Syntax Analysis
[Chapter 4]

Lecture 9

Edited by Dr. Ismail Hababeh
German Jordanian University
Adapted from slides by Dr. Robert A. Engelen

Top-Down Table-Driven Predictive LL(1) Parsing – (leftmost derivation) Example

$E \rightarrow T\ E_R$
$E_R \rightarrow +\ T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$T_R \rightarrow *\ F\ T_R \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \mathbf{id}$

Stack	Input	Production applied
$\$E$	**id+id*id$**	
$\$E_R T$	**id+id*id$**	$E \rightarrow T\ E_R$
$\$E_R T_R F$	**id+id*id$**	$T \rightarrow F\ T_R$
$\$E_R T_R \mathbf{id}$	**id+id*id$**	$F \rightarrow \mathbf{id}$
$\$E_R T_R$	**+id*id$**	
$\$E_R$	**+id*id$**	$T_R \rightarrow \varepsilon$
$\$E_R T+$	**+id*id$**	$E_R \rightarrow +\ T\ E_R$
$\$E_R T$	**id*id$**	
$\$E_R T_R F$	**id*id$**	$T \rightarrow F\ T_R$
$\$E_R T_R \mathbf{id}$	**id*id$**	$F \rightarrow \mathbf{id}$
$\$E_R T_R$	***id$**	$T_R \rightarrow \varepsilon$
$\$E_R T_R F*$	***id$**	$T_R \rightarrow *\ F\ T_R$
$\$E_R T_R F$	**id$**	
$\$E_R T_R \mathbf{id}$	**id$**	$F \rightarrow \mathbf{id}$
$\$E_R T_R$	**$**	$T_R \rightarrow \varepsilon$
$\$E_R$	**$**	$E_R \rightarrow \varepsilon$
$\$$	**$**	

Bottom-Up Parsing

- Construct a parse tree from the leaves to the root
- Shift-Reduce Parsing.
- LR (Left-to-right, Rightmost derivation):
  - SLR(1) Simple LR with 1 token of lookahead
  - Canonical LR or LR(1) parser is an LR(k) parser for k=1, i.e. with a single lookahead terminal.
  - LALR Look Ahead LR with k tokens of lookahead

Shift-Reduce Parsing

- Copy the process of “reducing” an input string to the start symbol of the grammar.
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

Shift-Reduce Parsing

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

Reducing input string:

a <u>b</u> b c d e

a <u>*A* b c</u> d e

a *A* <u>d</u> e

<u>a *A* *B* e</u>

S

Strings that match Grammar productions

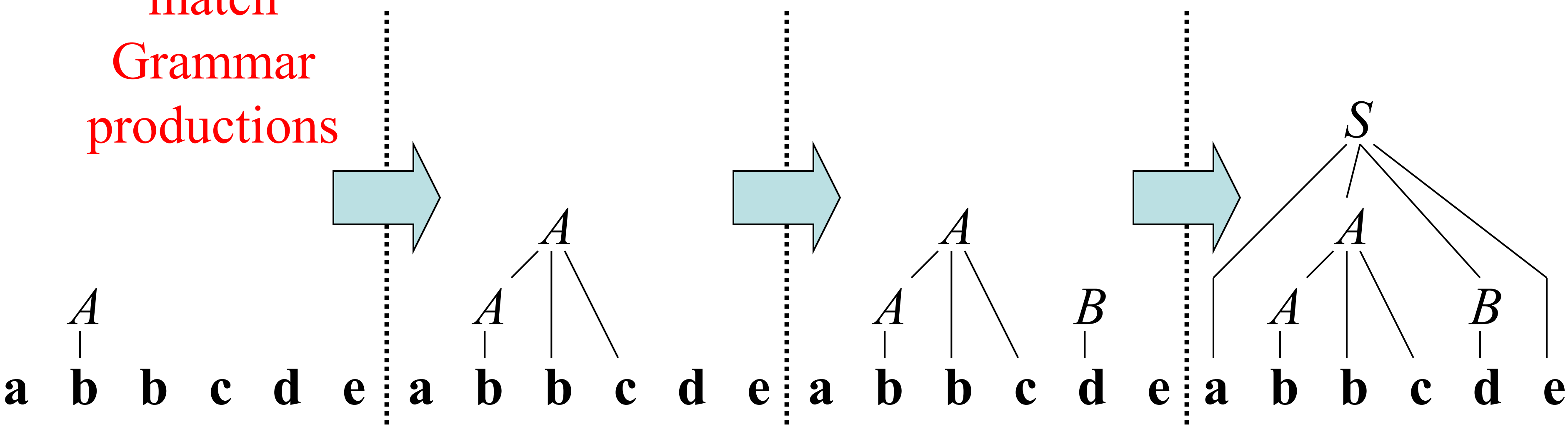
Shift-reduce corresponds to a rightmost derivation:

$S \Rightarrow_{rm} \mathbf{a}\ A\ B\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\Rightarrow_{rm} \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$



Handles

A *handle* is a substring of grammar symbols in a *right-sentential form* that matches a right-hand side of a production

Grammar:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ \underline{A\ \mathbf{b}\ \mathbf{c}}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{d}}\ \mathbf{e}$

$\underline{\mathbf{a}\ A\ B\ \mathbf{e}}$

S

Handles are used to reduce terminals by Non-terminals

$\mathbf{a}\ \underline{\mathbf{b}}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ \underline{\mathbf{b}}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\mathbf{a}\ A\ A\ \mathbf{e}$

… ?

