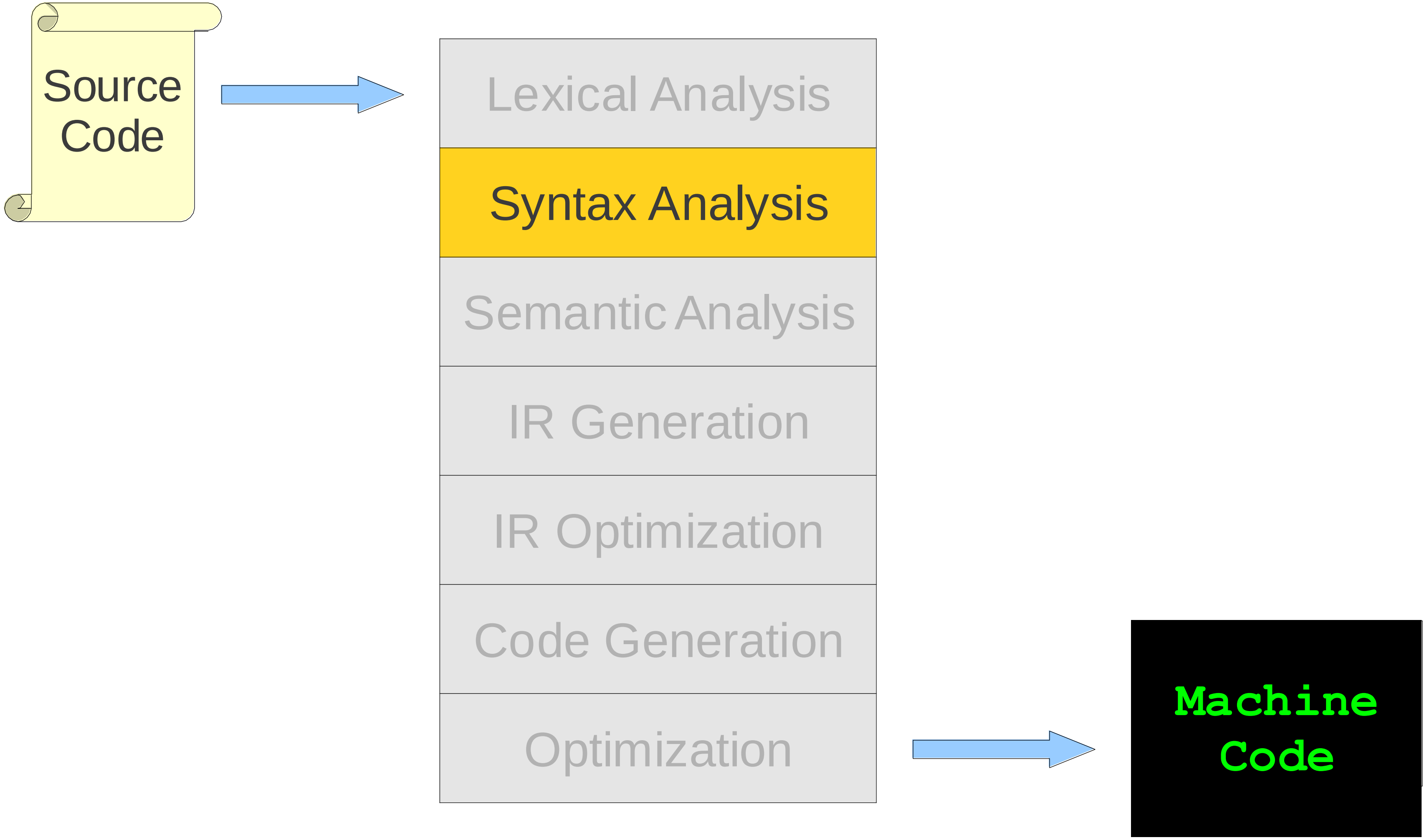
Compiler
– 3-4. Top-down Parsing –

JIEUNG KIM

jieungkim@yonsei.ac.kr

YONSEI UNIVERSITY

Where are we?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Outlines

- Role of the syntax analysis (parser)
- Context free grammar
- Push down automata
- Top-down parsing
- Bottom-up parsing
- Simple LR
- More powerful LR parsers and other issues in parsers
- Syntactic error handler
- Parser generator

Outlines

- **Top-down parsing**
  - Parsing overview
  - Top-down parser concept
  - Recursive decent parser
  - LL(1) grammar
  - LL(1) parser

Parsing overview

Parsing overview

- **Parsing (recall our memory)**
  - Discovering the derivation of a string: **If one exists.**
  - The Complexity of parsing
    - Parsers that work for any unambiguous grammar are complex and inefficient
      - $O(n^3)$, where n is the length of the input
      - E.g., CYK algorithm (Look at it if you are interested in NLP or automata theory)
    - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time
      - $O(n)$, where n is the length of the input
  - Two major approaches
    - Top-down parsing
    - Bottom-up parsing

YONSEI UNIVERSITY

Parsing overview

- Top-down parsing and bottom-up parsing

```
Sentence        ::= Subject Verb Object .
Subject         ::= I | a Noun | the Noun
Object          ::= me | a Noun | the Noun
Noun            ::= cat | mat | rat
Verb            ::= like | is | see | sees
```

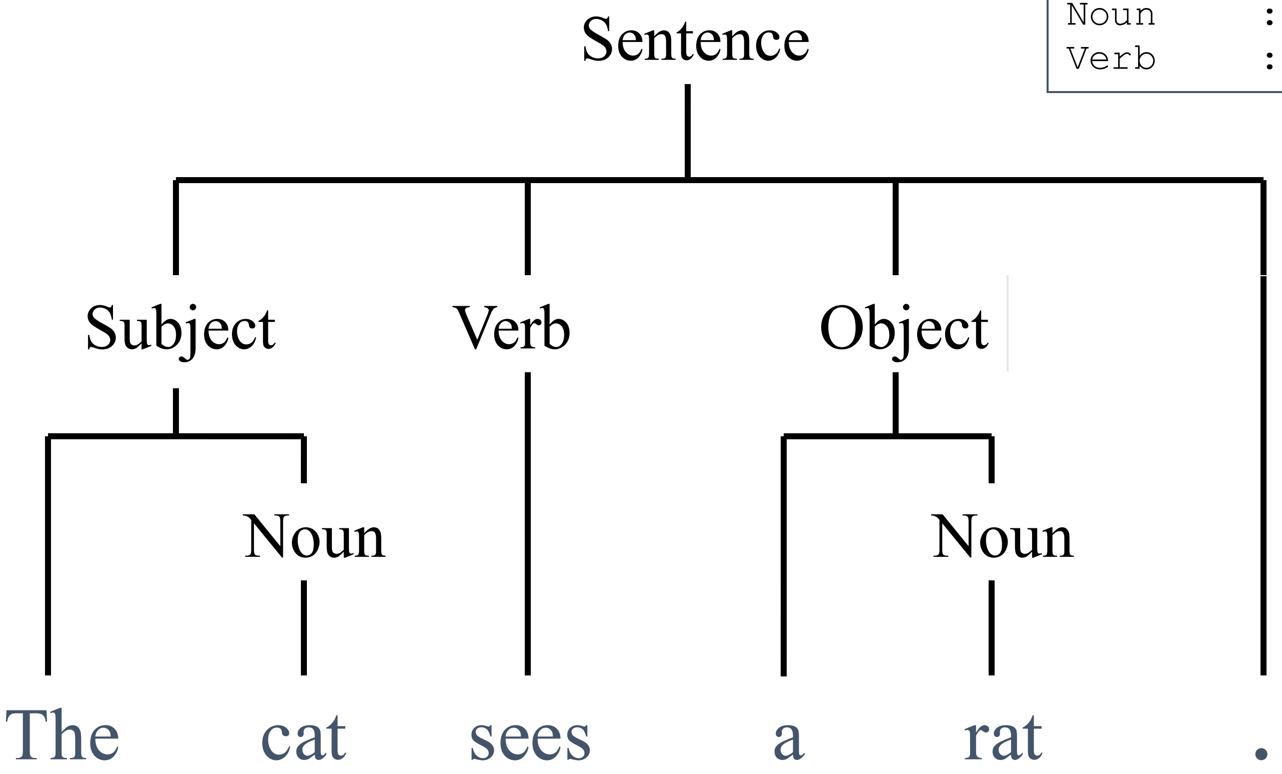
The cat sees the rat.
The rat sees me.
I like a cat.

