CD Moodle Assignment 3 🗒️

Jovial Joe Jayarson

13 June 2020

IES17CS016

1. Explain the main actions in a shift-reduce parser with an example.

Ans 1:

There are mainly four actions performed in shift-reduce parsing. They are:

- *Shift*: When the input string is parsed from left to right, the token to which the input pointer (`ip`) is directed, is push to a stack. This is also called `shift` operation.
- *Reduce*: Now the top most element if reduced using the production if it is a handle. It is called `reduce` operation.
- *Accept*: When the input string is completely parsed, and if the stack has nothing left except the `$` sign then the string is `accepted` by the shift-reduce parser.
- *Reject* (when Error): This action is performed when other symbols (except start symbol) are present in the stack, the input string is `rejected` even after input parsing is done.

Consider the grammar:

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E}$$

$$\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E}$$

$$\mathrm{E} \rightarrow (\ \mathrm{E}\ )$$

$$\mathrm{E} \rightarrow \mathrm{id}$$

Performing shift-reduce parsing on the input string $\mathrm{id} + \mathrm{id} * \mathrm{id}$:

Stack	Input	Input Pointer	Action	Production Used
$\$$	$\mathrm{id} + \mathrm{id} * \mathrm{id}\ \$$	id	`shift`	
$\$\ \mathrm{id}$	$+ \mathrm{id} * \mathrm{id}\ \$$	$+$	`reduce`	$\mathrm{E} \rightarrow \mathrm{id}$
$\$\ \mathrm{E}$	$+ \mathrm{id} * \mathrm{id}\ \$$	$+$	`shift`	
$\$\ \mathrm{E} +$	$\mathrm{id} * \mathrm{id}\ \$$	id	`shift`	
$\$\ \mathrm{E} + \mathrm{id}$	$* \mathrm{id}\ \$$	$*$	`reduce`	$\mathrm{E} \rightarrow \mathrm{id}$
$\$\ \mathrm{E} + \mathrm{E}$	$* \mathrm{id}\ \$$	$*$	`shift`	
$\$\ \mathrm{E} + \mathrm{E}\ *$	$\mathrm{id}\ \$$	id	`shift`	
$\$\ \mathrm{E} + \mathrm{E} * \mathrm{id}$	$\$$	$\$$	`reduce`	$\mathrm{E} \rightarrow \mathrm{id}$
$\$\ \mathrm{E} + \mathrm{E} * \mathrm{E}$	$\$$	$\$$	`reduce`	$\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E}$
$\$\ \mathrm{E} + \mathrm{E}$	$\$$	$\$$	`reduce`	$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E}$

Stack	Input	Input Pointer	Action	Production Used
$\$ \mathrm{E}$	$\$$	$\$$	`accept`	

> $\$ = eof$

There is nothing else in the stack other than the start symbol, so the input string $\mathrm{id} + \mathrm{id} * \mathrm{id}$ generated by the given grammar is accepted by the shift reduce parser.

2. Compare Handle and Handle Pruning.

Ans 2:

Handle:

- *Handle* of a string is a substring that matches the right side of a production.
- It's reduction to the non-terminal, on the left-side of the production, represents preceding step of the right-most derivation in reverse.
  e.g. In the production $\mathrm{S \rightarrow aABe}$, `aABe` is called a *handle*.

Handle Pruning:

- The process of replacing the *handle* by it's respective non-terminal is called *handle pruning*.
  e.g. If we replace `aABe` with `S` then that process is called *handle pruning*.

