

Compilation Principle 编译原理

张献伟

<u>xianweiz.github.io</u>

DCS290, 3/16/2021





Review Questions (1)

- Why don't we keep using RE in syntax parsing? RE is not powerful enough, it cannot express nested structures
- Formal definition of Grammar?
 (T, N, s, σ): T terminals; N non-terminals, s start, σ productions
- What is CFG?
 Context free grammar
- Language classification based on grammar rules?
 0 unrestricted, 1 context sensitive, 2 context free, 3 regular
- What is derivation? What is parse tree? Derivation is the sequence of applying production rules. Parse tree is a graphical representation of the derivation.

Review Questions (2)

- What is leftmost derivation? Always replace the leftmost non-terminal in each derivation step.
- Grammar G: $E \rightarrow E^*E \mid E+E \mid (E) \mid id$

```
T = {*, +, (, ), id}

N = {E}

s = E

\sigma = E → E*E | E+E | (E) | id
```

- Is id + E * E an sentence of grammar G?
 NO. It is a sentential form (句型), as E is non-terminal symbol.
- Is <u>id + id * id</u> an sentence of grammar G? YES. It can be derived using the production rules.

Derivation Example

• Grammar: $E \rightarrow E^*E \mid E+E \mid (E) \mid id$

• Leftmost derivation $E \Rightarrow E + E$ $\Rightarrow id + E$ $\Rightarrow id + E * E$ $\Rightarrow id + id * E$ $\Rightarrow id + id * id$







Derivation Example (cont.)

• Grammar $E \rightarrow E^*E \mid E+E \mid (E) \mid id$

• Leftmost derivation $E \Rightarrow E * E$ $\Rightarrow E + E * E$ $\Rightarrow id + E * E$ $\Rightarrow id + id * E$ $\Rightarrow id + id * id$





Derivation Example (cont.)

- Two distinct leftmost derivations for the sentence id + id * id
 - Above: id + (id * id)
 - Below: (id + id) * id
- How to evaluate a + b * c ?
 a + (b * c) ?
 (a + b) * c ?
- Grammar E→E*E | E+E | (E) | id is ambiguous





Ambiguity[二义性]

- grammar G is ambiguous if
 - It produces more than one parse tree some sentence
 - i.e., there exist a string $str \in L(G)$ such that
 - more than one parse tree derives str
 - ≡ more than one leftmost derivation derives *str*
 - ≡ more than one rightmost derivation derives *str*
- Unambiguous grammars are preferred for most parsers
 - If not, we cannot uniquely determine which parse tree to select for a sentence
 - In minor cases, it is convenient to use carefully chosen ambiguous grammars, together with disambiguating rules that "throw away" undesirable parse trees, leaving only one tree for each sentence



Ambiguity (cont.)

- Ambiguity is the property of the grammar, not the language
 - Just because G is ambiguous, does not mean L(G) is inherently ambiguous
 - A G' can exist where G' is unambiguous and $L(G') \equiv L(G)$
- Impossible to convert ambiguous to unambiguous grammar automatically
 - It is (often) possible to rewrite grammar to remove ambiguity
 - Or, use ambiguous grammar, along with disambiguating rules





Precedence and Associativity

- Two characteristics of operators that determine the evaluation order of sub-expressions w/o brackets
- Operator precedence[优先级]
 - Determines which operator is performed first in an expression with more than one operators with different precedence.
 - 10 + 20 * 30
- Operator associativity[结合性]
 - Is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.
 - 100 / 10 * 10 -> (100 / 10) * 10 //left associativity





How to Remove Ambiguity?

- Step I: Specify precedence[指定优先级]
 - Build precedence into grammar by recursion on a different nonterminal for each precedence level. Insight:
 - Lower precedence tend to be higher in tree (close to root)
 - Higher precedence tend to be lower in tree (far from root)
 - Place lower precedence non-terminals higher up in the tree
- Rewrite $E \rightarrow E^*E \mid E+E \mid (E) \mid id$ to:
 - $E \rightarrow E + E \mid T$ $T \rightarrow T * T \mid F$ $F \rightarrow (E) \mid id$
 - $E \rightarrow E + E \mid T$ //lowest precedence +
 - $T \rightarrow T * T | F$ //middle precedence *
 - //highest precedence ()



How to Remove Ambiguity (cont.)