The rat like me.
I see the rat.
I sees a rat.

Parsing overview

- **Top-down parsing**
  - The parse tree is constructed starting at the top.

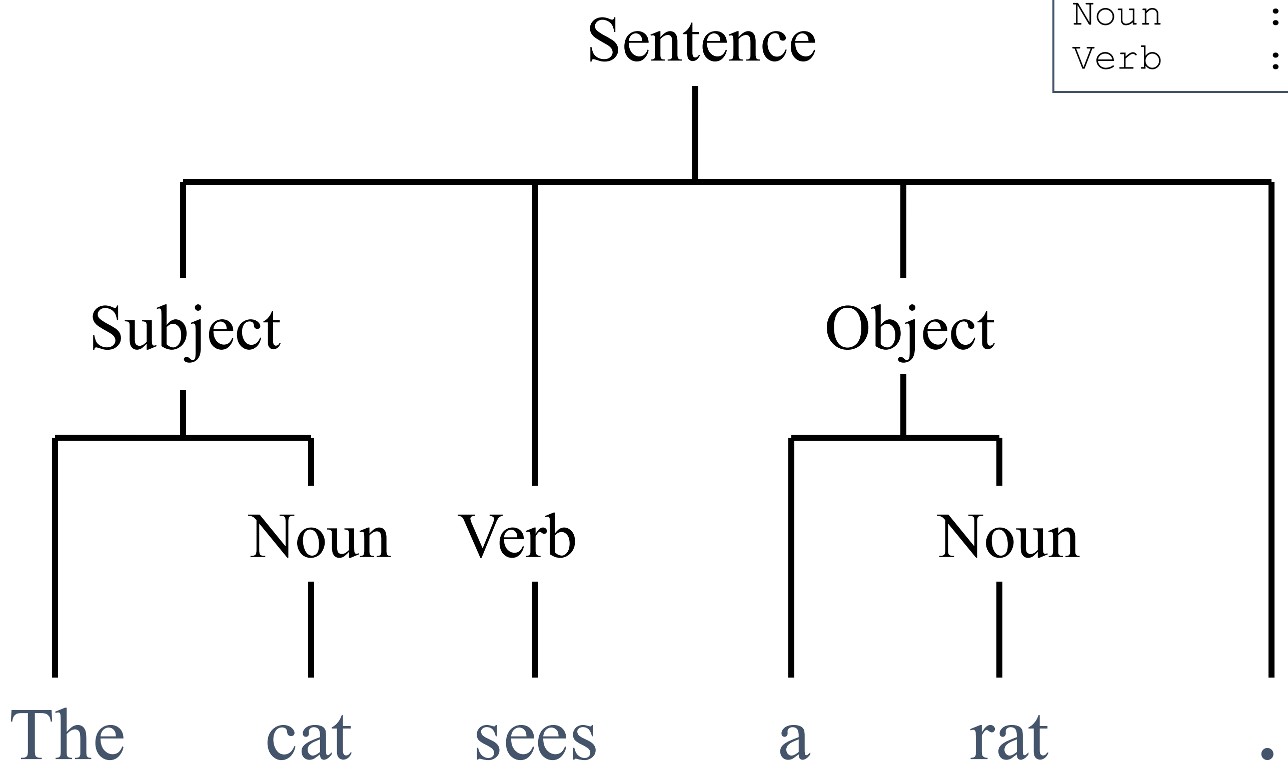
```
Sentence ::= Subject Verb Object .
Subject  ::= I | a Noun | the Noun
Object   ::= me | a Noun | the Noun
Noun     ::= cat | mat | rat
Verb     ::= like | is | see | sees
```



Parsing overview

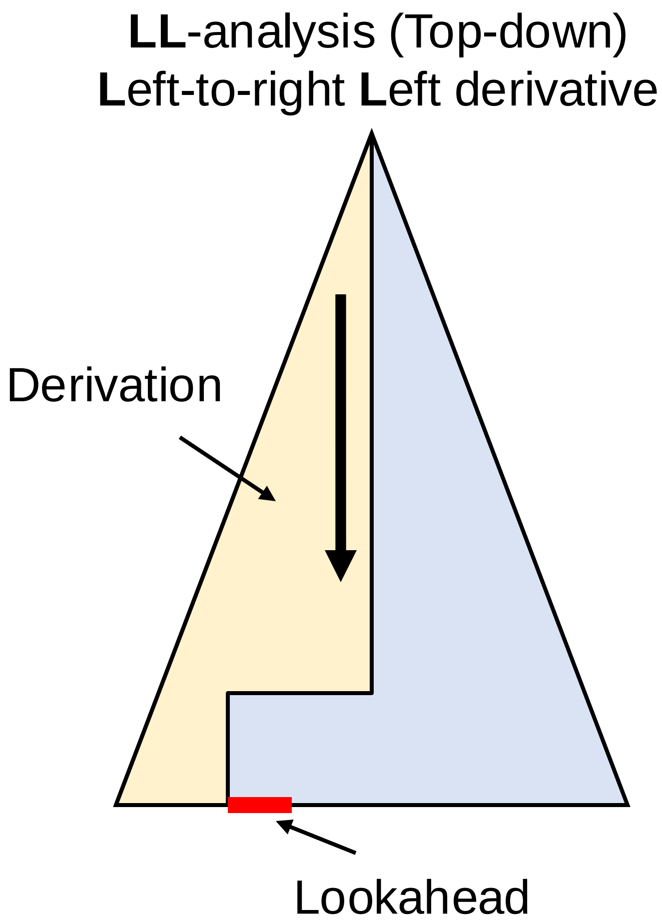
- Bottom-up parsing
  - The parse tree "grows" from the bottom up to the top.

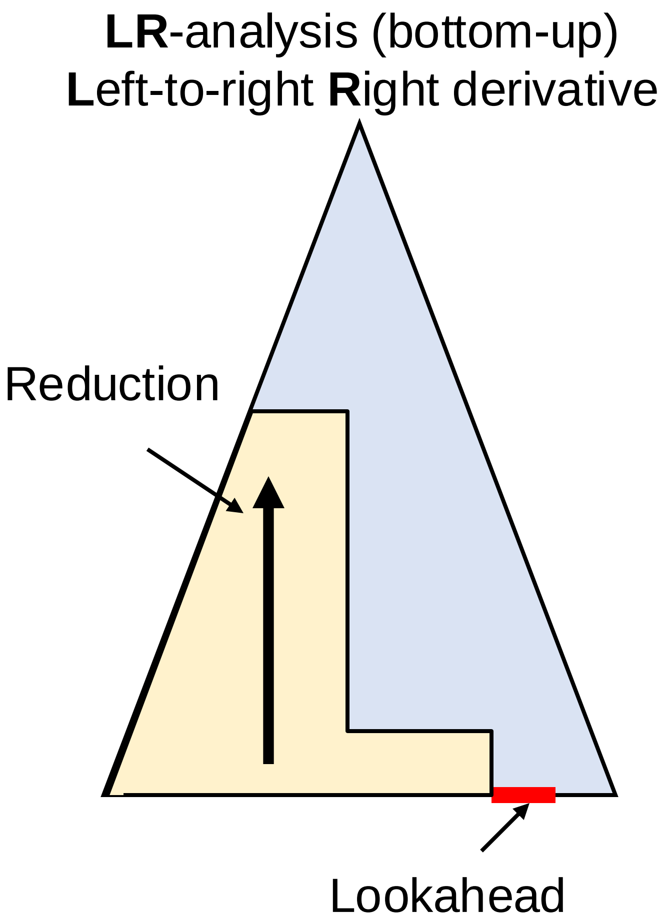
```
Sentence ::= Subject Verb Object .
Subject  ::= I | a Noun | the Noun
Object   ::= me | a Noun | the Noun
Noun     ::= cat | mat | rat
Verb     ::= like | is | see | sees
```



