Top-Down Parsing

Dragon ch. 4.4

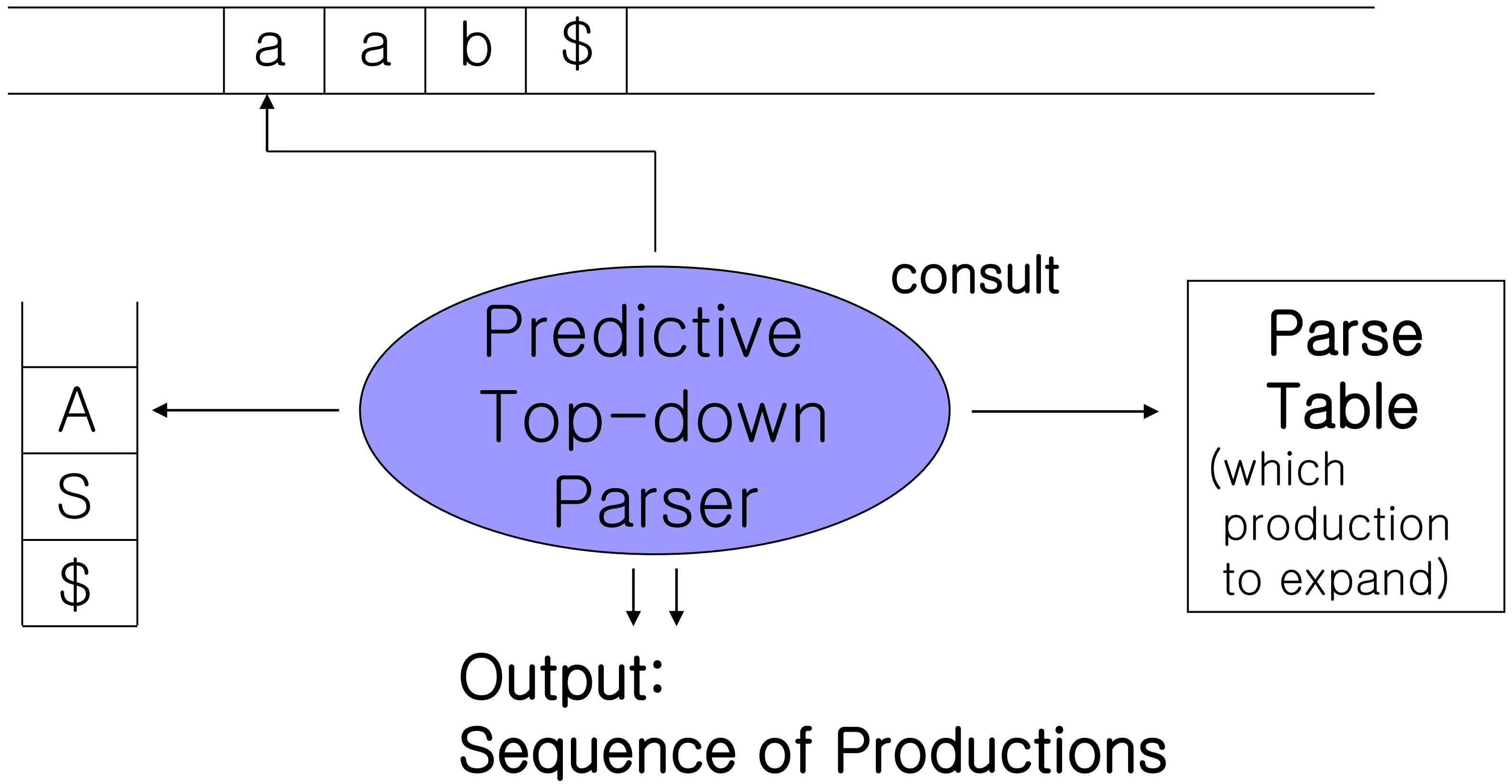
Recognizer and Parser

- A **recognizer** is a machine (system) that can accept a terminal string for some grammar and determine whether the string is in the language accepted by the grammar
- A **parser**, in addition, finds a derivation for the string
  - For grammar G and string **x**, find a derivation $S \Rightarrow^* \mathbf{x}$ if one exists
  - Construct a parse tree corresponding to this derivation
  - Input is read (scanned) from left to right
  - Two types of the parser: top-down vs. bottom-up

Top-down Parsing

- Top-down parsing expands a tree from the top (start symbol) using a stack
  - Put the start symbol on the stack top
  - Repeat
    - Expand a nonterminal on the stack top
    - Match stack tops with input terminal symbols
- Problem: which production to expand?
    - If there are multiple productions for a given nonterminal
  - One way: guess!

