Compilation Principle
编 译 原 理

第7讲：语法分析(4)

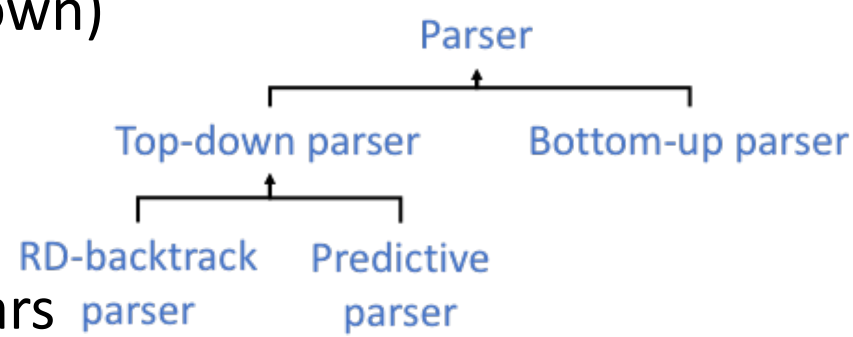
张献伟

xianweiz.github.io

DCS290, 03/23/2021

Bottom-up Parsing[自底向上]

- Begins at leaves and works to the top
  - Bottom-up: **reduce**s[归约] input string to start symbol
  - In the opposite direction from top-down
    - Top-down: expands start symbol to input string
  - In reverse order of rightmost derivation (In effect, builds tree from left to right, just like top-down)



- More powerful than top down
  - Don’t need left factored grammars
  - Can handle left recursion
  - Can express a larger set of languages
  - And just as efficient

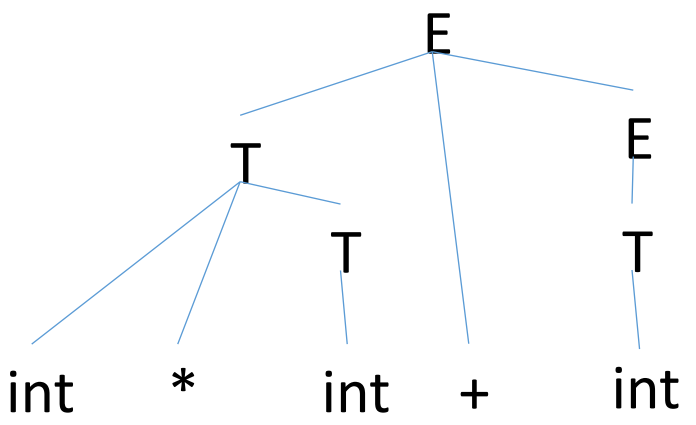
Example

- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String: int * int + int



- The rightmost derivation of the parse tree
  - E ⇒ T + E ⇒ T + T ⇒ T + int ⇒ int * T + int ⇒ int * int + int

- To recognize the string via bottom-up parsing
  - int * int + int ⇒ int * T + int ⇒ T + int ⇒ T + T ⇒ T + E ⇒ E

Bottom-up: Overview

- An important fact:
  - 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 string of terminals [i.e., 句子]
- Why? $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a rightmost derivation
- **Idea**: split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals
- The dividing point is marked by a #
  - The # is not part of the string
  - Initially, all input is unexamined $\#x_1x_2 \ldots x_n$

Bottom-up: Shift-Reduce[移入-归约]

- Bottom-up parsing is also known as **Shift-Reduce** parsing
  - Involves two types of operations: shift and reduce

- **Shift**: move # one place to the right
  - Shifts a terminal to the left string

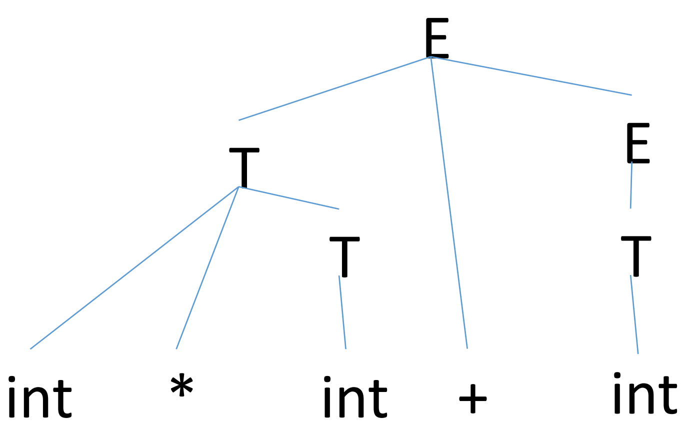
    ABC#xyz ⇒ ABCx#yz

- **Reduce**: apply an inverse production at the right end of the left string
  - If E → Cx is a production, then