- Step II: Specify associativity[指定结合性]
 - Allow recursion only on either left or right non-terminal
 Left associative recursion on left non-terminal
 Right associative recursion on right non-terminal
- Even after step 1, ambiguous due to associativity
 E → E + E; allows both left/right associativity
- Rewrite:

```
E \rightarrow E + E \mid T

T \rightarrow T * T \mid F

F \rightarrow (E) \mid id

to

E \rightarrow E + T \mid T

T \rightarrow T * F \mid F

F \rightarrow (E) \mid id
```

 $E \rightarrow E + E \mid T$ //lowest precedence + $T \rightarrow T * T \mid F$ //middle precedence * $F \rightarrow (E) \mid id$ //highest precedence ()

 $E \rightarrow E + T \mid T$ // + is left-associative $T \rightarrow T * F \mid F$ // * is left-associative





The Example

- Grammar E→E*E | E+E | (E) | id was ambiguous
 - Rewrite to
 - $E \rightarrow E + T | T$ T \rightarrow T * F | F F \rightarrow (E) | id
- The id + id * id has only one parse tree now
 - $E \Rightarrow E + T$
 - \Rightarrow T + T
 - \Rightarrow F + T

中山大學

- \Rightarrow id + T * F
- $\Rightarrow id + F * F$ $\Rightarrow id + id * F$
- \Rightarrow id + id * id





Grammar \rightarrow Parser

- What exactly is **parsing**, or syntax analysis?
 - To process an input string for a given grammar,
 - and compose the derivation if the string is in the language
 - Two subtasks
 - determine if string can be derived from grammar or not
 - build a representation of derivation and pass to next phase
- What is the best representation of the derivation?
 - Can be a parse tree or an abstract syntax tree
- An abstract syntax tree is[抽象语法树]
 - Abbreviated representation of a parse tree
 - Drops some details without compromising meaning
 - some terminal symbols that no longer contribute to semantics are dropped (e.g. parentheses)
 - internal nodes may contain terminal symbols





Example: Abstract Syntax Tree





Summary

- Compilers specify program structure using CFG
 - Most programming languages are not context free
 - Context sensitive analysis can easily be separated out to semantic analysis phase
- A parser uses CFG to
 - ... answer if an input str $\in L(G)$
 - ... and build a parse tree
 - ... or build an AST instead
 - ... and pass it to the rest of compiler





Parser Types[分析器类型]

- Most compilers use either top-down or bottom-up parsers
- Top-down parsing[自顶向下分析]
 - Starts from root and expands into leaves
 - Tries to expand start symbol to input string
 - □ Finds leftmost derivation[最左推导]
 - In each step
 - Which non-terminal to replace?
 - Which production of the non-terminal to use?
 - Parser code structure closely mimics grammar
 - Amenable to implementation by hand
 - Automated tools exist to convert to code (e.g. ANTLR)



Parser Types (cont.)

- Top-down parsing[自顶向下分析]
 - Starts from root and expands into leaves
- Bottom-up parser[自底向上分析]
 - Starts at leaves and builds up to root
 - Tries to reduce the input string to the start symbol
 - □ Finds reverse order of the rightmost derivation[最右推导的逆 → 最左 归约, 也称为规范归约]
 - Parser code structure nothing like grammar
 - Very difficult to implement by hand
 - Automated tools exist to convert to code (e.g. Yacc, Bison)
 - □ $LL \subset LR$ (Bottom-up works for a larger class of grammars)





Example

• Consider a CFG grammar G $S \rightarrow AB$ $A \rightarrow aC$ $B \rightarrow bD$ $D \rightarrow d$ $C \rightarrow c$

18

This language has only one sentence: L(G) = {acbd}
 Top-down (Leftmost Derivation)
 Bottom-up (reverse of



rightmost derivation) $S \Rightarrow AB(5)$ \Rightarrow AbD (4) \Rightarrow Abd (3) \Rightarrow aCbD (2) \Rightarrow acbd (1) R а b

Top-down Parsers[自顶向下]

- Recursive descent parser (RDP, 递归下降分析) with backtracking[回溯]
 - Implemented using recursive calls to functions that implement the expansion of each non-terminal
 - Goes through all possible expansions by trial-and-error until match with input; backtracks when mismatch detected
 - Simple to implement, but may take exponential time
- Predictive parser[预测分析]
 - Recursive descent parser with prediction (no backtracking)
 - Predict next rule by looking ahead k number of symbols
 - Restrictions on the grammar to avoid backtracking
 - Only works for a class of grammars called LL(k)



RDP with Backtracking

- **Approach**: for a non-terminal in the derivation, productions are tried in some order until
 - A production is found that generates a portion of the input, or
 - No production is found that generates a portion of the input, in which case backtrack to previous non-terminal
- Terminals of the derivation are compared against input
 - Match: advance input, continue parsing
 - Mismatch: backtrack, or fail
- Parsing fails if no derivation generates the entire input



Recursive Decent Example

- Consider the grammar $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- To construct a parse tree top-down for input string *w*=cad
 - Begin with a tree consisting of a single node labeled S
 - The input pointer pointing to *c*, the first symbol of *w*
 - S has only one production, so we use it to expand S and obtain the tree





Recursive Decent Example (cont.)