3. Construct canonical $\mathrm{LR}(0)$ collection of items for the grammar below:

$$\mathrm{S} \rightarrow \mathrm{L} = \mathrm{R}$$

$$\mathrm{S} \rightarrow \mathrm{R}$$

$$\mathrm{L} \rightarrow * \mathrm{R}$$

$$\mathrm{L} \rightarrow \mathrm{id}$$

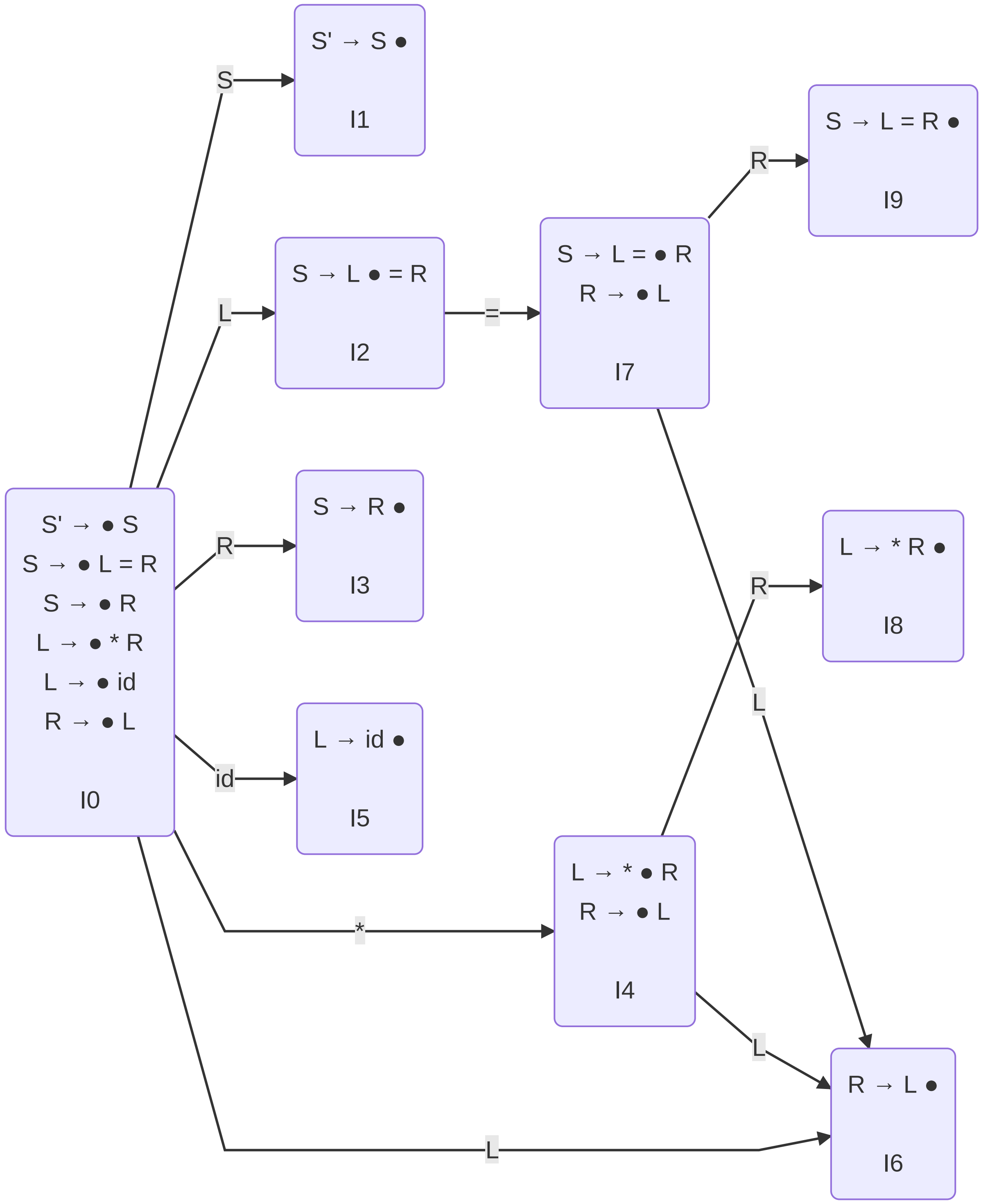
$$\mathrm{R} \rightarrow \mathrm{L}$$

Ans:

All those items with a '●' on the R.H.S. of a production is called LR(0) collection of items.

> Rules
>
> - Derive augmented grammar G' from the given grammar G.
> - To Find LR(0) collection of items:
>   - Find Closure
>   - Find GOTO

