Compilation Principle
编 译 原 理

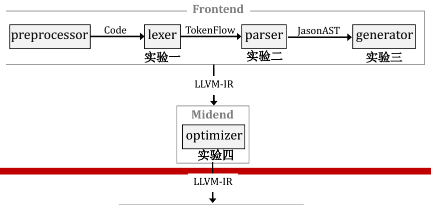
第7讲：语法分析(4)

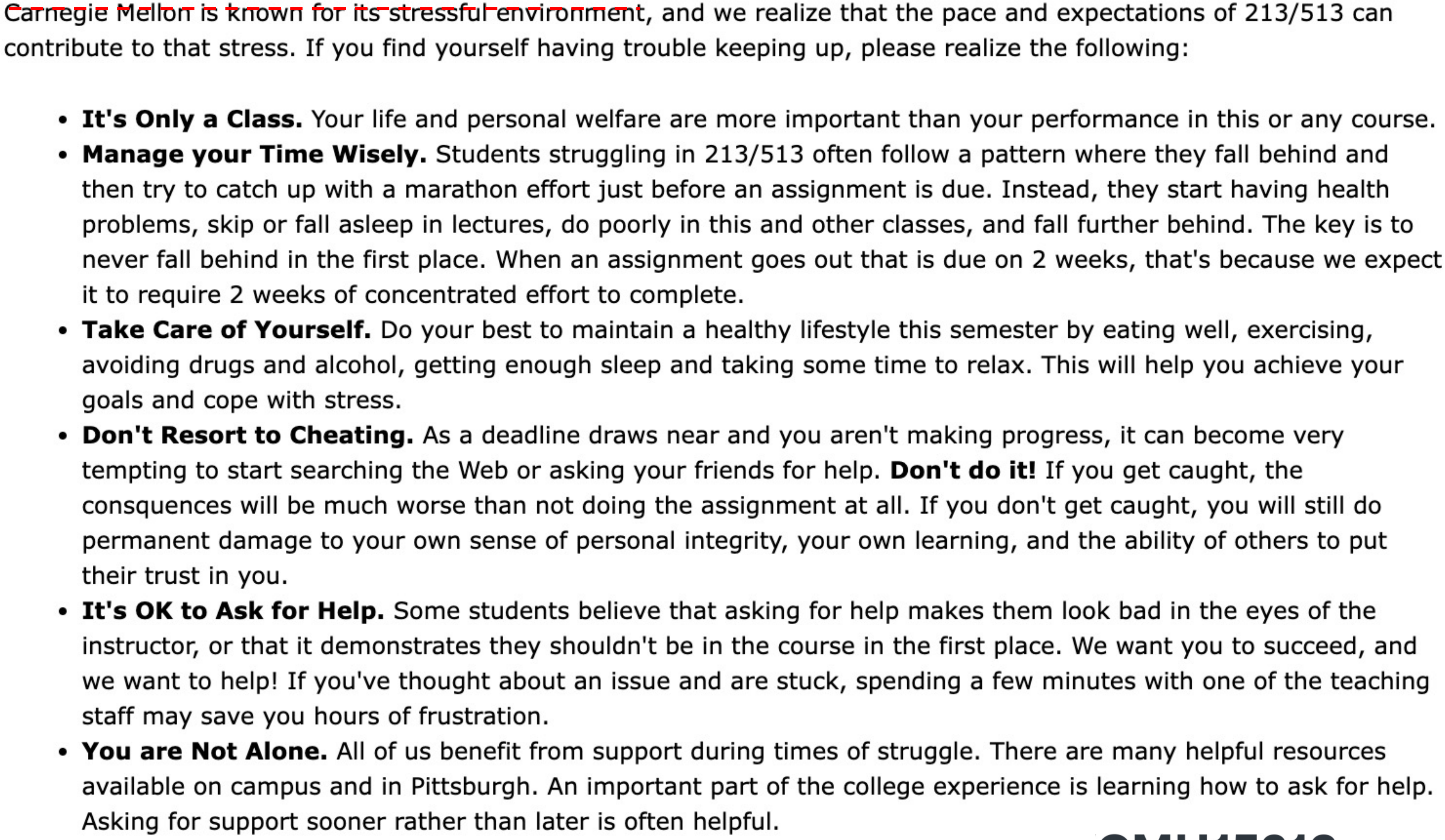
张献伟

xianweiz.github.io

DCS290, 3/16/2023

Your Well Being





~~Carnegie Mellon is known for its stressful environment~~, and we realize that the pace and expectations of 213/513 can contribute to that stress. If you find yourself having trouble keeping up, please realize the following:

- **It's Only a Class.** Your life and personal welfare are more important than your performance in this or any course.
- **Manage your Time Wisely.** Students struggling in 213/513 often follow a pattern where they fall behind and then try to catch up with a marathon effort just before an assignment is due. Instead, they start having health problems, skip or fall asleep in lectures, do poorly in this and other classes, and fall further behind. The key is to never fall behind in the first place. When an assignment goes out that is due on 2 weeks, that's because we expect it to require 2 weeks of concentrated effort to complete.
- **Take Care of Yourself.** Do your best to maintain a healthy lifestyle this semester by eating well, exercising, avoiding drugs and alcohol, getting enough sleep and taking some time to relax. This will help you achieve your goals and cope with stress.
- **Don't Resort to Cheating.** As a deadline draws near and you aren't making progress, it can become very tempting to start searching the Web or asking your friends for help. **Don't do it!** If you get caught, the consquences will be much worse than not doing the assignment at all. If you don't get caught, you will still do permanent damage to your own sense of personal integrity, your own learning, and the ability of others to put their trust in you.
- **It's OK to Ask for Help.** Some students believe that asking for help makes them look bad in the eyes of the instructor, or that it demonstrates they shouldn't be in the course in the first place. We want you to succeed, and we want to help! If you've thought about an issue and are stuck, spending a few minutes with one of the teaching staff may save you hours of frustration.
- **You are Not Alone.** All of us benefit from support during times of struggle. There are many helpful resources available on campus and in Pittsburgh. An important part of the college experience is learning how to ask for help. Asking for support sooner rather than later is often helpful.

CMU15213

https://www.cs.cmu.edu/~213/personal.html

Review Questions

- Grammar G: $E \rightarrow T \,/\, E \mid T$, result of 6 - 4 / 2?
  $T \rightarrow T - T \mid \text{id}$

  (6 - 4) / 2 = 1

- Regard id - id - id, is G ambiguous?

  Yes. No associativity is specified for operator -.

- How to remove the ambiguity?

  T → T - F, F → id

- Regard AST tree build, how to classify parser?

  Top-down (root to leaves), bottom-up (leaves to root).

- Which parser type is more similar to derivation?

  Top-down, mimics leftmost derivation.

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[非终结符-展开]
  - Goes through all possible expansions by **trial-and-error** until match with input; backtracks when mismatch detected[试错-回溯]
  - 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)

Classify rule: for a non-terminal, which production to use?

RDP with Backtracking[回溯]

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

Example

- Consider the grammar

  S → cAd   A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*

Example

- Consider the grammar

  S → cAd A → ab | a

- To construct a parse tree top-down for input string *w*=cad
  - Begin with a tree consisting of a single node labeled *S*

S

Example

- Consider the grammar

  S → cAd   A → ab | 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*

Example

- Consider the grammar

  S → cAd A → ab | 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

Example

- Consider the grammar

  S → cAd   A → ab | 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

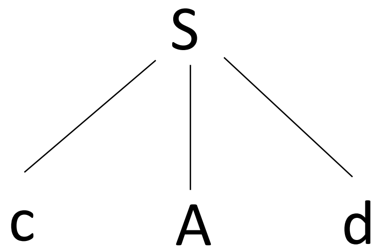
S

Example

- Consider the grammar

  S → cAd   A → ab | 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

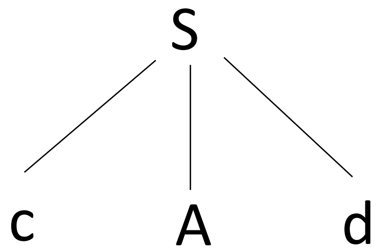


Example (cont.)

