow d$ at pos 3
	4	aABe	$G \rightarrow aABe$ at pos 1

- Why *right*-most derivation?
  - Expands non-terminals from right to left
  - (Everything to right is a terminal)
- Reverse derivation
  - Reduces handles from left to right
  - This is order the scanner provides tokens



## Implementation

### First:

- Data structures
  - Stack: builds the right-sentential form
  - Input buffer – or scanner
- Engine
  - *Read tokens and*
  - **Shift** – push a token on the stack
  - **Reduce** – top N tokens form a handle, replace with appropriate non-terminal from left-hand side
  - **Accept** – goal on stack, all input consumed
  - **Error** – no handle, or no more input



## Code outline

```

push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
  if the top of the stack is a handle A→β
    then // -- reduce β to A
      pop |β| 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
\$ $id$	$= num * id$		



## 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
\$ $Expr$	$= num * id$		



## 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
\$ $Expr = num$	$= id$		



## 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		

Tufts University Computer Science 13

## 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			

Tufts University Computer Science 14

## 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

Tufts University Computer Science 15

## Key assumption

- What are we assuming?
  - The handle will eventually occur at top of the stack
- Intuition
  - When non-terminal B is on top of the stack, next handle
    - (a) includes B – and possibly some terminals on either side
    - (b) is to the right of B – consists of only terminals
- Why?
  - Otherwise, it would mean B was in-tact in the previous sentential form
  - The derivation did not expand right-most non-terminal

Tufts University Computer Science 16

## Problems

- Shift-reduce parsing doesn't always work
- Dangling `else`
  - Shift-reduce conflict
- Array access vs function call
  - Reduce-reduce conflict

#	Production rule
1	$stmt \rightarrow \text{if } expr \text{ then } stmt$
2	$\quad \quad \quad   \text{if } expr \text{ then } stmt \text{ else } stmt$
3	$\quad \quad \quad   \dots \text{other statements} \dots$

#	Production rule
1	$stmt \rightarrow \text{id } ( \text{parameter-list } )$
2	$\quad \quad \quad   \text{expr}$
3	$\text{expr} \rightarrow \text{id } ( \text{expr } )$
	$\quad \quad \quad   \text{id}$
	$\text{parameter-list} \rightarrow \text{parameter-list } , \text{id}$
	$\quad \quad \quad   \text{id}$

Tufts University Computer Science 17

## Fixing problems

- Shift-reduce conflicts
  - Information on stack cannot resolve
  - Add productions to make choice explicit
- Reduce-reduce conflicts
  - Often cannot be handled in parser
  - Typically: make scanner more sophisticated
  - Example: make "proc name" a different token
  - Example: add *explicit states* to scanner

Tufts University Computer Science 18

## Explicit states

- User-defined states in scanner
- Regular expressions only match in state
- Actions can transition to states

```

%{
%}
state START
state COMMENT
%%
START /*      { BEGIN COMMENT; }
COMMENT */   { BEGIN START; }
COMMENT .    { }
START "for"  { return TOKEN_FOR; }
    
```



## LR parsers

*Particular technique for finding handles*

- Handle most language constructs
- Most general non-backtracking method known
- Can be implemented very efficiently
- Proper superset of predictive parsers
- Detects syntax errors as soon as it is possible in left-to-right scan of input

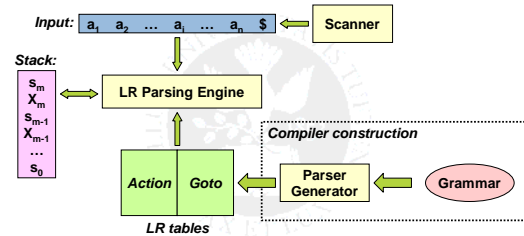


## Implementation

- Cannot be built by hand
- General idea:
  - Construct a DFA to recognize handles
  - Encode information in a table
  - Use a stack to keep track of progress
- Different table constructions
  - SLR, Canonical LR, LALR
  - Different levels of power, generality



## Engine



## Algorithm components

- Stack
  - String of the form :  $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$
  - $X_i$ : grammar symbol
  - $s_i$ : "state" summarizes the frontier of parse tree
- Two tables:
  - **action** – whether to shift, reduce, accept, error
  - **goto** – next state



## Table

- Action
  - Given state and input symbol,  $action[s_p, a] =$ 
    - **Shift**  $s'$ , where  $s'$  is a state
    - **Reduce** by a grammar production  $A \rightarrow \beta$
    - **Accept**
    - **Error**
- Goto
  - Given a state and a grammar symbol,  $goto[s_p, X] =$ 
    - Produce next state (after recognizing an X)



## Algorithm