- Consider the grammar
 - $S \rightarrow cAd A \rightarrow ab | a$
- To construct a parse tree top-down for input string w=cad
 - The leftmost leaf, labeled *c*, matches the first symbol of *w*
 - \square So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf A
 - Next, expand A using $A \rightarrow ab$
 - Have a match for the 2nd input symbol, a, so advance the input pointer to d, the 3rd input symbol





Recursive Decent Example (cont.)

- Consider the grammar $S \rightarrow cAd \quad A \rightarrow ab \mid a$
- To construct a parse tree top-down for input string w=cad
 - b does not match d, report failure and go back to A
 - See whether there is another alternative for A that has not been tried
 In going back to A, we must reset the input pointer as well
 - Leaf a matches the 2nd symbol of *w*, and leaf *d* matches the 3rd
 - We have produced a parse tree for *w*, we halt and announce successful completion of parsing





Left Recursion Problem[左递归]

- Recursive descent doesn't work with left recursion
 - Right recursion is OK
- Why is left recursion[左递归] a problem?
 - For left recursive grammar

A→Ab|c

We may repeatedly choose to apply A b

 $\mathsf{A} \Rightarrow \mathsf{A} \mathsf{b} \Rightarrow \mathsf{A} \mathsf{b} \mathsf{b} \dots$

- Sentence can grow indefinitely w/o consuming input
- How do you know when to stop recursion and choose c?
- Rewrite the grammar so that it is right recursive[改为右递 归]
 - Which expresses the same language



Left Recursion

- A grammar is left recursive if
 - It has a nonterminal A such that there is a derivation $A \Rightarrow + A\alpha$ for some string α
- Recursion types [直接和间接左递归]
 - Immediate left recursion, where there is a production $A \rightarrow A\alpha$
 - Non-immediate: left recursion involving derivation of 2+ steps
 - S → Aa | b
 - $A \rightarrow Sd \mid \epsilon$
 - $S \Rightarrow Aa \Rightarrow Sda$
- Algorithm to systematically eliminates left recursion from a grammar



Remove Left Recursion[消除左递归]

- Grammar: A \rightarrow A $\alpha \mid \beta (\alpha \neq \beta, \beta \text{ doesn't start with A})$
 - $A \Rightarrow A\alpha$
 - $\Rightarrow A\alpha\alpha$
 - •••
 - $\Rightarrow A\alpha...\alpha\alpha$
 - $\Rightarrow \beta \alpha ... \alpha \alpha$
 - **r**= βα*
- Rewrite to:
 - $A \rightarrow \beta A'$ $A' \rightarrow \alpha A' | \varepsilon$ $\Rightarrow \alpha \alpha A'$

...

//begins with β (A' is a new non-terminal) //A' is to produce a sequence of α

 $\Rightarrow \alpha ... \alpha A' \Rightarrow \alpha ... \alpha$



Remove Left Recursion (cont.)

- Grammar: $A \rightarrow A\alpha \mid \beta$ to $A \rightarrow \beta A'$ $A' \rightarrow \alpha A' | \epsilon$ $E \rightarrow TE'$ • $E \rightarrow E + T | T$ $E' \rightarrow +TE' | \epsilon$ ß α $T \rightarrow FT'$ • T \rightarrow T * F | F $T' \rightarrow *FT' | \epsilon$ α β
- $F \rightarrow (E) \mid id$ $F \rightarrow (E) \mid id$



Summary of Recursive Descent

- **Recursive descent** is a simple and general parsing strategy
 - Left-recursion must be eliminated first
 - Can be eliminated automatically using some algorithm
 - L(Recursive_descent) \equiv L(CFG) \equiv CFL
- However it is not popular because of backtracking
 - Backtracking requires re-parsing the same string
 - Which is inefficient (can take exponential time)
 - Also undoing semantic actions may be difficult
 E.g. removing already added nodes in parse tree
- Real world parsers do no (or minimal) backtracking ...
 At the cost of restricting the class of grammar