Structure of top-down parsing



Example of Parsing by Guessing

- P of an Example Grammar
  - S → AS|B, A → a, B→ b
  - Parsing process

input	stacktop	action
a a b $	S $	S -> AS
a a b $	A S $	A -> a
a a b $	a S $	Match a
a b $	S $	S -> AS
a b $	A S $	A -> a
a b $	a S $	Match a
b $	S $	S -> B
b $	B $	B -> b
b $	b $	Match b
$	$	End

- In reality, computers do not guess very well
  - So we use lookahead for correct expansion
- Before we do this, we must “condition” the grammar

Removal of Left Recursion

- Problem: infinite regression
  - $A \to A\alpha \mid \beta$, (the corresponding langauge is $\beta\alpha^*$)
  - Remove of immediate left recursion
    $A \to \beta B$, $B \to \alpha B \mid \varepsilon$
- More generally,
  - $A \to A\alpha_1$, $A \to A\alpha_2$
  - $A \to \beta_1$, $A \to \beta_2$
  - $A \to (\beta_1 \mid \beta_2)B$, $B \to (\alpha_1 \mid \alpha_2)B \mid \varepsilon$

Example of Removing Left Recursion

- Example of removing left immediate recursion

$$E \rightarrow E + T$$
$$E \rightarrow T$$

$$E \rightarrow TB$$
$$B \rightarrow +TB \mid e$$

  - Can remove all left recursions
  - Refer to Dragon Ch. 4.1 page 177

Left Factoring

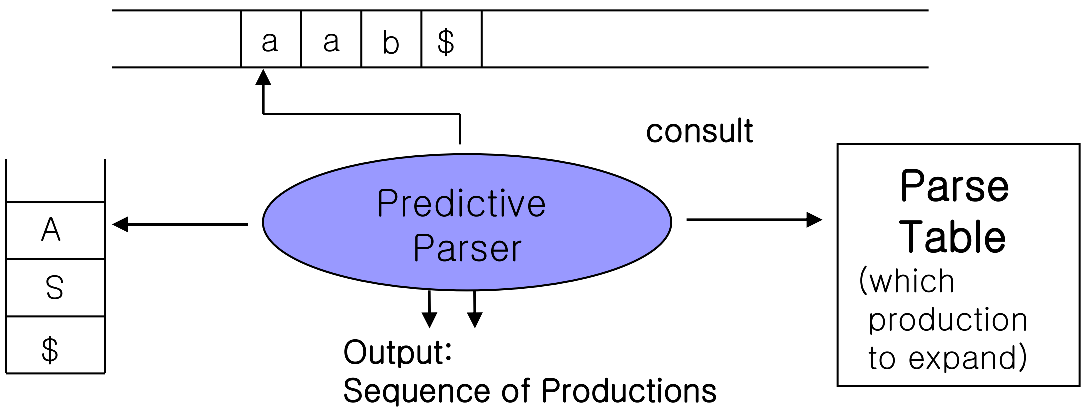
- Not have sufficient information right now
  - $A \rightarrow \alpha\beta | \alpha\gamma$
- Left factoring: turn two alternatives into one so that we match α first and hope it helps
  - $A \rightarrow \alpha B$, $B \rightarrow \beta | \gamma$
  - Example:

$$\frac{\begin{array}{l} E \rightarrow T + E \\ E \rightarrow T \end{array}}{\begin{array}{l} E \rightarrow TB \\ B \rightarrow +E \mid e \end{array}}$$

Predictive Top-Down Parsing

- Perform educated guess
  - Do not blindly guess productions that cannot even get the first symbol right
  - If the current input symbol is **a** and the top stack symbol is S, which of the two productions (S → **b**S, S → **a**) should be expanded?
- Two versions
  - Non-Recursive version with a stack
  - Recursive version: recursive descent parsing