Parsing overview

- Top−down and bottom−up parsings





Top-down parser concept

Top-down parser concept

- **Top-down parsers**
  - A top-down parser starts with the root of the parse tree
  - Repeatedly pick a non-terminal and expand
  - Treat parsing as a graph search.
  - Success when expanded tree matches input
  - How to:
    - **Situation**: have completed part of a derivation

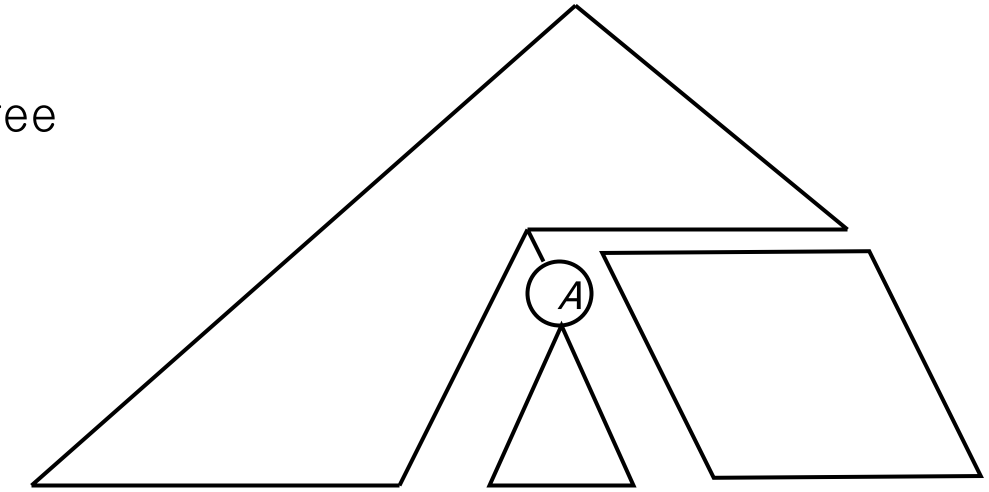
      $S \Rightarrow^* wA\alpha$

    - **Basic Step**: Pick some production that will properly expand to match the input

      $A ::= \beta_1\beta_2\cdots\beta_n$

      - The key is picking the right production, so the parser wants this to be deterministic

➔ **Why does determinism matter?**

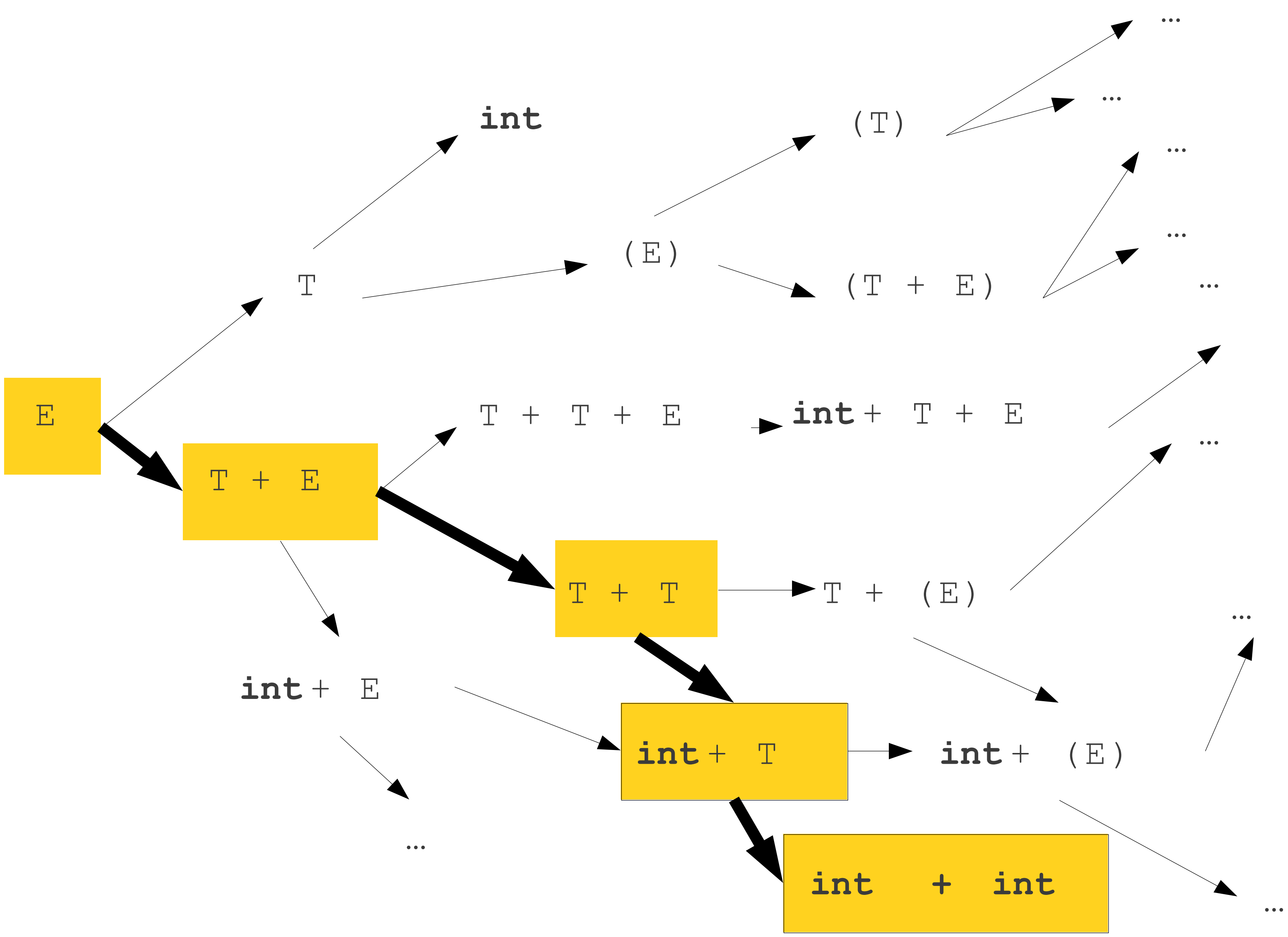


$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$



Top-down parser concept

- **Breadth-first Search & brute-force approach**
  - Maintain a worklist of sentential forms, initially just the start symbol S.
  - While the worklist isn't empty:
    - Remove an element from the worklist.
    - If it matches the target string, you're done.
    - Otherwise, for each possible string that can be derived in one step, add that string to the worklist.
  - Can recover a parse tree by tracking what productions we applied at each step.

