Lecture 8

Bottom up parsing

"**Shift-Reduce**" Parsing

Reduce a string to the start symbol of the grammar.

At every step a particular **sub-string is matched** (in left-to-right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Rightmost Derivation:

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

abbcde
aAbcde
aAde
aABe
S

Handles

Substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

a**<u>b</u>**bcde → a**<u>Abc</u>**de → aA**<u>d</u>**e → **<u>aABe</u>** → S

Shift Reduce Parsing with a Stack

Two problems

- locate a handle
- decide which production to use

Using stack

- **Shift --** input symbols into the stack until a handle is found on top of it.
- **Reduce --** the handle to the corresponding non-terminal.
- **Accept** -- when the input is consumed and only the start symbol is on the stack,
- **Error** -- when it does not lead to the start symbol

$E \rightarrow \quad E + E \mid E * E \mid (E) \mid id$

Reduce the string id + id * id using the above grammar.

STACK	INPUT	Remark
$	id + id * id$	Shift
$ id	+ id * id$	Reduce by E → id
$E	+ id * id$	Shift
$E+	id * id$	Shift
$E+id	* id $	Reduce E → id
$E + E	* id $	Reduce E→ E + E
$E	id$	Shift
$E *	id$	Shift
$E * id	$	Reduce E → id
$E * E	$	Reduce E→ E * E
$E	$	Accept
$		

Operator Grammar

It is a bottom-up parser **none of whose productions contain two or more consecutive NTs in any RHS alternative**

Non terminals occurring in RHS string are separated by one or more terminal symbols

All terminal symbols occurring in RHS strings are called operators of the grammar

Not an operator grammar

$$E \rightarrow E \text{ op } E \mid id$$
$$\text{op} \rightarrow + \mid *$$

An operator grammar

$$E \rightarrow E + E \mid E * E \mid id$$

Operator precedence table

Operator precedence parser can parse some of the ambiguous grammar by constructing an operator precedence (relations) table

E → E + E | E * E | id

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Example

We parse the string

a) id + id * id

b) id + id + id

Using the grammar before and following operator precedence table

$E \rightarrow E + E \mid E * E \mid id$

id + id * id

Stack	Input	Action
$	id + id * id $	$ <· id, shift
$id	+ id * id $	id ·> +, reduce E → id
$	+ id * id $	$ <· id, shift
$+	id * id $	+ <· id, shift
$+id	* id $	id ·> +, reduce E → id
$+	* id $	+ <· *, shift
$+*	id $	* <· id, shift
$+*id	$	id ·> $, reduce E → id
$+*	$	* ·> $, reduce E → E*E
$+	$	+ ·> $, reduce E → E+E
$	$	Accept

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

<u>Note</u>: if op1 <· op2, shift operation
Otherwise reduce operation

Just to Note…

($\doteq$)
$ $\lessdot$ (
id $\gtrdot$)
) $\gtrdot$ $
($\lessdot$ (
$ $\lessdot$ id
id $\gtrdot$ $
) $\gtrdot$)
($\lessdot$ id

The rules for operator precedence will also be based on the precedence of operators

Exercise

Create an operator precedence table for the grammar

E → E+E | E-E | E*E | E/E | E^E | id

Using the operator precedence table, reduce the string

id * id ^ id + id

Disadvantages

Cannot handle unary minus

Solution: let lexical analyser handle

1. The lexical analyser will return two different tokens for the unary minus and the binary minus

2. The lexical analyser will need a lookhead to distinguish the binary minus from the unary minus

Hence, we make

$\theta \lessdot$ unary-minus → *for any operator*

unary-minus $\gtrdot \theta$ → *if unary-minus has higher precedence than* θ

unary-minus $\lessdot \theta$ → *if unary-minus has lower (or equal) precedence than* θ

Advantages

- Simple
- Powerful enough for expressions in programming languages

LALR parser

Kindly self study this topic