Table-Driven Non-Recursive Parsing



- Input buffer: the string to be parsed followed by $
- Stack: a sequence of grammar symbols with $ at the bottom
  - Initially, the stack has the start symbol on top of $
- Parsing table: two dimensional array M[A,**a**], where A is a non-terminal and **a** is a terminal or $; it has productions
- Output: a sequence of productions expanded

Action of the Parser

When X is a symbol on top of the stack and **a** is the current input symbol

- If X = **a** = \$, a successful completion of parsing
- If X = **a** ≠ \$, pops X off the stack and advances the input pointer to the next input symbol
- If X is a nonterminal, consult M[X,**a**] which will be either an X-production or an error;
  - If M[X,a] = {X → UVW}, X on top of stack is replaced by WVU (with U on top) and print its production number
  - If [X,a] = error means a parsing error

An Example Grammar

Original grammar

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

After removing left recursion

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid \mathbf{id}$

An Example Parsing Table

	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → e	E’ → e
T	T → FT’			T → FT’		
T’		T’ → e	T’→ *F		T’ → e	T’ → e
F	F → id			F→ (E)		

How is **id + id * id** parsed?

How to Construct the Parse Table?

- For this, we use three functions
  - *Nullable()* : can it be a **null?**
    - Predicate, $V^* \rightarrow$ {true, false}
    - Telling if a string of nonterminals is nullable, i.e., can derive an empty string
  - *FNE()*: **first** but **no epsilon**
    - Terminals that can appear at the beginning of a derivation from a string of grammar symbols
  - *Follow()*: what can **follow** after a nonterminal?
    - Terminals (or $) that can appear after a nonterminal in some sentential form

Nullable()

- Nullable(α) = true if $\alpha \Rightarrow^* \varepsilon$
  = false, otherwise
  - Start with the obvious ones, e.g., $A \rightarrow \varepsilon$
  - Add new ones when $A \rightarrow \alpha$ and Nullable(α)
  - Keep going until there is no change
- More formally,
  - Nullable(ε) = true
  - Nullable($X_1X_2..X_n$) = true iff Nullable(X_i) $\forall i$
  - Nullable(A) = true if $A \rightarrow \alpha$ and Nullable(α)

FNE()

- Definition: $FNE(\alpha) = \{\mathbf{a} | \alpha \Rightarrow^* \mathbf{a}X\}$
- FNE() is computed as in Nullable()
  - $FNE(\mathbf{a}) = \{\mathbf{a}\}$
  - $FNE(X_1X_2...X_n) =$
    $\text{if}(!\text{Nullable}(X_1))$ then $FNE(X_1)$
    else $FNE(X_1) \cup FNE(X_2X_3...X_n)$
  - if $A \rightarrow \alpha$ then $FNE(A) \supseteq FNE(\alpha)$

FNE() Computation Example

- For our example grammar
  - $E \to TE'$
  - $E' \to +TE' \mid e$
  - $T \to FT'$
  - $T' \to *FT' \mid e$
  - $F \to (E) \mid id$
- We can compute FNE() as follows

Nullable(T) = false	FNE(E) = FNE(T) = {(,id}
	FNE(E') = {+}
Nullable(F) = false	FNE(T) = FNE(F) = {(,id}
	FNE(T') = {*}
	FNE(F) = {(, id}

First()

- The Dragon book uses First(), which is a combination of Nullable() and FNE()
  - If $\boldsymbol{\alpha}$ is nullable First($\boldsymbol{\alpha}$) = $\{\mathbf{a} \mid \boldsymbol{\alpha} \Rightarrow^* \mathbf{a}X\} \cup \{\boldsymbol{\varepsilon}\}$ else First($\boldsymbol{\alpha}$) = $\{\mathbf{a} \mid \boldsymbol{\alpha} \Rightarrow^* \mathbf{a}X\}$
- First() can be computed from Nullable() and FNE(), or directly (see Dragon book)

Follow()

- Follow(A)={**a**|S $\Rightarrow^*$ αA**a**β}, where **a** might be $
    - Follow() is needed if there is an ε−production
- To compute Follow(),
    - $ $\in$ Follow(S)
    - When A $\rightarrow$ αBβ,
      Follow(B) $\supseteq$ FNE(β)
    - When A $\rightarrow$ αBβ and Nullable(β),
      Follow(B) $\supseteq$ Follow(A)