Top-down parser concept

- Brute-force approach

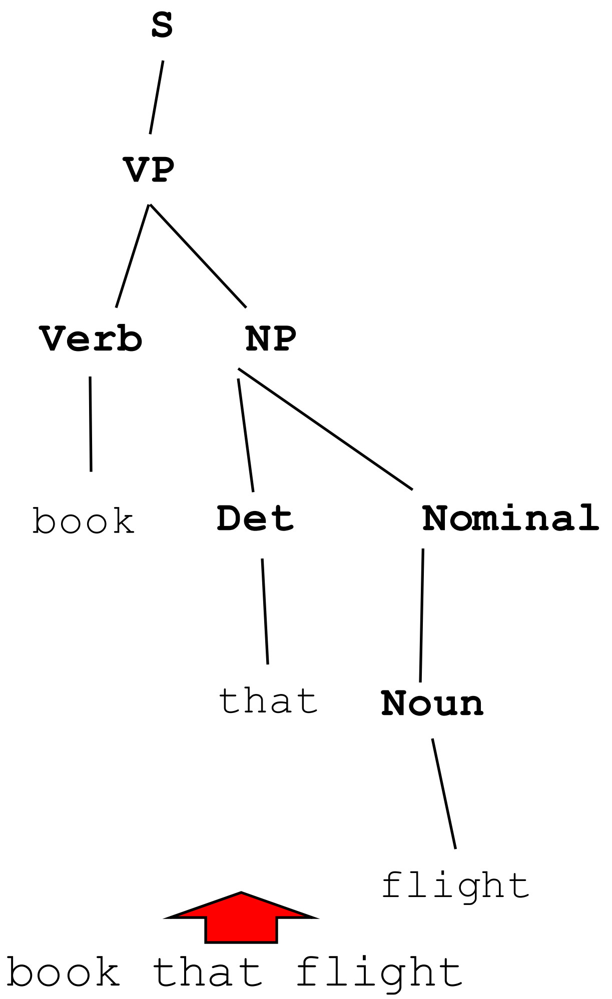
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```




```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

Top-down parser concept

- Brute-force approach

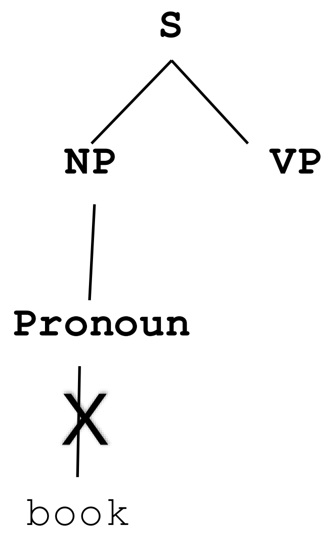
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```




```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

book that flight

Top-down parser concept

- Brute-force approach

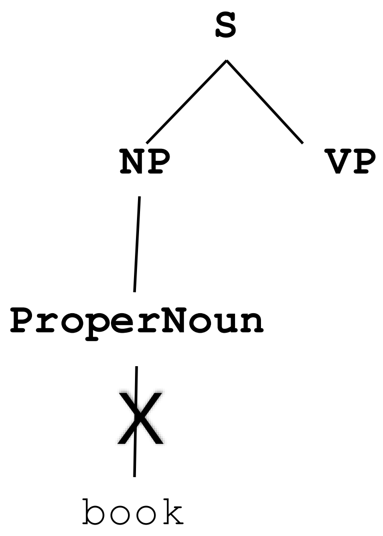
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```




```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

book that flight

Top-down parser concept

- Brute-force approach

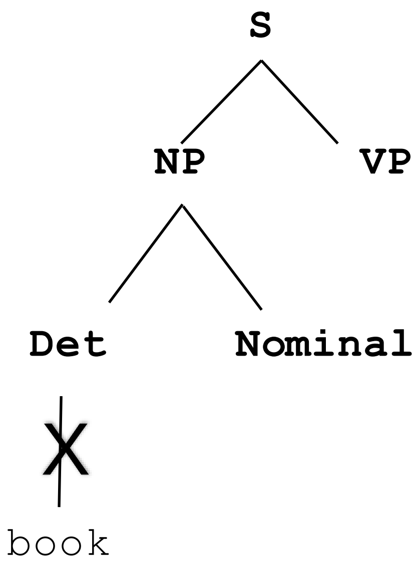
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```




```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

book that flight

Top-down parser concept

- Brute-force approach

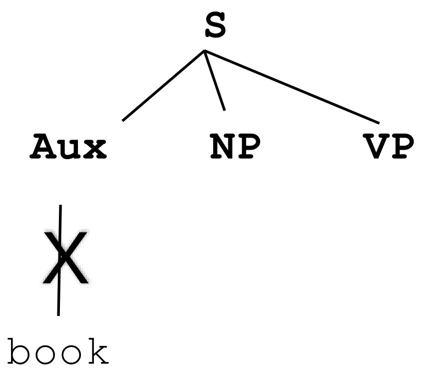
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```




```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

Top-down parser concept

- Brute-force approach

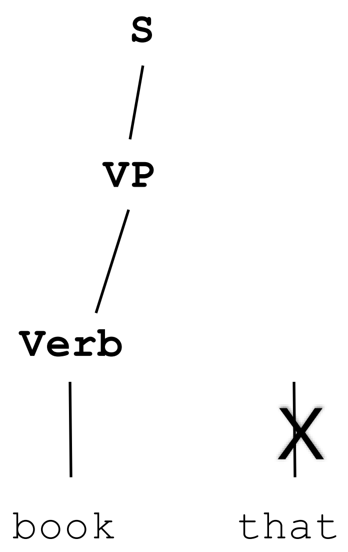
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```




```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

book that flight

Top-down parser concept

- Brute-force approach

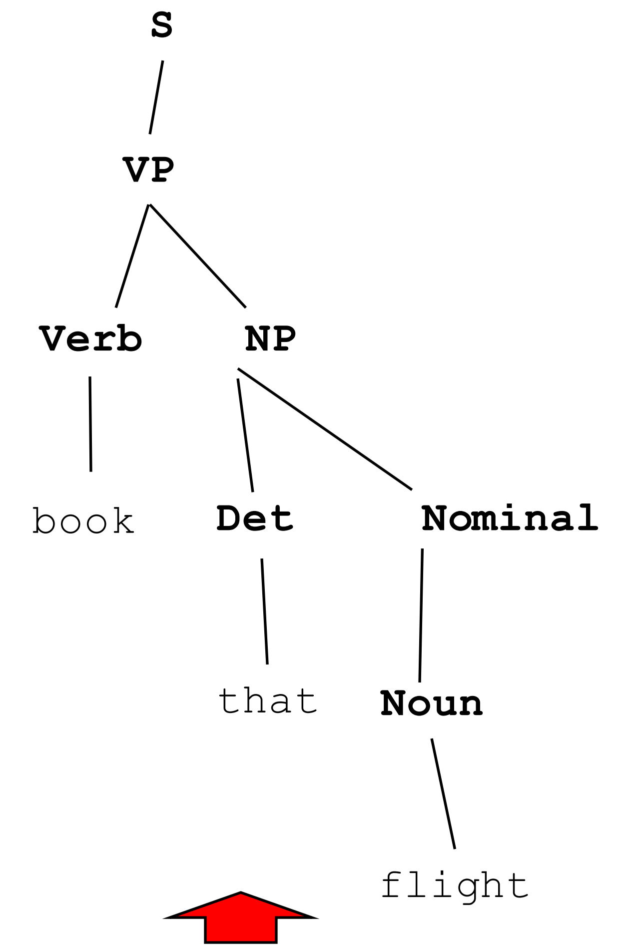
```
S ::= NP VP
S ::= Aux NP VP
S ::= VP

NP ::= Pronoun
NP ::= Proper-Noun
NP ::= Det Nominal

Nominal ::= Noun
Nominal ::= Nominal Noun
Nominal ::= Nominal PP

VP ::= Verb
VP ::= Verb NP
VP ::= VP PP

PP ::= Prep NP
```



book that flight

```
Det ::= the | a | that | this
Noun ::= book | flight | meal
Verb ::= book | prefer
Pronoun ::= I | he | she | me
Aux ::= does
….
```

Top-down parser concept

- Brute-force approach is slow
  - Enormous time and memory usage
  - Lots of wasted effort:
    - Generates a lot of sentential forms that couldn't possibly match.
    - But in general, extremely hard to tell whether a sentential form can match
    - that's the job of parsing!
  - High branching factor:
    - Each sentential form can expand in (potentially) many ways for each nonterminal it contains.

