LR Parsing

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Top-Down vs. Bottom Up

Grammar: $S \rightarrow A\ B$

$A \rightarrow c \mid \varepsilon$

$B \rightarrow cbB \mid ca$

Input String: ccbca

Top-Down/leftmost		Bottom-Up/rightmost	
$S \Rightarrow AB$	$S \rightarrow AB$	$ccbca \Leftarrow Acbca$	$A \rightarrow c$
$\Rightarrow cB$	$A \rightarrow c$	$\Leftarrow AcbB$	$B \rightarrow ca$
$\Rightarrow ccbB$	$B \rightarrow cbB$	$\Leftarrow AB$	$B \rightarrow cbB$
$\Rightarrow ccbca$	$B \rightarrow ca$	$\Leftarrow S$	$S \rightarrow AB$

Bottom-Up parsing

- Bottom-up parsing ***reduces*** a string to the start symbol by inverting the derivation

Derivation (read upward)	Production
id * id + id	
id * T + id	$T \to id$
T + id	$T \to id * T$
T + T	$T \to id$
T + E	$E \to T$
E	$E \to T + E$

$E \to T + E$
$E \to T$
$T \to id$
$T \to id * T$
$T \to (E)$

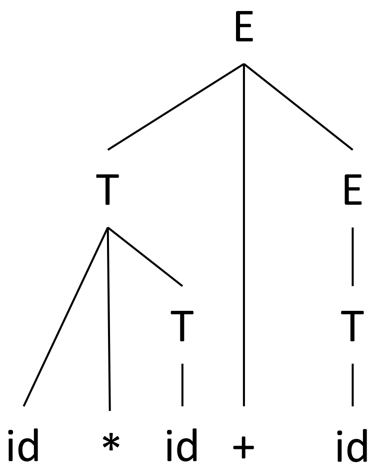
This is a rightmost derivation!

Bottom-up parse tree construction

Q: Write down the right sequence of shift-reduce actions and draw the parse tree for input `id*(id*id)`

- A shift-reduce parser traces a rightmost derivation in reverse

```
id * id + id
id * T  + id
T + id
T + T
T + E
E
```



Parse tree

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow id$

$T \rightarrow id * T$

$T \rightarrow (E)$

Notation

- Split string into two substrings: $\alpha \bullet \beta$
  - where $\alpha \in (N \cup T)^*$ and $\beta \in T^*$
  - Right sub-string is not examined yet; has only terminals
  - Left sub-string has terminals and non-terminals
- The dividing point is marked by a $\bullet$
  - $\bullet$ is not a part of the string
- Initially, all input is unexamined $\bullet$ $x_1 x_2 \ldots x_n$

Shift-Reduce Parsing

- Shift-reduce parsing uses only two kinds of actions:
  - **Shift**: Move ● one place to the right
    - Shift a terminal to the left string

      ABC ● xyz ⇒ ABCx ● yz
  - **Reduce**: Apply a CFG rule to the string left of the ●
    - If $A \rightarrow xy$ is a production, then reduce

      Cbxy ● ijk ⇒ CbA ● ijk

Shift-Reduce Parsing

$E \rightarrow T + E$
$E \rightarrow T$
$T \rightarrow id$
$T \rightarrow id * T$
$T \rightarrow (E)$

● id * id + id	Shift
id ● * id + id	Shift
id * ● id + id	Shift
id * id ● + id	Reduce T → id
id * T ● + id	Reduce T → id * T
T ● + id	Shift
T + ● id	Shift
T + id ●	Reduce T → id
T + T ●	Reduce E → T
T + E ●	Reduce E → T + E
E ●	

Shift-Reduce Parsing

E → T + E

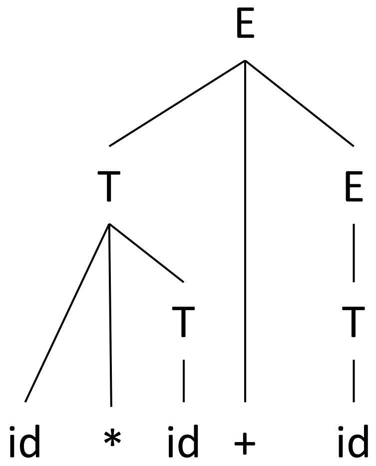
E → T

T → id

T → id * T

T → (E)

```
● id * id + id
id ● * id + id
id * ● id + id
id * id ● + id
id * T ● + id
T ● + id
T + ● id
T + id ●
T + T ●
T + E ●
E ●
```



Stack

- Left part of the string is implemented by a stack
  - Top of the stack is left of the ●
- Shift pushes a terminal on the stack
- Reduce
  - Pops 0 or more symbols off of the stack (rhs of one rule from the CFG)
  - Pushes a non-terminal on the stack (lhs of one rule from the CFG)

Conflicts

- In a given state, more than one action (shift/reduce) may lead to different valid parse
- If it is legal to either shift or reduce: shift-reduce conflict
  - Can be fixed (precedence and associativity declaration)
- If it is legal to reduce by two different rules: reduce-reduce conflict
  - There is ambiguity in the grammar
  - Might be fixed by additional lookahead

When to shift/reduce?

- Consider step id ● * id + id
- Shift action: id *● id +id
- Reduce action: reduce by T→id giving T●*id +id
- It causes fatal error:
  - No way to reduce to the start symbol E
- Reduce is possible, but it is not a valid action

E → T + E

E → T

T → id

T → id * T

T → (E)

Q: For the same input `id*id+id` find another shift/reduce choice in the derivation where a shift over reduce leads to `id*E` which cannot be reduced further.

Viable Prefix and Handle

- Intuition: reduce only if we can eventually reach the start symbol
- Assume a rightmost derivation
  - $S \Rightarrow^* \alpha X \beta \Rightarrow \alpha w \beta$

    ⟵ reduction
- Then αw is a viable prefix of $\alpha w \beta$
  - A handle w is valid if we can reduce w to X
  - We only reduce a handle
- A handle *always* appears on top of the stack, never inside

Bottom-up Shift-Reduce Parsing Algorithms

- LR(k) parsing:
  - L: scan input Left-to-right
  - R: produce Rightmost derivation
  - k: tokens of lookahead (k=1 is sufficient)
- LR(0): zero tokens of lookahead
- SLR: Simple LR, similar to LR(0), but uses Follow sets
- LALR(k)
- These algorithms work with left- or right-recursive grammars

Recognizing a Viable Prefix

- LR parsing algorithms are based on recognizing viable prefixes
- We can identify viable prefixes only for a subset of CFGs
- Adding lookahead helps: (0) or (1) or (k) symbols of lookahead
- For this subset of CFGs, LR parsing is a deterministic linear-time algorithm.

Hierarchy of grammars