Follow() Computation Example

- For our example grammar
  - $E \rightarrow TE'$
  - $E' \rightarrow +TE' \mid e$
  - $T \rightarrow FT'$
  - $T' \rightarrow *FT' \mid e$
  - $F \rightarrow (E) \mid id$

> - When $A \rightarrow \alpha B \beta$,
> $Follow(B) \supseteq FNE(\beta)$
> - When $A \rightarrow \alpha B \beta$ and $Nullable(\beta)$,
> $Follow(B) \supseteq Follow(A)$

- We can compute Follow() as follows

FNE(E) = FNE(T) = {(, id}	Follow(E) = {$,)}
FNE(E') = {+}	Follow(E') = {$,)}
FNE(T) = FNE(F) = {(, id}	Follow(T) = {+, $,)}
FNE(T') = {*}	Follow(T') = {+, $,)}
FNE(F) = {(, id}	Follow(F) = {*, +, $,)}

Predictive Parsing Table

- How to construct the parsing table
  - Mapping N x T → P
  - A → **α** ∈ M[A,**a**] for each **a** ∈ FNE(**α**Follow(A))
    - **a** ∈ FNE(**α**), or
    - Nullable(**α**) and **a** ∈ FOLLOW(A)
- Meaning of "Nullable(**α**) and **a** ∈ FOLLOW(A)"
  - Since the stack has (part of) a sentential form with A at the top, we can remove A (by expanding A→**α**) then try to match **a** with a symbol below A in the stack
    - Why? The symbol below A must be in Follow(A), so there is a chance that it can be **a** (※ or is this always guaranteed?)

Predictive Parsing Table

- For our example grammar
  - E → TE’
  - E’ → +TE’ | e
  - T → FT’
  - T’ → *FT’ | e
  - F → (E) | id
- The parsing table is as follows:

	FNE()	Follow()	id	+	*	(	)	$
E	(, id	$,)	E → TE’			E → TE’		
E’	+	$,)		E’ → +TE’			E’ → e	E’ → e
T	(, id	+, $,)	T → FT’			T → FT’		
T’	*	+, $,)		T’ → e	T’ → *F		T’ → e	T’ → e
F	(, id	*, +, $,)	F → id			F → (E)		

LL(1) Grammar

- Definition: a grammar G is LL(1) if there is at most one production for any entry in the table
  - So we can do top-down parsing with one lookahead

- LL(1) means left-to-right scan, performing leftmost derivation, with one symbol lookahead

LL(1) Conditions

- G is LL(1) iff whenever $A \to \alpha$ and $A \to \beta$ are distinct production of G, the following holds
  - α and β do not both derive strings beginning with **a** ($\in T$)
  - α and β do not both derive ε
  - if $\beta \Rightarrow^* \varepsilon$ then $FNE(\alpha) \cap Follow(A)$ is empty
- In other words, G is LL(1) if
  - if G is ε-free and unambiguous, $FNE(\alpha) \cap FNE(\beta) = \Phi$
  - If an ε-production is present, $FNE(\alpha Follow(A)) \cap FNE(\beta Follow(A)) = \Phi$

Testing for non-LL(1)ness

- In practice, for LL(1) testing, it is easiest to construct the parse table and check
- Some shortcuts to test if G is not LL(1)
  - G is left-recursive (e.g., $A \rightarrow A\alpha \mid \beta$)
  - Common left factors (e.g., $A \rightarrow \alpha\beta|\alpha\gamma$)
  - G is ambiguous (e.g., $S \rightarrow Aa \mid a,\ A \rightarrow \varepsilon$)

Non-LL(1) Grammar

- Consider the following grammar G1, which is not LL(1)
  $S \rightarrow Bbc$
  $B \rightarrow \varepsilon \mid b \mid c$
  - FNE(B) = FNE(S) = {b,c},
  - FOLLOW(S)={$}, FOLLOW(B)={b}

	FNE	FOLLOW	b	c
S	{b,c}	{$}	$S \rightarrow Bbc$	$S \rightarrow Bbc$
B	{b,c}	{b}	$B \rightarrow e$ $B \rightarrow b$	$B \rightarrow c$