Top-down parser concept

- Reducing wasted effort
  - Suppose we're trying to match a string γ.
  - Suppose we have a sentential form $\tau = \alpha\omega$, where α is a string of terminals and ω is a string of terminals and nonterminals.
  - If α isn't a prefix of γ, then no string derived from τ can ever match γ.
  - If we can find a way to try to get a prefix of terminals at the front of our sentential forms, then we can start pruning out impossible options
- Reducing the branching factor
  - If a string has many nonterminals in it, the branching factor can be high.
  - Sum of the number of productions of each nonterminal involved.
  - If we can restrict which productions we apply, we can keep the branching factor lower.

YONSEI UNIVERSITY

Top-down parser concept

- Predictive parsing
  - If we are located at some non-terminal `A`, and there are two or more possible productions

    `A ::= α | β`

    we want to make the correct choice by looking at just the next input symbol
    - We call the next input symbol, that the parse is looking at, *lookahead symbol*
  - If we can do this, we can build a ***predictive parser*** that can perform a top-down parse **without backtracking**
  - Programming language grammars are often suitable for predictive parsing
  - Typical example

```
stmt ::=    id = exp ; | return exp ; |
            if ( exp ) stmt  | while ( exp ) stmt
```

    - If the first part of the unparsed input begins with the tokens

      IF LPAREN ID(x) …

      we should expand `stmt` to an if-statement

Top-down parser concept

- **The most common top-down parsing algorithms**
  - Recursive descent – a straightforward coded implementation
  - LL parsers – table driven implementation

Recursive descent parser

Recursive descent parser

- **Recursive descent – a coded implementation**
  - An advantage of top-down parsing is that it is easy to implement by hand
  - An early implementation of top-down parsing was recursive descent
  - **Key ideas**
    - A parser was organized as **a set of parsing procedures, one for each non-terminal**
    - **Each parsing procedure** was **responsible for parsing a sequence of tokens derivable from its non-terminal**
  - Called recursive descent because the parsing procedures were typically recursive, and they descended down the input's parse tree
  - Example:
    - A parsing procedure, A, when called, would call the scanner and match a token sequence derivable from A
    - Starting with the start symbol's parsing procedure, we would then match the entire input, which must be derivable from the start symbol

Recursive descent parser

- Grammar

```
stmt ::= id = exp ;
       | return exp ;
       | if ( exp ) stmt
       | while ( exp ) stmt
```

- Method for this grammar rule

```
// parse stmt ::= id=exp; | …
void stmt( ) {
  switch(nextToken) {
    RETURN: parseReturnStmt(); break;
    IF: parseIfStmt(); break;
    WHILE: parseWhileStmt(); break;
    ID: parseAssignStmt(); break;
  }
}
```

Recursive descent parser

- Grammar

```
stmt ::= id = exp ;
      | return exp ;
      | if ( exp ) stmt
      | while ( exp ) stmt
```

- Method for this grammar rule

```
// parse while (exp) stmt
void whileStmt() {
  // skip "while ("
  getNextToken();
  getNextToken();

  // parse condition
  parseExp();

  // skip ")"
  getNextToken();

  // parse stmt
  parseStmt();
}
```

Recursive descent parser

- Grammar

```
stmt ::= id = exp ;
       | return exp ;
       | if ( exp ) stmt
       | while ( exp ) stmt
```

- Method for this grammar rule

```
// parse return exp ;
void returnStmt() {
  // skip “return”
  getNextToken();

  // parse expression
  parseExp();

  // skip “;”
  getNextToken();
}
```

Recursive descent parser

- Algorithm to convert BNF into a RD parser
  - The conversion of a BNF specification into a program for a recursive descent parser is so "mechanical" that it can easily be automated
  - We can describe the algorithm by a set of mechanical rewrite rules

```
N ::= X
```

```
void parseN() {
   parse X
}
```

parse t	where `t` is a terminal

```
accept(t);
```

parse N	where `N` is a non-terminal

```
parseN();
```

parse ε

```
// a dummy statement
```

parse XY

parse X
parse Y

```
parse X*
```

Condition: *the starters of x* must be disjoint from the set of tokens that can immediately follow *X* *

```
while (getCurToken() in starters(X)) {
   parse X
}
```

```
parse X|Y
```

Condition: *the starters of x* and *the starters of y* must be disjoint sets.

```
switch (getCurToken()) {
   case 1) starters(X):
     parse X
     break;
   case 2) starters(Y):
     parse Y
     break;
   default: syntax error
}
```

YONSEI UNIVERSITY

Recursive descent parser

- Problems during parsing

```
C ::= SC | C; SC
SC ::= IDENT := Expr /* assignment */
     | IDENT (Expr)  /* inner expr */
     | ...
```

```
Switch (getCurToken()) {
  case 1)starters(SC)
    parseSC();
  case 2)starters(C) `
    parseC();
    accept(SEMICOLON);
    parseSC();
  default: syntax error
  }
}
```

```
switch (getCurToken()) {
  case 1) IDENT:
    parseAssignment()
  case 2) IDENT:
    parseInnerExpr()
   ...other cases...
  default: syntax error
}
```

wrong: overlapping cases

YONSEI UNIVERSITY

LL(1) grammar

LL(1) grammar

- Possible problems in LL parsers
  - Ambiguous – we talked about it a lot, so let's ignore it at this moment
  - Left recursion
  - Common prefixes on the right-hand side of productions

Left recursion (infinite looping problem)

```
C ::= SC | C; SC
SC ::= IDENT := Expr /* assignment */
     | IDENT (Expr)  /* inner expr */
     | ...
```

Common prefixes on the right-hand side of productions

LL(1) grammar

- **Left recursion**
  - A grammar is left-recursive if it has a non-terminal A, such that there is a derivation

    $$A \Rightarrow A\alpha, \quad \text{for some } \alpha$$

  - Top-Down parsing can't reconcile this type of grammar, **since it could consistently make choice which wouldn't allow termination**
  - So, we must convert our left-recursive grammar into an equivalent grammar which is not left-recursive

    $$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \ \ldots \ \text{etc.} \qquad A ::= A\alpha \mid \beta$$

  - The left-recursion may appear in a single step of the derivation (immediate left recursion), or may appear in more than one step of the derivation

LL(1) grammar

- Removing left-recursion (immediate left recursion)
  - `A ::= Aα | β` where β does not start with `A`
  - `A ::= βA′` where `A′` is a new nonterminal
    `A′::= αA′ | ε` an equilvanet grammar
  - More general (but still immediate)
    - $A ::= A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$
  - Transform into:
    - $A ::= \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$
      $A' ::= \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$

YONSEI UNIVERSITY

LL(1) grammar

- Example – Removing left-recursion (immediate left recursion)

$$E ::= E + T \mid T \longrightarrow \begin{cases} E ::= TE' \\ E' ::= +TE' \mid \varepsilon \end{cases}$$

$$T ::= T * F \mid F \longrightarrow \begin{cases} T ::= FT' \\ T' ::= *FT' \mid \varepsilon \end{cases}$$

$$F ::= (E) \mid id \longrightarrow F ::= (E) \mid id$$

LL(1) grammar

- Removing left-recursion (in more than one step)
  - A grammar is not immediately left-recursive, but it still can be left-recursive
  - By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive
  - Example
    - `S ::= Af | b`
`A ::= Ac | Sd | e`
    - Is `A` left recursive? Yes.
    - Is `S` left recursive? Yes, but not immediate left recursion
      - `S ⇒ Af => Sdf`