- Consider the grammar

  S → cAd   A → 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*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*

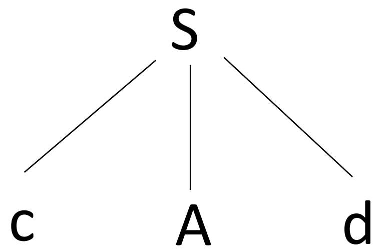


Example (cont.)

- Consider the grammar

  S → cAd   A → 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*
    - So we advance the input pointer to *a* (i.e., the 2nd symbol of *w*) and consider the next leaf *A*

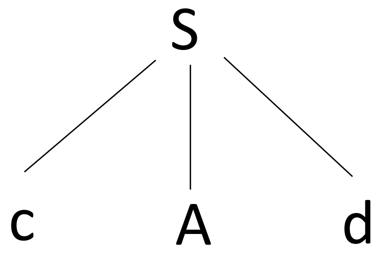


Example (cont.)

- Consider the grammar

  S → cAd   A → 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*
    - 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* → *ab*
    - Have a match for the 2nd input symbol, *a*, so advance the input pointer to *d*, the 3rd input symbol

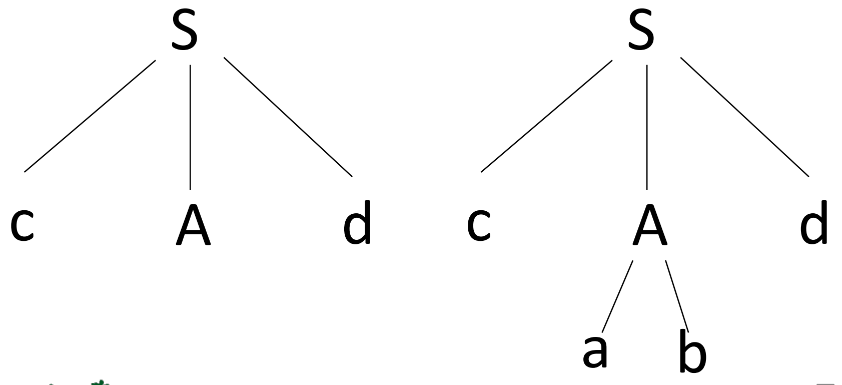


Example (cont.)

- Consider the grammar

  S → cAd   A → 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*
    - 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* → *ab*
    - Have a match for the 2nd input symbol, *a*, so advance the input pointer to *d*, the 3rd input symbol

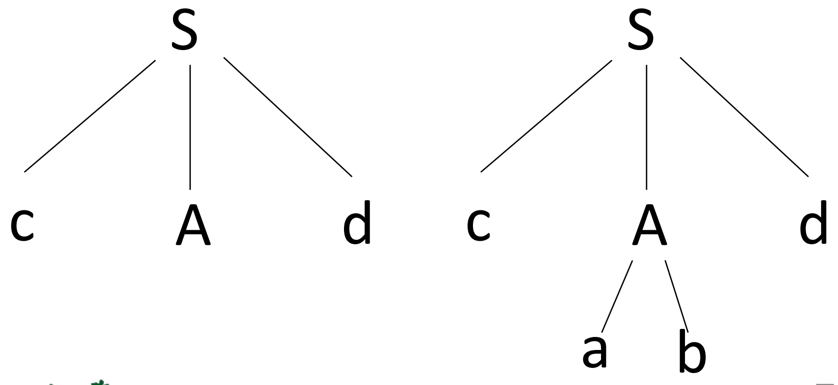


Example (cont.)