- Given Grammar:
  $\mathrm{S} \rightarrow \mathrm{L} = \mathrm{R}$
  $\mathrm{S} \rightarrow \mathrm{R}$
  $\mathrm{L} \rightarrow * \mathrm{R}$
  $\mathrm{L} \rightarrow \mathrm{id}$
  $\mathrm{R} \rightarrow \mathrm{L}$
- Augment Grammar:
  $\mathrm{S'} \rightarrow \mathrm{S}$
  $\mathrm{S} \rightarrow \mathrm{L} = \mathrm{R}$
  $\mathrm{S} \rightarrow \mathrm{R}$
  $\mathrm{L} \rightarrow * \mathrm{R}$
  $\mathrm{L} \rightarrow \mathrm{id}$
  $\mathrm{R} \rightarrow \mathrm{L}$
- Finding Closure
  $\mathrm{S'} \rightarrow \bullet\, \mathrm{S}$
  $\mathrm{S} \rightarrow \bullet\, \mathrm{L} = \mathrm{R}$
  $\mathrm{S} \rightarrow \bullet\, \mathrm{R}$
  $\mathrm{L} \rightarrow \bullet\, * \mathrm{R}$
  $\mathrm{L} \rightarrow \bullet\, \mathrm{id}$
  $\mathrm{R} \rightarrow \bullet\, \mathrm{L}$
- Finding GOTO



This is the canonical collection of $\mathrm{LR}(0)$ items.

4. Explain about the procedure of operator precedence parsing with an example.

- An operator precedence parser is a bottom-up parser that interprets an operator-precedence grammar.
- For example, most calculators use operator precedence parsers to convert from the human-readable infix notation relying on order of operations to a format that is optimized for evaluation such as Reverse Polish notation.
- This parsing tolerates ambiguous grammar.

> Rules for operator precedence parsing:
>
> - There should not be any ϵ production
> - No two non-terminals can be adjacent

e.g. Consider the grammar:

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid \mathrm{id}$$

We'll create an operator relation table:

	id	$+$	$*$	$\$$
id	—	$\gtrdot$	$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$*$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$\$$	$\lessdot$	$\lessdot$	$\lessdot$	—

Algorithm to perform operator precedence parsing:

```
for ip in input_string:
    if top == '$'  and ip == '$':
        print('Accept')
    else:
        if top == 'a' and ip == 'b':
            if 'a' ⋖ 'b' or 'a' == 'b':
                top == 'b' # push('b')
            elif 'a' ⋗ 'b':
                pop(top) # pop top-of-stack until ip is related to top by ⋖
                reduce(top)
            else:
```

```
                log('Error')
    # increment input-pointer
```

Performing the operator precedence parsing on the input string $\mathrm{id} + \mathrm{id} * \mathrm{id}$:

Stack	**Input**	**Input Pointer** ip	**Top of Stack** top	**Action**	**Reason**
$\$$	$\mathrm{id} + \mathrm{id} * \mathrm{id}\ \$$	id	$\$$	`push(id)`	$\because\ \$ \ \langle\bullet\ \mathrm{id}$
$\$\ \mathrm{id}$	$+\ \mathrm{id} * \mathrm{id}\ \$$	$+$	id	`pop(id)`	$\because\ \mathrm{id}\ \bullet\rangle\ +$
$\$$	$+\ \mathrm{id} * \mathrm{id}\ \$$	$+$	$\$$	`push(+)`	$\because\ \$ \ \langle\bullet\ +$
$\$ +$	$\mathrm{id} * \mathrm{id}\ \$$	id	$+$	`push(id)`	$\because\ +\ \langle\bullet\ \mathrm{id}$
$\$ + \mathrm{id}$	$*\ \mathrm{id}\ \$$	$*$	id	`pop(id)`	$\because\ \mathrm{id}\ \bullet\rangle\ *$
$\$ +$	$*\ \mathrm{id}\ \$$	$*$	$+$	`push(*)`	$\because\ + = *$
$\$ + *$	$\mathrm{id}\ \$$	id	$*$	`push(id)`	$\because\ *\ \langle\bullet\ \mathrm{id}$
$\$ + * \mathrm{id}$	$\$$	$\$$	id	`pop(id)`	$\because\ \mathrm{id}\ \bullet\rangle\ \$$
$\$ + *$	$\$$	$\$$	$*$	`pop(*)`	$\because\ *\ \bullet\rangle\ \$$
$\$ +$	$\$$	$\$$	$+$	`pop(+)`	$\because\ +\ \bullet\rangle\ \$$
$\$$	$\$$	$\$$	$\$$	`Accept`	$\because$ $\mathrm{ip} = \mathrm{top} = \$$

$\therefore$ The input string $\mathrm{id} + \mathrm{id} * \mathrm{id}$ generated by the given grammar is accepted by the operator precedence parser because both `top of stack` and `input pointer` have concluded in $\$$.