LL(1) grammar

```
S ::= Af | b
A ::= Ac | Sd | e
```

- **Removing left-recursion (in more than one step)**
  - Approach
    - Look at the rules for `S` only (ignoring other rules) – no left recursion
    - Look at the rules for `A`
      - Do any of `A`'s rules start with `S`? Yes

        ```
        A ::= Sd
        ```

      - Get rid of the `S`, and substitute in the righthand sides of `S`

        ```
        A ::= Afd | bd
        ```

      - The modified grammar

        ```
        S ::= Af | b
        A ::= Ac | Afd | bd | e
        ```

      - Now eleiminate immediate left recursion involving `A`

        ```
        S ::= Af | b
        A ::= bdA′ | eA′
        A′::= cA′ | fdA′ | ε
        ```

YONSEI UNIVERSITY

LL(1) grammar

- Removing left-recursion
  - Input: Grammar G with no cycles or ε-productions
  - Arrange the non terminals (variables) in some order A_1, A_2, …, A_n
  - Perform the following process

```
for each nonterminal A_i (for i = 1 to n):
  for each nonterminal A_j (for j = 1 to i - 1):
    Let A_j ::= β_1 | β_2 | β_3 | … | β_N  be all the rules for A_j
    if there is a rule of the form
      A_i ::= A_jα
    then replace it by
      A_i ::= β_1α | β_2α | β_3α | … | β_Nα
    endif
  endfor
  eliminate the immediate left recursion in among the A_i rules
endfor
```

A_1
A_2
A_3
A_4 Inner loop
A_5
…
A_j
…
A_i ← Outer loop

LL(1) grammar – self study slide

- Example – removing left-recursion
  - Original:

```
S  ::= Af | b
A  ::= Ac | Sd | Be
B  ::= Ag | Sh | k
```

  - For A:

```
S  ::= Af | b
A  ::= bdA′ | BeA′
A′::= cA′ | fdA′ | ε
```

  - For B

```
S  ::= Af | b
A  ::= bdA′ | BeA′
A′::= cA′ | fdA′ | ε
B  ::= Ag | Sh | k
```

Does any rhs start with “S”?

LL(1) grammar – self study slide

- Example – removing left-recursion
  - For B

```
S  ::= Af | b
A  ::= bdA′ | BeA′
A′ ::= cA′ | fdA′ | ε
B  ::= Ag | Afh | bh | k
```

Substitutions with the rules for “S”

Does any rhs start with “A”?

  - For B

```
S  ::= Af | b
A  ::= bdA′ | BeA′
A′ ::= cA′ | fdA′ | ε
B  ::= bdA′g | BeA′g | bdA′fh | BeA′fh | bh | k
```

Substitutions with the rules for “A”

  - For B

```
S  ::= Af | b
A  ::= bdA′ | BeA′
A′ ::= cA′ | fdA′ | ε
B  ::= bdA′gB′ | bdA′fhB′ | bhB′ | kB′
B′ ::= eA′gB′ | eA′fhB′ | ε
```

Eliminate immediate left recursion related to “B”

YONSEI UNIVERSITY

LL(1) grammar – self study slide

- Example – removing left-recursion
  - Observe that

```
expr ::= expr + term | term
```

  - generates the sequence

```
term + term + term + … + term
```

  - We can sugar the original rule to show this

```
expr ::= term { + term }*
```

  - This leads directly to parser code

LL(1) grammar – self study slide

```
// parse
// expr ::=  term { + term }*
void parseExpr() {
  parseTerm();
  while (next symbol is PLUS) {
    getNextToken();
  parseTerm()
  }
}
// parse
// term ::= factor { * factor }*
void parseTerm() {
  parseFactor();
  while (next symbol is TIMES) {
    getNextToken();
    parseFactor()
  }
}
```

```
// parse
// factor ::= int | id | ( expr )
void parseFactor() {
  switch(nextToken) {
    case INT:
      process int constant;
      getNextToken();
      break;
    case ID:
      process identifier;
      getNextToken();
      break;
    case LPAREN:
      getNextToken();
      expr();
      getNextToken();
         …
  }
}
```

LL(1) grammar

- **Common prefix problem and left factoring**
  - Uncertain which of 2 rules to choose
    - `ifStmt ::=  if ( expr  ) then stmt      |`
      `           if ( expr  ) then stmt  else stmt`
  - When do you know which one is valid?
  - What's the general form of `stmt`?
    - $A ::= \alpha\beta_1 \mid \alpha\beta_2$
      - α : `if ( expr  ) then stmt`
      - β_1 : `else stmt`
      - β_2 : ε
  - How can we refactor the grammar?
    - $A ::= \alpha A'$
      $A' ::= \beta_1 \mid \beta_2$

YONSEI UNIVERSITY

LL(1) grammar

- Example – common prefix problem and left factoring
  - Original:
    ```
    A   ::=  abB | aB | cdg | cdeB | cdfB
    ```
  - First step:
    ```
    A   ::= aA' | cdg | cdeB | cdfB
    A'  ::= bB | B
    ```
  - Second step:
    ```
    A   ::= aA' | cdA''
    A'  ::= bB | B
    A''::= g | eB | fB
    ```

LL(1) grammar – self study slide

- Example – common prefix problem and left factoring
  - Original:

```
A   ::= ad | a | ab | abc | b
```

  - First step:

```
A   ::= aA′ | b
A′  ::= d |  ε | b | bc
```

  - Second step:

```
A   ::= aA′ | b
A′  ::= d | ε | bA′′
A′′::= ε | c
```

LL(1) grammar – self study slide

- Example – common prefix problem and left factoring
  - Original grammar

```
ifStmt ::=   if ( expr ) stmt   |
             if ( expr ) stmt  else stmt
```

  - Factored grammar

```
ifStmt ::= if ( expr ) stmt  ifTail
ifTail ::= else stmt  | ε
```

➔ But it's easiest to just code up the "`else` matches closest `if`" rule directly

```
// parse
// if (expr) stmt [ else stmt ]
void ifStmt() {
  getNextToken();
  getNextToken();
  parseExpr();
  getNextToken();
  parseStmt();
  if (next symbol is ELSE) {
    getNextToken();
    parseStmt();
  }
}
```

LL(1) grammar

- LL(1) grammar
  - The grammar must not have ambiguity
  - The grammar must not be left recursive
  - The rule which should be chosen when developing a nonterminal must be determined by that nonterminal and the (at most) next token on the input
  - Key benefits of LL(1) grammar
    - LL(1) grammar ahs the conceptual and practical advantage that they allow the compiler writer to view the grammar as a program
    - this allows a more natural positioning of semantic actions and a simple attribute mechanisms

LL(1) parser

LL(1) parser

- **LL(K) parser and LL(1) parser**
  - LL(K) parsers
    - Scans the input Left to right
    - Constructs a Leftmost derivation
    - Looking ahead at most k symbols
  - LL(1) parser
    - 1-symbol lookahead is enough for many practical programming language grammars
    - LL(k) for k > 1 is very rare in practice
    - It can parse LL(1) languages

LL(1) parser

- Why LL(1)
  - A few characteristics of recursive descent parsers
    - Actual pieces of code that can be read by programmers and extended
    - Fairly easy to understand how parsing is done
    - Quite inconvenient to change the grammar being parsed
    - Any change, even a minor one, may force parsing procedures to be reprogrammed
  - Alternative way to make a parser
    - Encode all prediction in a parsing table
    - A pre-programed driver program can use a parse table (and list of productions) to parse any LL(1) grammar
    - If a grammar is changed, the parse table and list of productions will change, but the driver need not be changed