- Consider the grammar

  S → cAd   A → 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*
    - 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* → *ab*
    - Have a match for the 2nd input symbol, *a*, so advance the input pointer to *d*, the 3rd input symbol

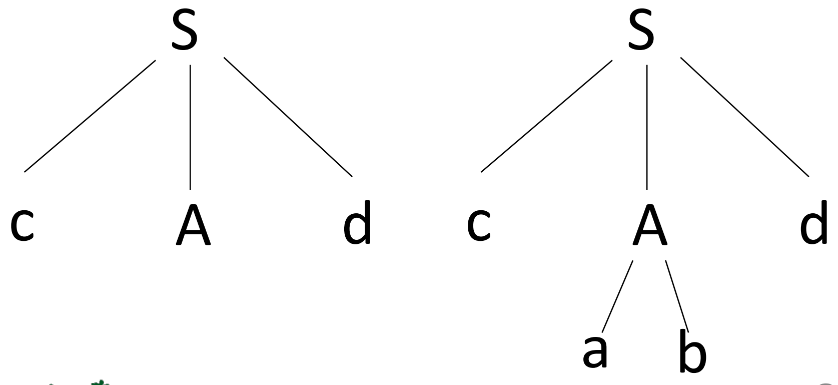


Example (cont.)

- Consider the grammar

  S → cAd A → ab | 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

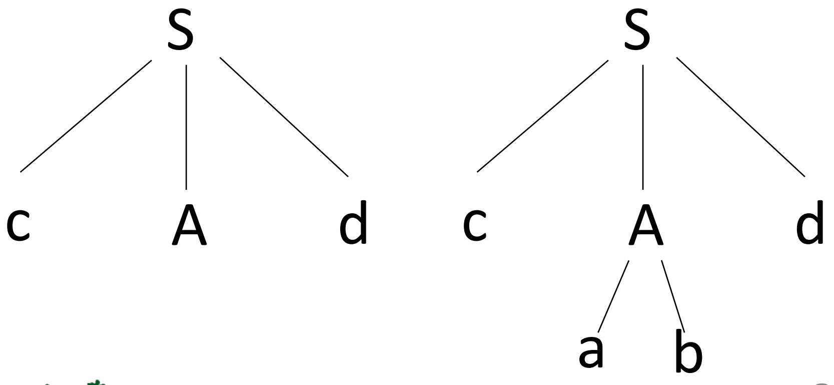


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | 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

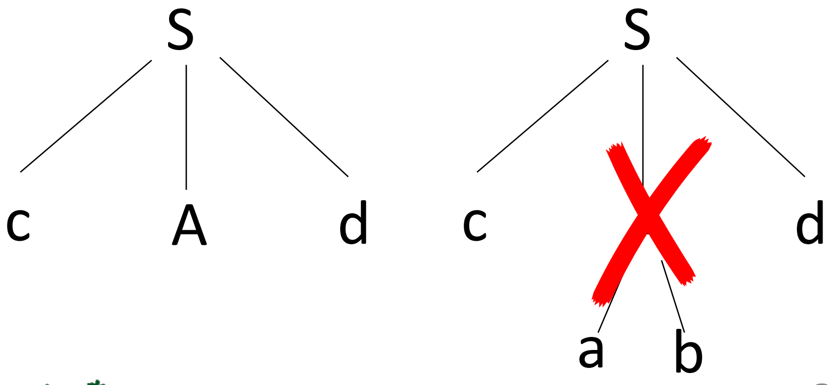


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | 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

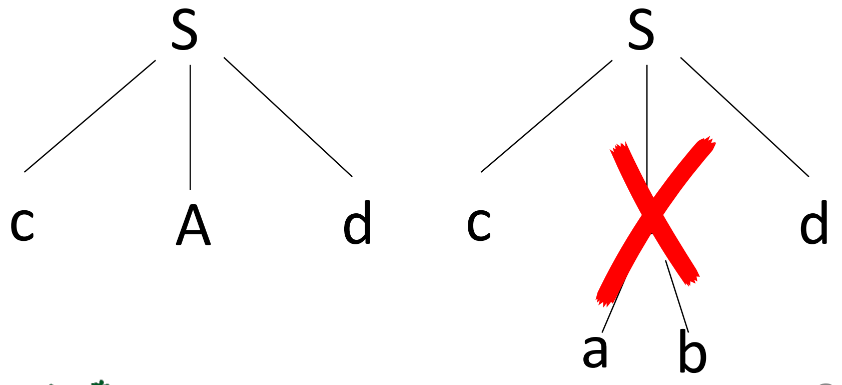


Example (cont.)