```

push s0 on stack
token = scanner.next_token()
repeat
  s = top of stack
  if action[s, token] = reduce A → β then
    pop |β| pairs (Xi, sm) off the stack
    s' = top of stack
    push A on stack
    push goto[s', A] on stack
  else if action[s, token] = shift s' then
    push token on stack
    push s on stack
    token = scanner.next_token()
  else if action[s, token] = accept then
    return true
  else error()

```

Top of stack is handle  
A → β

- Work
  - Shift each token
  - Pop each token
- Errors
  - Input exhausted
  - Error entry in table




Tufts University Computer Science

25

## Key idea

- Handle can be identified by looking at stack
  - X<sub>i</sub> symbols form right-sentential form
  - Handles are patterns of symbols
  - ➔ We can construct a DFA to recognize them
- Avoid scanning the stack
  - State s<sub>i</sub> represents handle DFA state after reading the X<sub>i</sub> symbols from bottom to top
  - The goto function encodes this DFA




Tufts University Computer Science

26

## Building tables

- Conceptually
  - Enumerate all **viable prefixes** of right derivations
  - Build DFA to recognize them
    - Actually, construct an NFA
    - Apply subset construction
- In practice
  - Cannot enumerate all viable prefixes
  - Build DFA directly from grammar




Tufts University Computer Science

27

## LR items

- an LR(k) **item** is a pair [p, l] where
  - p is a production with a •  
"Dot" represents the top of the stack
  - l is a lookahead string of length k
  - Generate an item for each possible position of the dot, and each lookahead
- Example: A → αβγ generates four items
  - [A → •αβδ, l]
  - [A → α•βχ, l]
  - [A → αβ•γ, l]
  - [A → αβγ•, l]




Tufts University Computer Science

28

## LR items

### Interpreting items

- [A → •βγ, a]
  - Input so far is consistent with a use of A → βγ, but parser has not recognized any part of β or γ.
- [A → β •γ, a]
  - Input so far is consistent with a use of A → βγ, and parser has recognized an instance of β.
- [A → βγ •, a]
  - Parser has seen βγ and the lookahead symbol a is consistent with a reduction using A → βγ.




Tufts University Computer Science

29

## Algorithm overview

- Build **canonical collection** of LR items
  - Collect sets of equivalent items using **closure**
  - Compute **goto** function for transitions
  - Each set becomes a state
  - Determine parsing actions for each state
    - "Shift" if the dot is in the middle somewhere
    - "Reduce" if the dot is at the end



Tufts University Computer Science

30

## Closure function

For a set of items  $I$ , construct  $closure(I)$  as

- Add all items in  $I$  to  $closure(I)$
- If  $[A \rightarrow \alpha \bullet B \beta]$  is in  $closure(I)$  and  $B \rightarrow \gamma$  is a production, then add  $[B \rightarrow \bullet \gamma]$  to  $closure(I)$
- Continue until fixpoint

### Intuition:

- At  $A \rightarrow \alpha \bullet B \beta$  we expect to see  $B \beta$  next
- Since  $B \rightarrow \gamma$  then we could also see a  $\gamma$



## Example

- Compute  $closure([G \rightarrow \bullet E])$

- Rule 2 – add  $[E \rightarrow \bullet E + T]$
- Rule 3 – add  $[E \rightarrow \bullet T]$
- Rule 4 – add  $[T \rightarrow \bullet T * F]$
