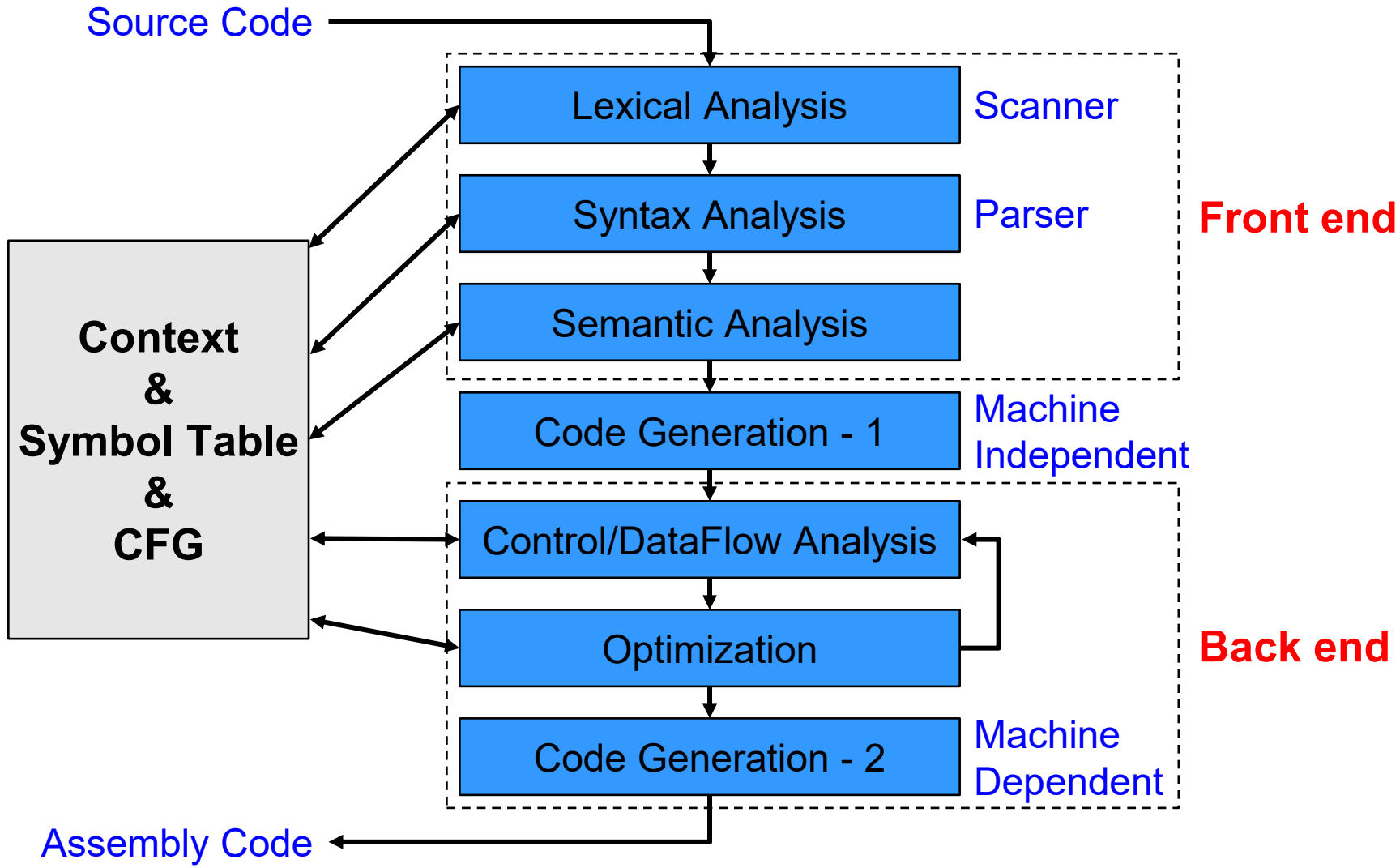
3. Syntax Analysis

2025 Fall

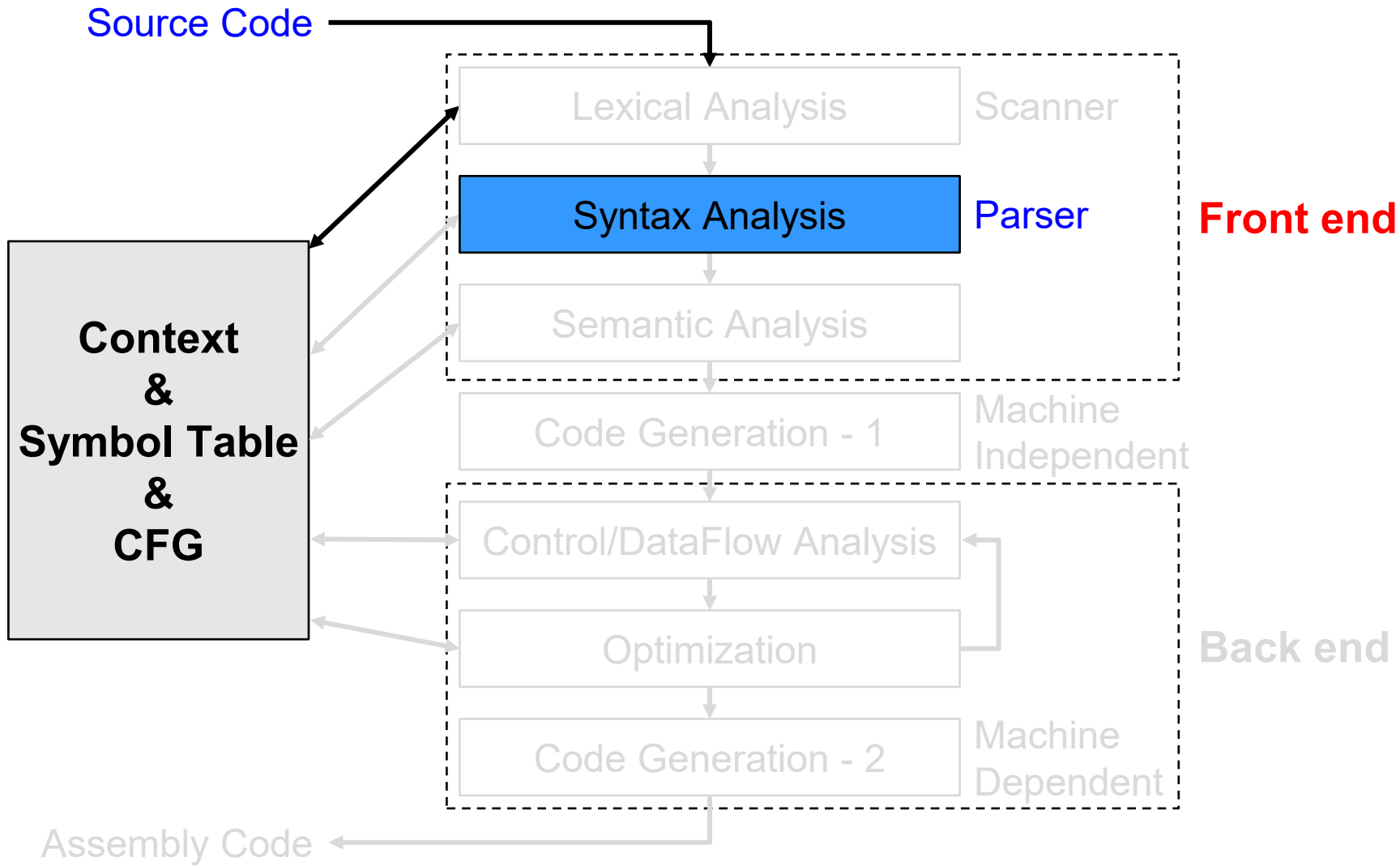
Hunjun Lee

Hanyang University

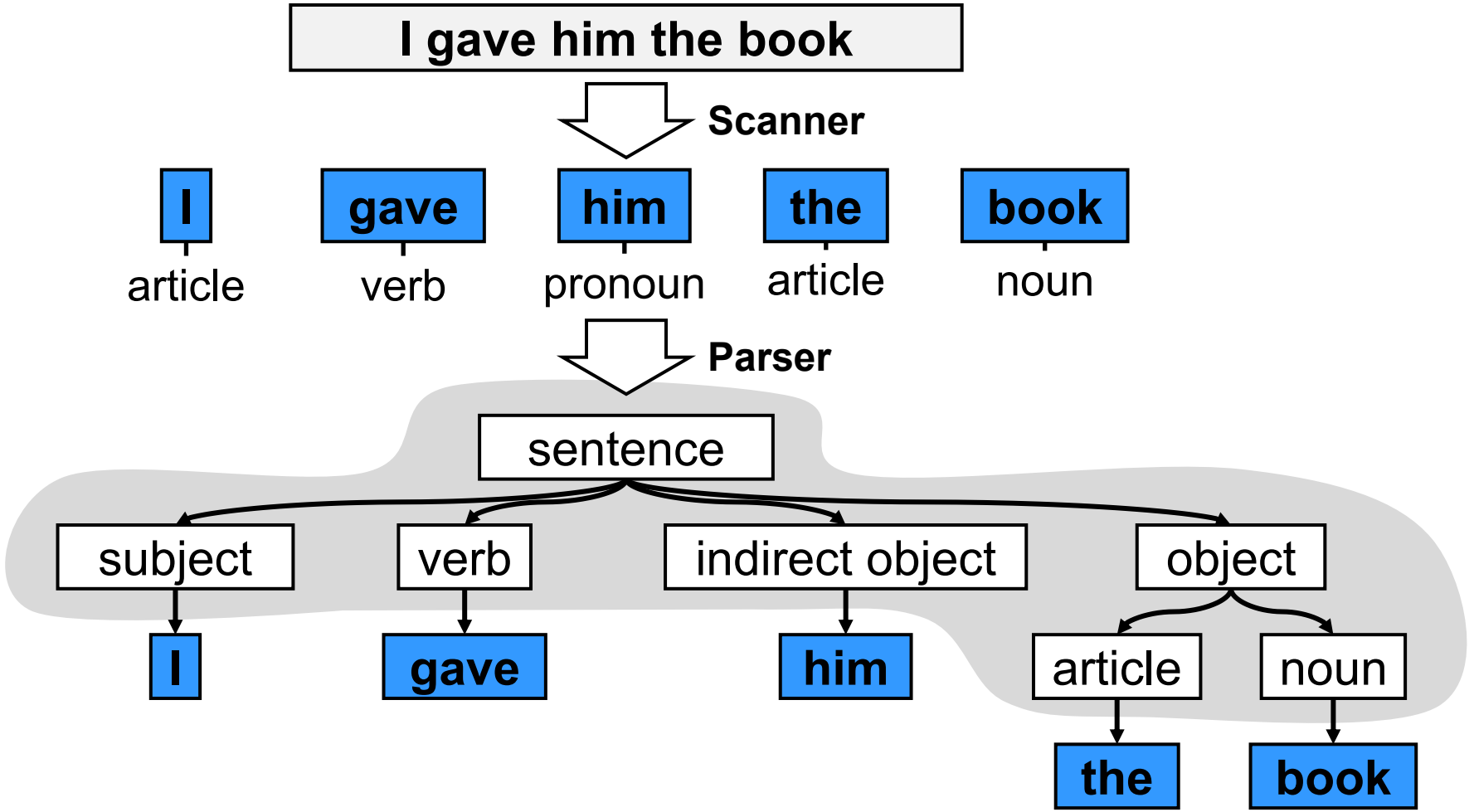
Syntax Analysis



Syntax Analysis

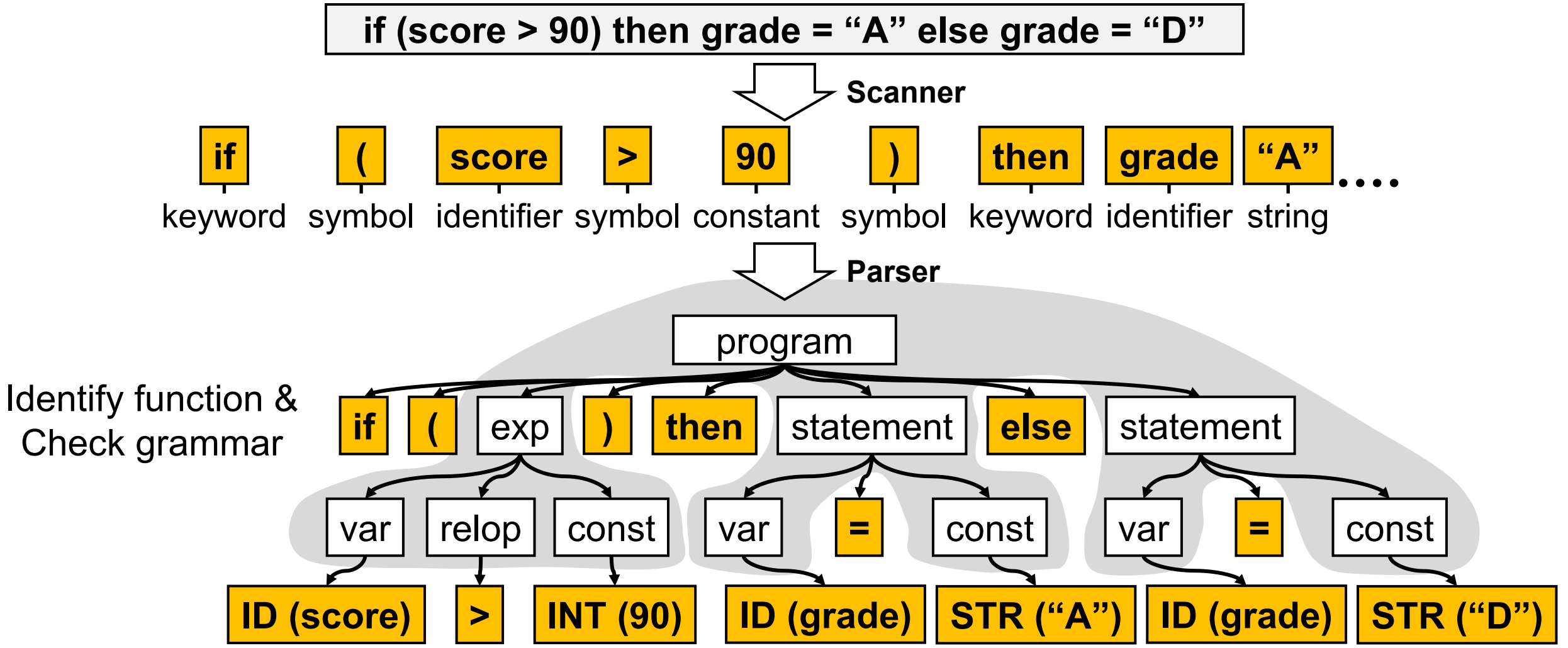


Syntax Analysis in Language



Identify function &
Check grammar

Syntax Analysis in Program



Recap: Specification, Recognition, and Automation in Scanner

- **Specification: how to specify lexical patterns?**
  - Regular expression (RE)
- **Recognition: how to recognize the specified patterns?**
  - Deterministic finite automata (DFA)
- **Automation: how to generate DFA from RE?**
  - Automatic generation tool (Lex)
  - Thompson's construction and subset construction

Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid sequence of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid sequence of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

Context-Free Grammars (CFG) - 1

- **There are four main components of CFG**
  - Terminal symbols — token or ε
  - Non-terminal symbols — syntactic variables
  - Start symbol S — special non-terminal
  - Productions — translating an LHS to RHS (derivations)
    - LHS: a single non-terminal
    - RHS: string of terminals and/or non-terminals

