LR Parsing

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Bottom-Up Parsing

- Bottom-up parsing is more general than (deterministic) top-down parsing
  - Just as efficient
  - Builds on ideas in top-down parsing
- Preferred method in practice
- Do not need left-factored grammars!

Bottom-Up parsing

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

int * int + int	$T \rightarrow$ int
int * T + int	$T \rightarrow$ int * T
T + int	$T \rightarrow$ int
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

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

Note the productions, read reverse (i.e. from bottom to top)

This is a rightmost derivation!

Bottom-up parse

- Fact #1: A bottom-up parser traces a rightmost derivation in reverse

int * int + int

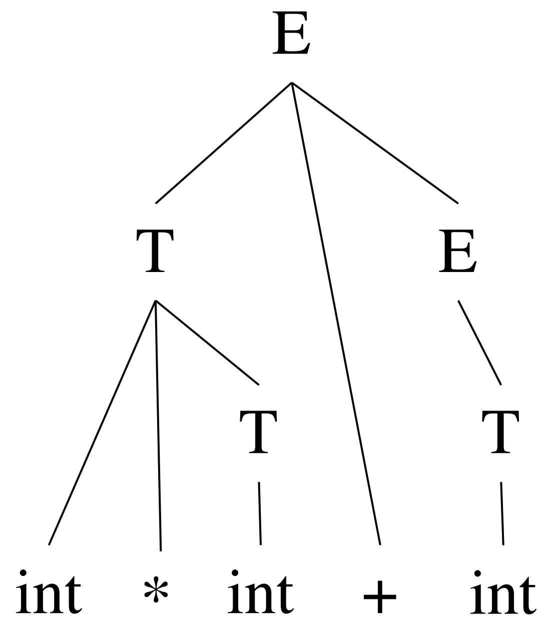
int * T + int

T + int

T + T

T + E

E



Parse tree

$E \rightarrow T + E$

$E \rightarrow T$

$T \rightarrow \text{int}$

$T \rightarrow \text{int} * T$

$T \rightarrow (E)$

Reductions during Parsing

- Fact #1 has an interesting consequence:
  - Let $\alpha\ \beta\ \omega$ be a step of a bottom-up parse
  - Assume the next reduction is by $X \rightarrow \beta$
  - Then ω is a (possibly empty) string of terminals
- Why? Because $\alpha X \omega \Rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Notation

- Idea: Split string into two substrings
  - Right sub-string is as yet unexamined by parsing
  - Left sub-string has terminals and non-terminals
- The dividing point is marked by a |
  - | is not a part of the string
- Initially, all input is unexamined | $x_1\ x_2 \ldots x_n$

Shift-Reduce Parsing

- Bottom-up parsing uses only two kinds of actions:
  - Shift: Move | one place to the right
    - Shift a terminal to the left string

      $ABC \mid xyz \Rightarrow ABCx \mid yz$
  - Reduce: Apply an inverse production at the right end of the left string
    - If $A \to xy$ is a production, then reduce

      $Cbxy \mid ijk \Rightarrow CbA \mid ijk$

Shift-Reduce Parsing

| int * int + int	Shift
int | * int + int	Shift
int * | int + int	Shift
int * int | + int	Reduce T → int
int * T | + int	Reduce T → int * T
T | + int	Shift
T + | int	Shift
T + int |	Reduce T → int
T + T |	Reduce E → T
T + E |	Reduce E → T + E
E |	

Shift-Reduce Parsing

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

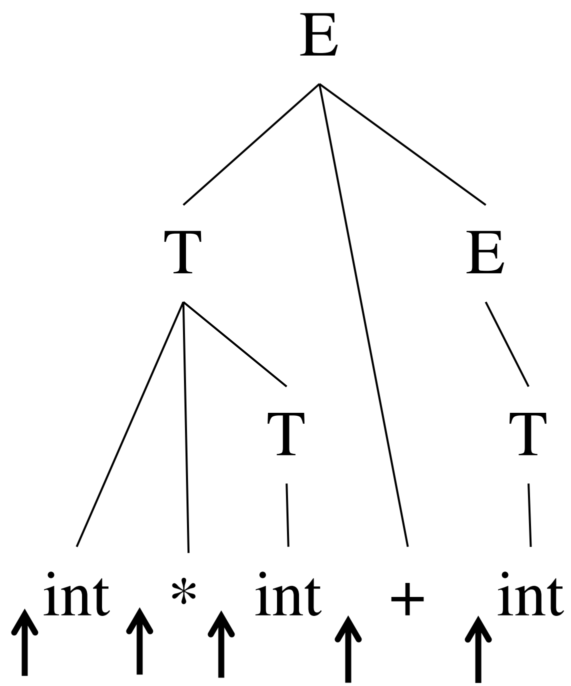
T + | int

T + int |

T + T |

T + E |

E |



Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce
  - Pops 0 or more symbols off of the stack (production rhs)
  - Pushes a non-terminal on the stack (production lhs)

Conflicts

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

When to shift/reduce?

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

- Consider step int | * int + int
  - We should shift, int * | int +int
  - We could reduce by $T \to$int giving T|*int +int
  - It causes fatal error:
    - No way to reduce to the start symbol E
  - Reduce is possible, but it is not a valid action

Handles

- Intuition: we want to reduce only if the result can still be reduced to the start symbol
- Assume a rightmost derivation
  - $S \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$

    reduction
- Then $\alpha\beta$ is a handle of $\alpha\beta\omega$
  - It says: it is OK to reduce β to X

Handles

- Handles formalize the intuition
  - A handle is a reduction that also allows further reductions back to the start symbol
- We only want to reduce at handles

- Important Fact: Handles just appear on top of the stack, never inside

Recognizing Handles

- Bottom-up parsing algorithms are based on recognizing handles
- No efficient algorithms to recognize handles
- There are good heuristics for guessing handles
- On some CFGs, the heuristics always work correctly

Bottom-up Parsing Algorithms

- LR(k) parsing:
  - L: scan input Left-to-right
  - R: produce Rightmost derivation
  - k: tokens of lookahead (in practice k=1)
- LR(0): zero tokens of lookahead
- SLR: Simple LR, similar to LR(0), but uses Follow sets
- LALR(k)

Bottom-up Parsing Algorithms