LL(1) parser

- FIRST, FOLLOW, and nullable – Required for building parsing tables
  - FIRST(A), where a is any string of grammar symbols, to be the set of terminals that begin strings derived from A
    - If FIRST(α) and FIRST(β) disjoint, we can choose A::=α or A::=β properly
  - FOLLOW(A) is a set of terminals a that can immediately follow A in some derivation, for nonterminal A
    - But to do this, we need to compute FIRST(γ) for strings γ that can follow *A*
  - nullable(A) is true if A can derive the empty string
    - If γ = AYZ, then then FIRST(γ) is FIRST(A), but when A::=ε?
    - Then, FIRST(γ) includes anything that can follow an A – i.e., FOLLOW(A)
  - Given a string γ of terminals and non-terminals, FIRST(γ) is the set of terminals that can begin strings derived from γ
  - All three of these are computed together

YONSEI UNIVERSITY

LL(1) parser

- FIRST
  - Informal definition:
    - The first set of a RE X is the set of terminal symbols that can occur as the start of any string generated by X
  - Example:

`FIRST[(+|-|ε)(0|1| … |9)*] = {+,-,0,1,…,9}`

  - Formal definition:

$\mathrm{FIRST}[\varepsilon]$	$= \{\}$	
$\mathrm{FIRST}[t]$	$= \{t\}$	(where t is a terminal symbol)
$\mathrm{FIRST}[X\ Y]$	$= \mathrm{FIRST}[X] \cup \mathrm{FIRST}[Y]$	(if X generates ε)
$\mathrm{FIRST}[X\ Y]$	$= \mathrm{FIRST}[X]$	(if not X generates ε)
$\mathrm{FIRST}[X\|Y]$	$= \mathrm{FIRST}[X] \cup \mathrm{FIRST}[Y]$	
$\mathrm{FIRST}[X^*]$	$= \mathrm{FIRST}[X]$	

YONSEI UNIVERSITY

LL(1) parser

```
Expression ::= PrimaryExp Operator Expression
             | PrimaryExp
PrimaryExp ::= Identifiers | ( Expression )
Identifiers::= [a-z]
```

- FIRST
  - Informal definition:
    - The first set of RE can be generalized to BNF
  - Formal definition:

`FIRST[N] = FIRST[X]` (for production rules `N ::= X`)

  - Example:

```
FIRST[Expression]     = FIRST[PrimaryExp (Operator Expression)*]
                      = FIRST[PrimaryExp]
                      = FIRST[Identifiers] ∪ FIRST[(Expression)]
                      = FIRST[a | b | c | ... |z] ∪ {(}
                      = {a, b, c,…, z, (}
```

YONSEI UNIVERSITY

LL(1) parser

- FOLLOW
  - Informal definition:
    - The set of terminals that can immediately follow nonterminal `A` (`Follow(A)` is the set of prefixes of strings of terminals that can follow any derivation of `A` in the grammar)

```
FOLLOW(A) =
  if A is the start symbol S then
    add $ to FOLLOW(A)
  for all (B ::= α A β) ∈ P do
    add FIRST(β)-{ε} to FOLLOW(A)
  for all (B ::= α A β) ∈ P and ε ∈ FIRST(β) do
    add FOLLOW(B) to FOLLOW(A)
```

  - The definition of follow usually results in recursive set definitions.
  - In order to solve them, you need to do several iterations on the equations.

YONSEI UNIVERSITY

LL(1) parser – self study slide

- FIRST, FOLLOW, and nullable
  - Initialization

```
set FIRST and FOLLOW to be empty sets
set nullable to false for all non-terminals
set FIRST[a] to a for all terminal symbols a
```

  - Repeat the following process until FIRST, FOLLOW, and nullable do not change

for each production $X := Y_1\ Y_2 \dots Y_k$
  if $Y_1 \dots Y_k$ are all nullable (or if k = 0): set nullable[X] = true
  for each i from 1 to k and each j from i+1 to k
    if $Y_1 \dots Y_{i-1}$ are all nullable (or if i = 1)
      add FIRST[Y_i] to FIRST[X]
    if $Y_{i+1} \dots Y_k$ are all nullable (or if i = k)
      add FOLLOW[X] to FOLLOW[Y_i]
    if $Y_{i+1} \dots Y_{j-1}$ are all nullable (or if i + 1 = j)
      add FIRST[Y_j] to FOLLOW[Y_i]

LL(1) parser – self study slide

- Example – FIRST & FOLLOW

```
E ::= T E′
E′::= + T E′   | ε
T ::= F T′
T′::= * F T′   | ε
F ::=( E )     | id
```

```
First(E)  = First(T) = First(F) = {(,id}
First(E′) = {+, ε}
First(T′) = {*, ε}
```

```
Follow(E) = Follow(E′) = { ), $ }
Follow(T) = Follow(T′) = {+, ), $ }
Follow(F) = {+, *, ), $}
```

LL(1) parser – self study slide

- Example – FIRST, FOLLOW, and nullable

Grammar

Z ::= d

Z ::= X Y Z

Y ::= ε

Y ::= c

X ::= Y

X ::= a

	FIRST	FOLLOW	nullable
X			
Y			
Z			

LL(1) parser

- **Formal definition of LL(1) grammar**
  - A grammar G is LL(1) iff
    - For each set of productions $M ::= X_1 \mid X_2 \mid \ldots \mid X_n$:
      - $FIRST(X_1)$, $FIRST(X_2)$, …, $FIRST(X_n)$ are all pairwise disjoint
      - If $X_i ::=^* \varepsilon$ then $FIRST(X_j) \cap FOLLOW(X_i) = \emptyset$, for $1 \leq j \leq n.\ i \neq j$
      - If G is ε-free, then the first condition is sufficient
  - Properties
    - No left-recursive grammar is LL(1)
    - No ambiguous grammar is LL(1)
    - A ε-free grammar, where each alternative X_j for $N ::= X_j$ $(1 \leq j \leq n)$ begins with a distinct terminal, is a simple LL(1) grammar

LL(1) parser

- Example – LL(1) grammar
  - `S  ::= aS | a`
    - Is not LL(1)
    - `FIRST(aS)=FIRST(a)={a}`
  - `S’ ::= aS`
    `S  ::= aS | ε`
    - Accepts the same language and is LL(1)

LL(1) parser

- Construction of parsing table
  - An LL(1) parse table, T, is a two dimensional array.
  - Entries in T are production numbers or blank (error) entries.
  - T is indexed by:
    - A, a non-terminal.
      - A is the nonterminal we want to expand.
    - CT, the current token that is to be matched.
    - $T[A][CT] = A ::= X_1 \ldots X_n$
      - if CT is in FIRST($A ::= X_1 \ldots X_n$), then perform the proper derivation
      - if CT predicts no production with A as its lefthand side: T[A][CT] = error

YONSEI UNIVERSITY

LL(1) parser

```
Prog  ::= { Stmts }
Stmts ::= Stmt Stmts | ε
Stmt  ::= id = Expr ; | if ( Expr ) Stmt
Expr  ::= id Etail
Etail ::= + Expr | - Expr | ε
```

- Example – LL(1) parsing table

1	`Prog  ::= { Stmts } $`	{
2	`Stmts ::= Stmt Stmts`	id if
3	`Stmts ::= ε`	}
4	`Stmt  ::= id = Expr ;`	id
5	`Stmt  ::= if ( Expr ) Stmt`	if
6	`Expr  ::= id Etail`	id
7	`Etail ::= + Expr`	+
8	`Etail ::= - Expr`	-
9	`Etail ::= ε`	) ;

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

LL(1) parser

- LL(1) parser driver
  - A driver that uses the parsing table.
  - LL(1) pasing also uses a *parse stack* that remembers symbols we have yet to match.

```
void LLDriver(){
  Push(StartSymbol);
  while(!stackEmpty()){
    //Let X=Top symbol on parse stack
    //Let CT = current token to match
    if (isTerminal(X)) {
      match(X);                          // CT is updated
      pop();                             // X is updated
    } else if (T[X][CT] != Error){
      //Let T[X][CT] = X::=Y1...Ym
      Replace X with Y1...Ym on parse stack
    } else { SyntaxError(CT); }
  }
}
```

LL(1) parser

```
Prog  ::= { Stmts }
Stmts ::= Stmt Stmts | ε
Stmt  ::= id = Expr ; | if ( Expr ) Stmt
Expr  ::= id Etail
Etail ::= + Expr | - Expr | ε
```

- **Example – LL(1) parsing** ({ a = b + c; } $)
  - Start by placing Prog (the start symbol) on the parse stack.