NOT a handle, because further reductions will fail (result is not a sentential form)

The substring to the right of the handle must contain only terminals

Bottom-Up Parsing - Conflicts

- Conflicts happen when the Context Free Grammar has an *inadequate state*.
- Two possible actions, don't know what to put in parse: Shift-Reduce, or Reduce-Reduce. (Shift-Shift is not action).
- Reasons of Conflicts:
  - Ambiguity
    - Two or more possible parse trees for a string
    - Determining general grammar ambiguity is undecidable.

Bottom-Up Parsing Actions
Shift-Reduce Parsing Example

Grammar:

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow \textbf{id}$

Stack	Input	Action
$	**id+id*id$**	shift
$<u>id</u>	**+id*id$**	reduce $E \rightarrow \textbf{id}$
**$**$E$	**+id*id$**	shift
$**$E$+**	**id*id$**	shift
$**$E$+<u>id</u>**	***id$**	reduce $E \rightarrow \textbf{id}$
**$**$E$**+**$E$	***id$**	shift (or reduce?)
$**$E$**+**$E$***	**id$**	shift
$**$E$**+**$E$*<u>id</u>**	**$**	reduce $E \rightarrow \textbf{id}$
**$**$E$**+**$\underline{E * E}$	**$**	reduce $E \rightarrow E * E$
**$**$\underline{E + E}$	**$**	reduce $E \rightarrow E + E$
**$**$E$	**$**	accept

Find handles to reduce

How to resolve conflicts?

We choose shift because in the lookahead we have * but not $

Scan the input left to right, and the parser shifts 0 or more input symbols to the stack until it is ready to reduce the string of grammar symbols on the top of the stack … repeat until we reach the start symbol.

Shift-Reduce Conflicts

- Shift-Reduce: we cannot decide whether to shift a symbol or reduce the top of stack.
- Grammars used in compliers are usually lookahead LR(1).
- Conflicts might be caused by the fact that we read one symbol of lookahead (LR(1)).

Shift-Reduce Conflicts

- Shift-Reduce and Reduce-Reduce conflicts are caused by:
  - Ambiguity of the grammar
  - The limitations of the LR parsing method (even when the grammar is unambiguous).

Shift-Reduce Conflicts - Example

Ambiguous grammar:

$S \rightarrow$ **if** E **then** S
| **if** E **then** S **else** S
| **other**

Stack	Input	Action
$…	…**$**	…
$…**if** E **then** S	**else**…**$**	shift or reduce?

Resolve in favor of shift, so **else** matches closest **if**

Shift-Reduce Conflicts

- An LR grammar can never be ambiguous.
- In the example, we cannot tell whether if expr then stmt is the handle.
- Here we have a shift/reduce conflict.
- It depends on what follows else in the input:
  - it might be correct to reduce if expr then stmt to stmt

  or
  - it might be correct to shift else and then look for another stmt to complete the alternative if expr then stmt else stmt

- We can adapt the grammar by favoring the shifting

  the parser will associate each else with the previous unmatched then (the parser will behave correctly as we expect).

Reduce-Reduce Conflicts

- Reduce-Reduce: we don't know which reduction to take.
- We have a handle but the stack content and the next input symbol are insufficient to determine which production should be used in a reduction.
- Suppose the lexical analyzer returns token **id** for all names (functions, arrays, variables, ...)
- A procedure call or array reference would appear as **id (id, id)**

Reduce-Reduce Conflicts - Example

Grammar:

stmt → **id** (*parameter_list*)
  | *expr* = *expr*

parameter_list → *parameter_list*, *parameter*
  | *parameter*

parameter → **id**

expr → **id** (*expr_list*)
  | **id**

expr_list → *expr_list*, *expr*
  | *expr*

Reduce-Reduce Conflicts - Example

After pushing the first three tokens of **id(id**, **id)** into the stack:

The lexical analyzer should be modified to return **procid** token for procedure names

Stack	Input	Action
… id (id	**, id) …**	reduce *parameter* → **id** If we have a procedure or *expr* → **id** If we have an array reference

We know we need to reduce id on the top of the stack:

parameter → id *if we have a procedure*

expr → id *if we have an array reference*

Reduce-Reduce Conflicts - Example

After reading the first three tokens of **procid(id**, **id**) onto the stack:

Grammar:
stmt → **procid** (parameter_list)
.
…
…

Stack	Input	Action
… procid (id,	**id) …**	reduce *parameter* → **id**(parameter_list) Note: the 3rd symbol from the top of the stack determined the reduction