- Rule 5 – add  $[T \rightarrow \bullet F]$
- Rule 6 – add  $[F \rightarrow \bullet (E)]$
- Rule 7 – add  $[F \rightarrow \bullet id]$

#	Production rule
1	$G \rightarrow E$
2	$E \rightarrow E + T$
3	$E \rightarrow T$
4	$T \rightarrow T * F$
5	$T \rightarrow F$
6	$F \rightarrow (E)$
7	$F \rightarrow id$

- All the possible right-sentential forms at the start of parsing



## Goto operation

Given set of items  $I$ , and symbol  $X$

- $goto(I, X)$  is

All items  $[A \rightarrow \alpha X \bullet \beta]$  such that  $[A \rightarrow \alpha \bullet X \beta]$  is in  $I$

### Intuition:

If the parser was in state  $[A \rightarrow \alpha \bullet X \beta]$  and then recognized an instance of  $X$ , then the new state is  $[A \rightarrow \alpha X \bullet \beta]$



## Example

- Given  $I = \{ [G \rightarrow E \bullet], [E \rightarrow E + T] \}$

- Compute  $goto(I, +)$

- Start with  $[E \rightarrow E + T]$
- Compute closure
  - Rule 4 – add  $[T \rightarrow \bullet T * F]$
  - Rule 5 – add  $[T \rightarrow \bullet F]$
  - Rule 6 – add  $[F \rightarrow \bullet (E)]$
  - Rule 7 – add  $[F \rightarrow \bullet id]$

#	Production rule
1	$G \rightarrow E$
2	$E \rightarrow E + T$
3	$E \rightarrow T$
4	$T \rightarrow T * F$
5	$T \rightarrow F$
6	$F \rightarrow (E)$
7	$F \rightarrow id$



## Canonical collection of sets

- Algorithm

- Initially
  - $C = \{closure([G \rightarrow \bullet X])\}$
- Repeat
  - For each set  $I$  in  $C$ , and each symbol  $X$
  - If  $goto(I, X)$  is not empty, add it to  $C$
- Until nothing new added to  $C$

- Result

- Each set represents equivalent parsing states
- Collection results from exploring all transitions  $X$



## Adding lookahead

Modify  $closure(I)$  as

- Add all items in  $I$  to  $closure(I)$
- If  $[A \rightarrow \alpha \bullet B \beta, \underline{a}]$  is in  $closure(I)$  and  $B \rightarrow \gamma$  is a production then
  - For all  $\underline{b}$  in  $FIRST(\gamma \underline{a})$   *$\gamma$  could be empty*
  - Add  $[B \rightarrow \bullet \gamma, \underline{b}]$  to  $closure(I)$

- Idea:

The  $FIRST(\gamma)$  lookahead will help distinguish between different productions of  $B$



## LR tables (finally)

- Given  $C = \{I_1, I_2, \dots, I_n\}$
- State  $s_i$  is constructed from  $I_i$ 
  - If  $[A \rightarrow \alpha \underline{a} \beta]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  
Set  $\text{action}[i, a] = \text{"shift } j\text{"}$
  - If  $[A \rightarrow \alpha \bullet]$  is in  $I_i$  then  
Set  $\text{action}[i, a] = \text{"reduce } \alpha \text{ to } A\text{"}$  for all  $a$  in  $\text{FOLLOW}(A)$
  - If  $[G \rightarrow X \bullet]$  is in  $I_i$  then  
Set  $\text{action}[i, \$] = \text{"accept"}$
  - All other entries = "error"
- If  $\text{goto}(i, A) = j$  then  $\text{goto}[i, A] = j$



## Optimizing tables

Three options:

- Combine terminals such as  $\pm$  and  $-$ ,  $*$  and  $/$ 
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup
  - Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
  - Both LALR(1) and SLR(1) produce smaller tables
  - Implementations are readily available



## Next time...

- Beyond syntax
  - Types
  - Semantic analysis
  - Attribute grammars