Parse Stack	Remaining Input
Prog	{ a = b + c; } $

{
Stmts
}
$
	{ a = b + c; } $

Stmts } $	a = b + c; } $
Stmt Stmts } $	a = b + c; } $

1	Prog ::= { Stmts } $	{
2	Stmts ::= Stmt Stmts	id if
3	Stmts ::= ε	}
4	Stmt ::= id = Expr ;	id
5	Stmt ::= if (Expr) Stmt	if
6	Expr ::= id Etail	id
7	Etail ::= + Expr	+
8	Etail ::= - Expr	-
9	Etail ::= ε	) ;

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

Parse Stack	Remaining Input
id = Expra ; Stmts } $	a = b + c; } $
= Expr ; Stmts } $	= b + c; } $
Expr ; Stmts } $	b + c; } $
id Etail ; Stmts } $	b + c; } $

```
Prog  ::= { Stmts }
Stmts ::= Stmt Stmts | ε
Stmt  ::= id = Expr ; | if ( Expr ) Stmt
Expr  ::= id Etail
Etail ::= + Expr | - Expr | ε
```

1	Prog ::= { Stmts } $	{
2	Stmts ::= Stmt Stmts	id if
3	Stmts ::= ε	}
4	Stmt ::= id = Expr ;	id
5	Stmt ::= if (Expr) Stmt	if
6	Expr ::= id Etail	id
7	Etail ::= + Expr	+
8	Etail ::= - Expr	-
9	Etail ::= ε	) ;

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

Parse Stack	Remaining Input
Etail ; Stmts } $	+ c; } $
+ Expr ; Stmts } $	+ c; } $
Expr ; Stmts } $	c; } $
id Etail ; Stmts } $	c; } $

```
Prog  ::= { Stmts }
Stmts ::= Stmt Stmts | ε
Stmt  ::= id = Expr ; | if ( Expr ) Stmt
Expr  ::= id Etail
Etail ::= + Expr | - Expr | ε
```

1	Prog ::= { Stmts } $	{
2	Stmts ::= Stmt Stmts	id if
3	Stmts ::= ε	}
4	Stmt ::= id = Expr ;	id
5	Stmt ::= if (Expr) Stmt	if
6	Expr ::= id Etail	id
7	Etail ::= + Expr	+
8	Etail ::= - Expr	-
9	Etail ::= ε	) ;

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

Parse Stack	Remaining Input
Etail ; Stmts } $	; } $
; Stmts } $	; } $
Stmts } $	} $
} $	} $
$	$
Done!	All input matched

```
Prog  ::= { Stmts }
Stmts ::= Stmt Stmts | ε
Stmt  ::= id = Expr ; | if ( Expr ) Stmt
Expr  ::= id Etail
Etail ::= + Expr | - Expr | ε
```

1	Prog ::= { Stmts } $	{
2	Stmts ::= Stmt Stmts	id if
3	Stmts ::= ε	}
4	Stmt ::= id = Expr ;	id
5	Stmt ::= if (Expr) Stmt	if
6	Expr ::= id Etail	id
7	Etail ::= + Expr	+
8	Etail ::= - Expr	-
9	Etail ::= ε	) ;

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

YONSEI UNIVERSITY

LL(1) parser – self study slide

- Examples – LL(1) parser

Grammar G

E::=TE'
E'::=+TE'|ε
T::=FT'
T'::=*FT'|ε
F::=(E)|id

	(	)	id	+	*	$
E	E::=TE'		E::=TE'			
E'		E'::=ε		E'::=+TE'		E'::=ε
T	T::=FT'		T::=FT'			
T'		T'::=ε		T'::=ε	T'::=*FT'	T'::=ε
F	F::=(E)		F::=id			

FIRST(E)=FIRST(T)={(, id}   FIRST(E')={+, ε}
FIRST(F)={(, id}   FIRST(T')={*, ε}
FIRST(F)={(, id}
FOLLOW(E)={$,)}
FOLLOW(E')={$,)}
FOLLOW(T)=FIRST(E') ∪ FOLLOW(E) ∪ FOLLOW(E')
FOLLOW(T')=FOLLOW(T)={+, $,)}
FOLLOW(F)=FIRST(T')={*} ∪ FOLLOW(T) ∪
FOLLOW(T')={*, +, $,)}

Parse
id+id*id

stack	Input	output
E$	id+id*id$	
TE'$	id+id*id$	E::=TE'
FT'E'$	id+id*id$	T::=FT'
idT'E'$	id+id*id$	F::=id
T'E'$	+id*id$	pop
E'$	+id*id$	T::=ε
+TE'$	+id*id$	E'::=+TE'
TE'$	id*id$	pop

LL(1) parser

- Examples – LL(1) parser

Grammar G

```
S::=iEtSS' | a
S'::=eS | ε
E::=b
```

	s	b	e	i	t	$
S	S::=a			S::=iEtSS'		
S'			S'::=eS S'::=ε			S'::=ε
E		E::=b				

```
FIRST(S)    = {i, a}
FIRST(S')   = {e, ε}
FIRST(E)    = {b}
FOLLOW(S)   = {$,e}∪FOLLOW(S') = {$, e}
FOLLOW(S')  = FOLLOW(S) = {e, $}
FOLLOW(E)   = {t}
```

LL(1) parser

- Syntax errors in LL(1) parsing
  - Syntax errors are automatically detected as soon as the first illegal token is seen.
  - When an illegal token is seen by the parser, either it fetches an error entry from the LL(1) parsing table, or it fails to match an expected token.
- Example – LL(1) parsing (`{ b + c = a; } $`)

Parse Stack	Remaining Input
Prog	{ b + c = a; } $

{
Stmts
}
$
	{ b + c = a; } $

1	Prog ::= { Stmts } $	{
2	Stmts ::= Stmt Stmts	id if
3	Stmts ::= ε	}
4	Stmt ::= id = Expr ;	id
5	Stmt ::= if (Expr) Stmt	if
6	Expr ::= id Etail	id
7	Etail ::= + Expr	+
8	Etail ::= - Expr	-
9	Etail ::= ε	) ;

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

YONSEI UNIVERSITY

Parse Stack	Remaining Input

`Stmts`
`}`
`$`
	`b + c = a; } $`

`Stmt Stmts`
`}`
`$`
	`b + c = a; } $`

`id`
`=`
`Expr`
`;`
`Stmts`
`}`
`$`
	`b + c = a; } $`

`=`
`Expr`
`;`
`Stmts`
`}`
`$`
	`+ c = a; } $`

Current token (+) fails to match expected token (=)!	`+ c = a; } $`

1	`Prog ::= { Stmts } $`	`{`
2	`Stmts ::= Stmt Stmts`	`id if`
3	`Stmts ::= ε`	`}`
4	`Stmt ::= id = Expr ;`	`id`
5	`Stmt ::= if ( Expr ) Stmt`	`if`
6	`Expr ::= id Etail`	`id`
7	`Etail ::= + Expr`	`+`
8	`Etail ::= - Expr`	`-`
9	`Etail ::= ε`	`) ;`

	{	}	if	(	)	id	=	+	-	;	$
Prog	1										
Stmts		3	2			2					
Stmt			5			4					
Expr						6					
Etail					9			7	8	9	

Questions?