- Consider the grammar

  S → cAd   A → ab | 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

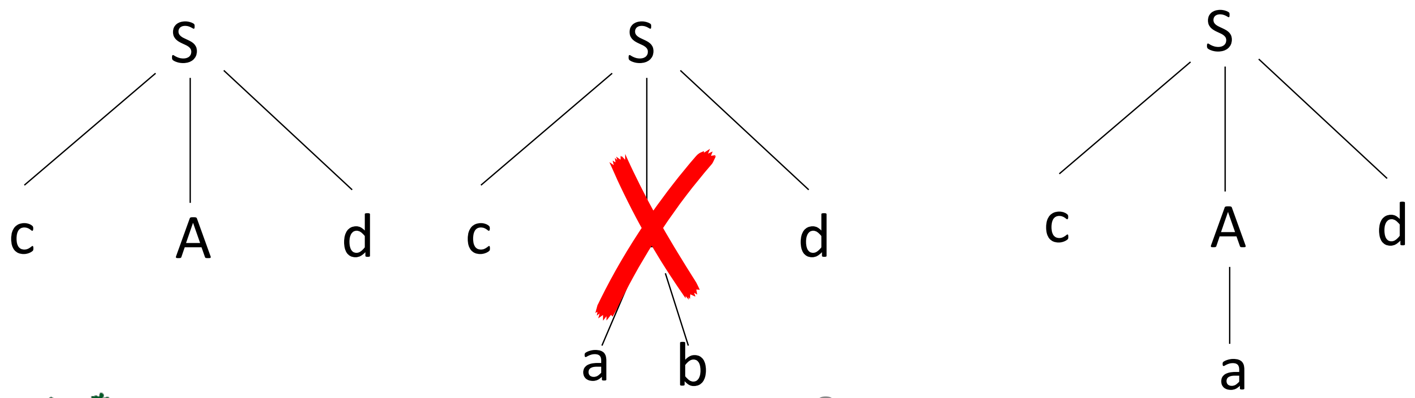


Example (cont.)

- Consider the grammar

  S → cAd A → ab | 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[左递归问题]

- 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

    A ⇒ A b ⇒ A b b ...
  - Sentential form can grow indefinitely w/o consuming input[句型无限增长而不消耗输入]
    - Non-terminal: expand, terminal: match
  - 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[等价]

Left Recursion[左递归]

- A grammar is left recursive if
  - It has a nonterminal *A* such that there is a derivation A ⇒+ Aα for some string α
- Recursion types [直接和间接左递归]
  - **Immediate left recursion**, where there is a production A → Aα
  - Non-immediate: left recursion involving derivation of 2+ steps

    S → Aa | b

    A → Sd | ε
  - S ⇒ Aa ⇒ Sda

- ☞ Algorithm to systematically eliminates left recursion from a grammar

Remove Left Recursion[消除左递归]

- Grammar: $A \rightarrow A\alpha \mid \beta$ ($\alpha \neq \beta$, β doesn’t start with A)

$$
\begin{aligned}
A &\Rightarrow A\alpha \\
&\Rightarrow A\alpha\alpha \\
&\quad \ldots \\
&\Rightarrow A\alpha\ldots\alpha\alpha \\
&\Rightarrow \beta\alpha\ldots\alpha\alpha
\end{aligned}
$$

- Rewrite to:

$A \rightarrow \beta A'$ // begins with β (A’ is a new non-terminal)

$A' \rightarrow \alpha A' \mid \varepsilon$ // A’ is to produce a sequence of α

$$
\begin{aligned}
&\Rightarrow \alpha\alpha A' \\
&\quad \ldots \\
&\Rightarrow \alpha\ldots\alpha A' \Rightarrow \alpha\ldots\alpha
\end{aligned}
$$

Remove Left Recursion[消除左递归]

- Grammar: A → Aα | β (α≠β, β doesn't start with A)

  A ⇒ Aα

  ⇒ Aαα

  …

  ⇒ Aα…αα

  ⇒ βα…αα

  r= βα*

- Rewrite to:

  A → βA' // begins with β (A' is a new non-terminal)

  A' → αA'|ε // A' is to produce a sequence of α

  ⇒ ααA'

  …

  ⇒ α…αA' ⇒ α…α

Remove Left Recursion (cont.)