Context-Free Grammars (CFG) - 2

- **Terminals are the tokens: keywords, symbols, ID, NUM …**
- **Non-terminals**
  - var / function
  - expression (e.g., var + var)
  - return statement / for loop / if statement
- **Start Symbol (S): “program” in compiler**
- **Productions**
  - type-specifier ➔ int | float | bool …
  - var-declaration ➔ type-specifier ID ; | type-specifier ID [NUM] ;
  - function-declaration ➔ type-specifier ID (params) compound-stmt

More Specifically …
➔ To Be Covered in the Project

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* `ID ;` | *type-specifier* `ID [ NUM ] ;`
5. *type-specifier* → `int` | `void`
6. *fun-declaration* → *type-specifier* `ID (` *params* `)` *compound-stmt*
7. *params* → *param-list* | `void`
8. *param-list* → *param-list* `,` *param* | *param*
9. *param* → *type-specifier* `ID` | *type-specifier* `ID [ ]`
10. *compound-stmt* → `{` *local-declarations statement-list* `}`
11. *local-declarations* → *local-declarations var-declarations* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* `;` | `;`

More Specifically …
➔ To Be Covered in the Project

15. *selection-stmt* → **if** (*expression*) *statement* | **if** (*expression*) *statement* **else** *statement*
16. *iteration-stmt* → **while** (*expression*) *statement*
17. *return-stmt* → **return** ; | **return** *expression* ;
18. *expression* → *var* = *expression* | *simple-expression*
19. *var* → **ID** | **ID** [*expression*]
20. *simple-expression* → *additive-expression relop additive-expression* | *additive-expression*
21. *relop* → <= | < | > | >= | == | !=
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → + | -
24. *term* → *term mulop factor* | *factor*
25. *mulop* → * | /
26. *factor* → (*expression*) | *var* | *call* | **NUM**
27. *call* → **ID** (*args*)
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* , *expression* | *expression*

Language in CFG

- **Language generated by a CFG is set of strings of terminals by repeatedly applying productions to the non-terminals**
  - L(G) indicates a language generated by the grammar G

Productions

S ➔ if C then S |
if C then S else S |
return ; |
ID (ID) ; |
E

C ➔ ID == ID | ID > ID | ID < ID

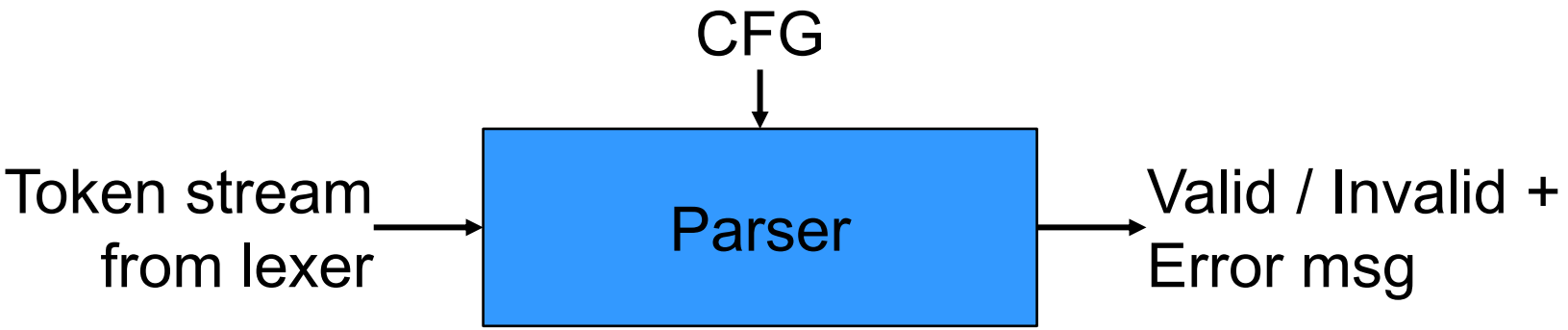
E ➔ ID + ID ; | ID * ID ; | ID / ID ;