  - Since FNE(εFOLLOW(B))=FNE(bFOLLOW(B))={b}
- We want consider a larger class of LL parsing, **LL(k),** which look-ahead more symbols

LL(K) Parsing

- Begin by extending the definition of FNE() and FOLLOW()
  - Definitions of $FNE_k()$ and $FOLLOW_k()$

$$FNE_k(\boldsymbol{\alpha}) = \{w \mid (|w| < k \text{ and } \boldsymbol{\alpha} \Rightarrow^* w) \text{ or } (|w| = k \text{ and } \boldsymbol{\alpha} \Rightarrow^* wx \text{ for some } x\}$$

$$FOLLOW_k(A) = \{w \mid S \Rightarrow^* \boldsymbol{\alpha} A \boldsymbol{\beta} \text{ and } w \in FNE_k(\boldsymbol{\beta})\}$$

  - As with FOLLOW(), we will implicitly augment the grammar with $S' \rightarrow S\$^k$ so that out definitions are: $FOLLOW_k(a) = \{w \mid S \Rightarrow^* \boldsymbol{\alpha} A \boldsymbol{\beta} \text{ and } \boldsymbol{\omega} \in FNEk(\boldsymbol{\beta}\$^k)\}$

LL(K) Parsing Definition

- G is **LL(k)** for some fixed k if, whenever there are two leftmost derivations,

  $S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx$, and
  $S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy$ and $\beta \neq \gamma$,
  then $FNE_k(x) \neq FNE_k(y)$

Strong-LL(K) Parsing

- Simplest way to implementing LL(k) parsing table
  - Insert $A \rightarrow \boldsymbol{\alpha} \in M[A, x]$ for each $x \in FNE_k(\boldsymbol{\alpha} Follow_k(A))$
  - A grammar G is **strong-LL(k)** if there is at most one production for any entry in the table
    - If $FNE_k(\boldsymbol{\beta} FOLLOW_k(A)) \cap FNE_k(\boldsymbol{\gamma} FOLLOW_k(A)) = \boldsymbol{\Phi}$ for all $A \rightarrow \boldsymbol{\beta}$ and $A \rightarrow \boldsymbol{\gamma}$ in G

Non-LL(1), but Strong-LL(2) Grammar

- Consider our non-LL(1) grammar G1 again

  $S \rightarrow B\mathbf{bc}$

  $B \rightarrow \mathbf{\epsilon} \mid \mathbf{b} \mid \mathbf{c}$

  - $FNE_2(BbcFOLLOW_2(S)) = \{bc,bb,cb\}$
  - $FNE_2(\varepsilon FOLLOW_2(B)) = \{bc\}$, $FNE_2(bFOLLOW_2(B)) = \{bb\}$, $FNE_2(cFOLLOW_2(B)) = \{cb\}$

	bc	bb	cb
S	$S \rightarrow Bbc$	$S \rightarrow Bbc$	$S \rightarrow Bbc$
B	$B \rightarrow e$	$B \rightarrow b$	$B \rightarrow c$

  - So, G1 is Strong-LL(2) even though it is not LL(1)

LL(2) but Non-Strong LL(2) Grammar

- Consider the following grammar G2

  $S \rightarrow B\mathbf{bc} \mid a B\mathbf{cb}$

  $B \rightarrow \mathbf{\epsilon} \mid \mathbf{b} \mid \mathbf{c}$

  - $FNE_2()$ and $FOLLOW_2()$ functions:
    - $FNE_2(S)$ = {ab, ac, bb, bc, cb}, $FNE_2(B)$ = {b,c}
    - $FOLLOW_2(S)$ = {$$}, $FOLLOW_2(B)$ = {bc,cb}
      - $FNE_2(\varepsilon FOLLOW2(B))$ = {bc,cb}
      - $FNE_2(bFOLLOW2(B))$ = {bb,bc}, so not strong-LL(2)
  - But isn't G LL(2), either?
    - Check with the LL(k) definition
      - $S \Rightarrow B\mathbf{bc} \Rightarrow$
      - $S \Rightarrow \mathbf{a}B\mathbf{cb} \Rightarrow$

Modified Grammar G2'