- Grammar:

  $A \rightarrow A\alpha \mid \beta$

  to

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \varepsilon$

- $E \rightarrow E \underbrace{+\ T}_{\alpha} \mid \underbrace{T}_{\beta}$ $\Longrightarrow$ $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

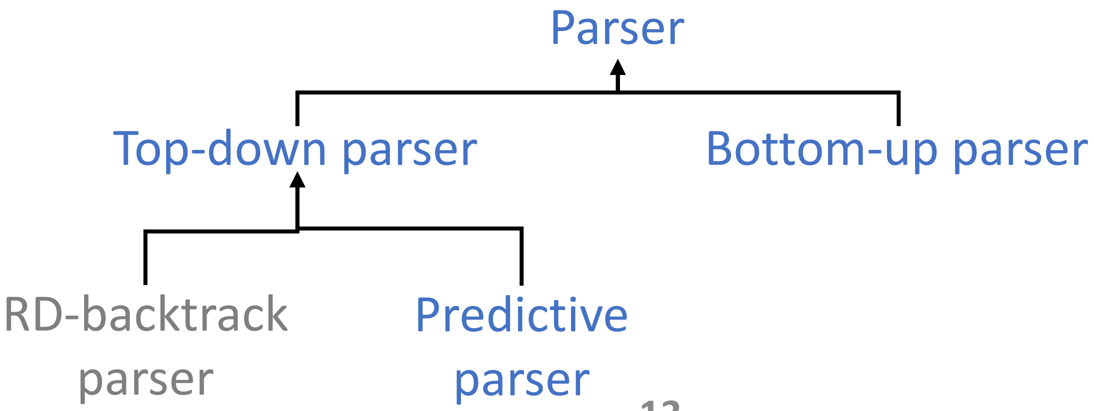
- $T \rightarrow T \underbrace{*\ F}_{\alpha} \mid \underbrace{F}_{\beta}$ $\Longrightarrow$ $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

- $F \rightarrow (E) \mid id$ $\Longrightarrow$ $F \rightarrow (E) \mid id$

Summary of RD-backtrack[小结]

- **RD-backtrack** is a simple and general parsing strategy
  - Left-recursion must be eliminated first
    - Can be eliminated automatically using some algorithm
  - L(Recursive_descent) ≡ L(CFG) ≡ 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



Predictive Parsers[预测分析]

- In recursive descent with backtracking[有回溯]:
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- A parser with no backtracking[无回溯]: **predict** correct next production given next input terminal(s) [以下面一些输入来预测]
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking[候选产生式首符号唯一]
  - If not unique, grammar cannot use predictive parsers[不唯一]

A→aBD | bBB

B→c | bce

D→d

parsing input “abced” requires no backtracking

Predictive Parsers[预测分析]

- In recursive descent with backtracking[有回溯]:
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- A parser with no backtracking[无回溯]: **predict** correct next production given next input terminal(s)? [以下面一些输入来预测]
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking[候选产生式首符号唯一]
  - If not unique, grammar cannot use predictive parsers[不唯一]

A→aBD | bBB

B→c | bce

D→d

parsing input “abced” requires no backtracking

Predictive Parsers[预测分析]

- In recursive descent with backtracking[有回溯]:
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- A parser with no backtracking[无回溯]: **predict** correct next production given next input terminal(s)? [以下面一些输入来预测]
  - If first terminal of every alternative production is **unique**, then parsing requires no backtracking[候选产生式首符号唯一]
  - If not unique, grammar cannot use predictive parsers[不唯一]

A→aBD | bBB

B→c | bce

D→d

parsing input “abced” requires no backtracking

?: 如果只往前看一个，那么next terminal其实就是current terminal，即要匹配的那个（注意backtrack是完全不看）

Predictive Parsers (cont.)