    ABCx#yz ⇒ ABE#yz

The Example

- Grammar
  - E → T+E | T
  - T → int*T | int | (E)
- String
  - int * int + int



Sentential form	Operation
#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 #	Shift
T + T #	Reduce T → int
T + E #	Reduce E → T
E #	Reduce E → T+E

Stack[栈]

- Left string can be implemented by a **stack**
  - Top of the stack is the #

- **Shift** pushes a terminal on the stack

- **Reduce** does the following:
  - pops zero or more symbols off of the stack
    - production rhs
  - pushes a non-terminal on the stack
    - production lhs

Key Issue

- How to decide when to shift or reduce?
  - Example grammar:

    E → T+E|T

    T → int*T | int | (E)

Sentential form	**Operation**
#int * int + int	Shift
int# * int + int	Reduce T → int
T # * int + int	Shift
… … …	

  - Consider the step int # * int + int
  - We could reduce by T → int giving T#*int + int
    - A fatal mistake: no way to reduce to the start symbol E

- Intuition: want to reduce only if the result can still be reduced to the start symbol

Handle[句柄]

- Informally:
  - RHS of a production rule that, when reduced to LHS, will lead to the start symbol
- Definition: let $\alpha\beta\omega$ be a sentential form where:
  - α, β is a string of terminals and non-terminals (yet to be derived)
  - ω is a string of terminals (already derived)
  - Then β is a **handle** of $\alpha\beta w$ if:

    $S \Rightarrow^* \alpha X\omega \Rightarrow \alpha\beta\omega$ by a rightmost derivation (apply rule $X \to \beta$)
- We only want to reduce at handles, and there is exactly one handle per sentential form
  - But where to find it?

Handle: Example

- Grammar

  E → T+E | T

  T → int*T | int | (E)

- String

  int * int + int

Step	Operation
#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** #	Shift
T + **T** #	Reduce T → int
T + E #	Reduce E → T
E #	Reduce E → T+E

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Handle Always Occurs at Stack Top

- Why can't a handle occur on right side of #?
  - It can
  - But handle will eventually be shifted in, placing it at top of stack
  - In int * #int + int ⇒ int * int # + int, int is eventually shifted to the top
- Why can't a handle occur on left side of #, i.e., in middle of the stack?
  - Can int * int + # int occur? No.
  - Means parser shifted when it could have reduced when the handle was on top
  - If parser eagerly reduces when handle is at top of stack, never occurs
- Makes life easier for parser (need only access top of stack)

Ambiguous Grammars

- Conflicts arise with ambiguous grammars
  - Bottom up parsing predicts action w/ lookahead (just like LL)
  - If there are multiple correct actions, parse table will have conflicts
- Example:
  - Consider the ambiguous grammar $E \rightarrow E * E \mid E + E \mid (E) \mid int$

Sentential form	Actions
int * int + int	shift
…	…
E * E # + int	**reduce E → E * E**
E # + int	shift
E + # int	shift
E + int #	reduce E → int
E + E #	reduce E → E + E
E #	

Sentential form	Actions
int * int + int	shift
…	…
E * E # + int	**shift**
E * E + # int	shift
E * E + int #	reduce E → int
E * E + E #	reduce E → E + E
E * E #	reduce E → E * E
E #	

Ambiguous Grammars (cont.)

- In the red step shown, can either shift or reduce by E → E * E
  - Both okay since precedence of + and * not specified in grammar
  - Same problem with associativity of + and *
- As usual, remove conflicts due to ambiguity ...
  - 1. Rewrite grammar/parser to encode precedence and associativity
    - Rewriting grammar results in more convoluted grammars
    - Parser tools have other means to encode precedence and association
  - 2. Get rid of remaining ambiguity (e.g. if-then-else)
    - No choice but to modify grammar
- Is ambiguity the only source of conflicts?
  - Limitations in lookahead-based prediction can cause conflicts
  - But these cases are very rare

Properties of Bottom-up Parsing

- Handles always appear at the **top of the stack**
  - Never in middle of stack
  - Justifies use of stack in shift – reduce parsing
- Results in an easily generalized **shift – reduce** strategy
  - If there is no handle at the top of the stack, shift
  - If there is a handle, reduce to the non-terminal
  - Easy to automate the synthesis of the parser using a table
- Can have conflicts
  - If it is legal to either shift or reduce then there is a shift-reduce conflict
  - If there are two legal reductions, then there is a reduce-reduce conflict
  - Most often occur because of ambiguous grammars
    - In rare cases, because of non-ambiguous grammars not amenable to parser