Languages

1. if ID == ID then return ;
2. return;
3. ID (ID) ;
4. if ID > ID then return; else ID (ID) ;
5. if ID < ID then if ID < ID then …
6. if ID == ID then ID (ID);
7. ID + ID ;

Context-Free Grammars

- **We can use CFGs to express the syntax of the target programming languages**
  - Derivation: Apply productions to the non-terminals in the strings
  - Acceptance: If an input token stream is a derivation

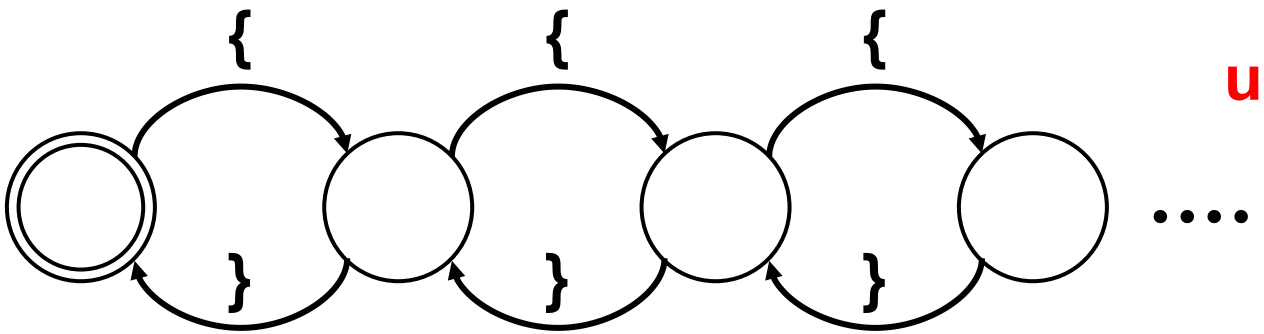


CFG Example - 1

- **To define balanced-curly brace using CFG (e.g., {{}}, {}{}{}, …)**
  - Terminals: {, } / Non-Terminals: S
  - Start symbol S
  - Productions
    - S ➔ ε | { S } S
- **CFG accepts a string if it is described using the productions**
  - For an input {{}}
    - **Step 1)** S ➔ { S } S
    - **Step 2)** { S } S ➔ {{ S } S } S
    - **Step 3)** {{ S } S } S ➔ {{ ε } S } S
    - **Step 4)** {{} S } S ➔ {{} ε } S
    - **Step 5)** {{ }} S ➔ {{}} ε = {{}}

Regular Expression in Syntax Analysis

- **Why not rely on regular expressions to specify the syntax?**
  - They are not expressive enough to describe valid syntax
  - Representative examples: nested constructs
    - Define {i}i as a valid syntax ➔ There are *i* open curly braces and *i* close curly braces (e.g., {}, {{}}, {{{}}})



REs cannot support
unbounded counting

CFG Example - 2

- **To define sum expression (e.g., 5+4, (5+2)+6, 3+9+1)**
  - Terminals: number, (,), + / Non Terminals: S, E
  - Start symbol S
  - Productions
    - S ➔ E + S | E
    - E ➔ number | (S)
- **CFG accepts a string if it is described using the productions**
  - For an input (5 + 3) + 4 (number + number) + number
    - **Step 1)** S ➔ E + S
    - **Step 2)** E + S ➔ (S) + S
    - **Step 3)** (S) + S ➔ (E+ S) + S
    - **Step 4,5)** (number+E) + S … (I'll skip step 6,7,8)

Class Exercise

- **Let's say we have a G**
  - Terminals: a, b, c / Non-Terminals: S, X, Y
  - Productions
    - S ➔ aXa (start symbol S)
    - X ➔ ε | bY
    - Y ➔ ε | cXc
- **Choose which of the strings are in L(G)**
  - abcba (X) aXa → abYa → abcXca ???
  - acca (X) aXa → abYa ???
  - aba (O) aXa → abYa → aba
  - abcbcba (X) aXa → abYa → abcXca ??