- A predictive parser chooses the production to apply solely on the basis of[选取产生式的依据]
  - Next input symbol(s)[下一输入符号/终结符]
  - Current nonterminal being processed[当前正处理的非终结符]
- Patterns in grammars that prevent predictive parsing[并非总是能预测分析]
  - **Common prefix**[共同前缀]:

    S → cAd   A → ab | a

    A→αβ | αγ

    Given input terminal(s) α, cannot choose between two rules
  - **Left recursion**[左递归]:

    A → Ab | a

    A→Aβ | α   从不匹配（一直展开） input: abbbb

    Lookahead symbol changes only when a terminal is matched

Rewrite Grammars for Prediction[改写]

- **Left factoring**[左公因子提取]: removes common left prefix
  - In previous example: A→αβ | αγ
  - can be changed to

    A → α A'

    A' → β | γ

    stmt → **if** *expr* **then** *stmt* **else** *stmt* | **if** *expr* **then** *stmt*

    ☞ *stmt* → **if** *expr* **then** *stmt S'*

    S' → **else** *stmt* | ε
  - After processing α, A' can can choose between β or γ

  (assuming β or γ do not start with α)

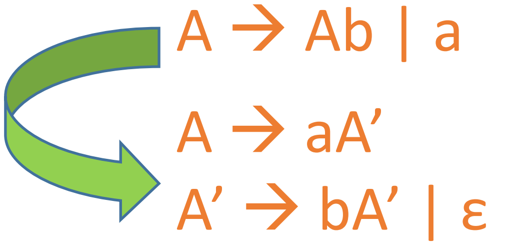
☞ 推迟选择，直到可区分

- **Left-recursion removal**[左递归消除]: same as recursive descent
  - In previous example: A→Aβ|α
  - can be changed to

    A → α A'

    A'→βA' | ε
  - After processing α, A' can can choose between β or ε

  (assuming β doesn't start with α or A' isn't followed by α)



A → Ab | a

A → aA'

A' → bA' | ε

逐步匹配 input: abbbb

LL(k) Parser / Grammar / Language

- **LL(k) Parser**
  - A predictive parser that uses *k* lookahead tokens
  - **L**: scans the input from left to right[从左往右]
  - **L**: produces a leftmost derivation[生成最左推导]
  - **k**: using *k* input symbols of lookahead at each step to decide[向前看k个符号]
- **LL(k) Grammar**
  - A grammar that can be parsed using an LL(k) parser
  - LL(k) ⊂ CFG
    - Some CFGs are not LL(k): common prefix or left-recursion
- **LL(k) Language**
  - A language that can be expressed as an LL(k) grammar
- Many languages are LL(k) ...
  - In fact many are **LL(1)**!

LL(k) Parser Implementation[实现]

- Implemented in a recursive or non-recursive fashion[递归/非递归]
  - Recursive: recursive descent (recursive function calls, implicit stack)
  - Non-recursive: explicit stack to keep track of recursion[栈]
- Recursive LL(1) parser for: A→B | C, B→b, C→c
  - Parser consists of small functions, one for each non-terminal

```
void A() {
   token = peekNext(); // lookahead token
   switch(token) {
      case ’b’: // ’B’ starts with ’b’
             B(); // call procedure B()
      case ’c’:  // ’C’ starts with ’c’
             C(); // call procedure C()
      default: // Reject
             return;
}
```

LL(k) Parser Implementation (cont.)

- Recursive LL(1) parser for: A→B | C, B→b, C→c

```
void A() {
    token = peekNext(); // lookahead token
    switch(token) {
        case ’b’: // ’B’ starts with ’b’
                B(); // call procedure B()
        case ’c’:  // ’C’ starts with ’c’
                C(); // call procedure C()
        default: // Reject
                return;
}
```

- Is there a way to express above code more concisely?[简洁]
  - Non-recursive LL(k) parsers use a **state transition table** (just like finite automata)[状态转换表]
  - Easier to automatically generate a non-recursive parser[自动化]