- G2 is indeed LL(2), then what's wrong with strong-LL(2) algorithm? Why can't it generate a LL(2) parsing table?
  - Because of Follow(), which does not always tell the truth
- Let us rewrite G2 with two new nonterminals, B_{bc} and B_{cb}, to keep track of local lookahead (context) information
  - $S \rightarrow B_{bc}bc \mid aB_{cb}cb$
  - $B_{bc} \rightarrow \varepsilon \mid b \mid c$
  - $B_{cb} \rightarrow \varepsilon \mid b \mid c$
- Now, in place of $FNE_2(\beta FOLLOW_2(B))$ to control putting $B \rightarrow \beta$ into table, use $FNE_2(\beta R)$ to control $B_R \rightarrow \beta$, where R is local lookahead

- For $S \rightarrow B_{bc}bc$, $FNE_2(B_{bc}bc\$\$) = \{bc,bb,cb\}$
- For $S \rightarrow aB_{cb}cb$, $FNE_2(aB_{cb}cb\$\$) = \{ac,ab\}$
- For $B_{bc} \rightarrow \varepsilon$, $FNE_2(\varepsilon\{bc\}) = \{bc\}$
- For $B_{bc} \rightarrow b$, $FNE_2(b\{bc\}) = \{bb\}$
- For $B_{bc} \rightarrow c$, $FNE_2(c\{bc\}) = \{cb\}$
- For $B_{cb} \rightarrow \varepsilon$, $FNE_2(\varepsilon\{cb\}) = \{cb\}$
- For $B_{cb} \rightarrow b$, $FNE_2(b\{cb\}) = \{bc\}$
- For $B_{cb} \rightarrow c$, $FNE_2(c\{cb\}) = \{cc\}$

Corresponding LL(2) Table: G2' is strong-LL(2)

	bc	bb	cb	ab	ac	cc
S	S->B_{bc}bc	S->B_{bc}bc	S->B_{bc}bc	S->aB_{cb}cb	S->aB_{cb}cb	
B_{bc}	B_{bc}-> e	B_{bc}-> b	B_{bc}-> c			
B_{cb}	B_{cb}-> b		B_{cb}-> e			B_{cb}-> c

LL(k) vs. Strong-LL(k)

- LL(k) definition says
  - $\omega A\alpha \Rightarrow \omega\beta\alpha$, $\omega A\alpha \Rightarrow \omega\gamma\alpha$
    $FNE_k(\beta\alpha) \cap FNE_k(\gamma\alpha) = \Phi$
  - $xA\delta \Rightarrow x\beta\delta$, $xA\delta \Rightarrow x\gamma\delta$
    $FNE_k(\beta\delta) \cap FNE_k(\gamma\delta) = \Phi$
- Strong-LL(k) definition adds additional constraint
  - $FNE_k(\beta\alpha) \cap FNE_k(\gamma\delta) = \Phi$
  - $FNE_k(\beta\delta) \cap FNE_k(\gamma\alpha) = \Phi$
  - Why? Because it relies on Follow(A) to get the context information, which always includes both α and δ

LL(1) = Strong LL(1) ?

- One question:
  - We saw an example grammar that is LL(2), yet not strong-LL(2)
  - Then, are there any example grammars that are LL(1), yet not Strong-LL(1)?
- The issue is the granularity of the lookahead
- The lookahead of LL(2) is finer than LL(1) since it look aheads more
- A nice exam question

Recursive-Descent Parsing

- Instead of stack, use recursive procedures
  - Sequence of production calls implicitly define parse tree
  - Given a parse table M[A,a], it is easy to write one

```
extern token lookahead;
void match(token tok) {
    if (lookahead != tok) error();
    else lookahead = get_next_token();
}
void E() {
    switch (lookahead) {
        case 'id':
        case '(' : T(); Ep(); break;
        default : error();
    }
}
```

```
void Ep() {
    switch (lookahead) {
        case '+' : match('+'); T(); Ep(); break;
        case ')' :
        case '$': break;
        default : error();
    }
}
...
main() {
    lookahead = get_next_token();
    E();
}
```

LL(1) Summary

- LL(1) parsing
  - Stack, lookahead, parsing table
  - Parsing table construction
    - Nullable(), FNE(), Follow()
- LL(1) grammar
  - Actually represent limited class of languages
    - i.e., many programming languages are not LL(1)
  - So, consider a larger class: LR bottom-up parsing