CFG Example - 3

Target Program:

```
if ( ID == ID ) ID = ID ;
```

Context-Free Grammar (CFG)

```
program ➔ statement-list
statement-list ➔ statement-list statement | empty
statement ➔ if-else-statement | assign-statement | cmp-statement
if-else-statement ➔ if ( cmp-statement ) statement
assign-statement ➔ ID = ID ;
cmp-statement ➔ ID == ID
```

CFG Example - 3

CFG Derivation Example:

program ➔ statement-list

statement-list ➔ statement-list statement

statement-list statement ➔ statement

statement ➔ if-statement

if-statement ➔ if (cmp-statement) statement

if (cmp-statement) statement ➔ if (cmp-statement) assign-statement

if (cmp-statement) assign-statement ➔ if (ID == ID) assignment-statement

if (ID == ID) assignment-statement ➔ if (ID == ID) ID = ID ;

Every symbols are terminals

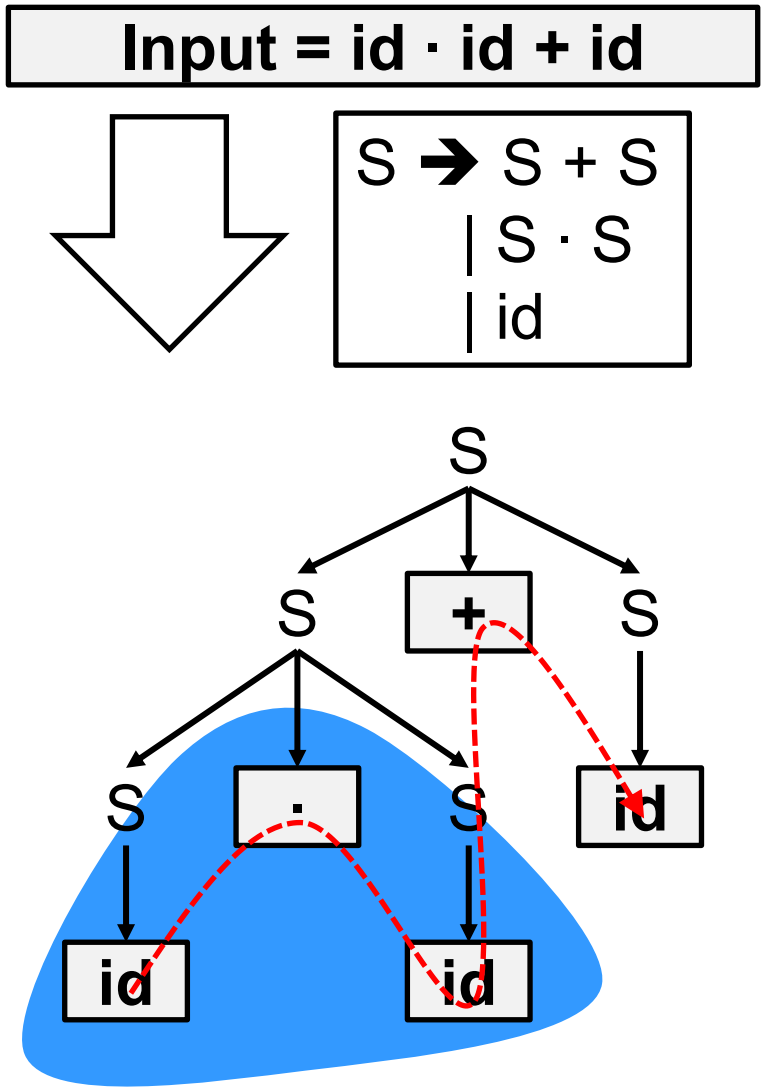
Specification, Recognition, and Automation in Parser

- Specification: how to specify valid sequence of tokens?
  - Context-free grammars (CFG)

- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)

- Automation: how to generate parse tree from CFG?
  - Top-down and bottom-up parsing

Parse Tree



Binds more tightly (· > +)

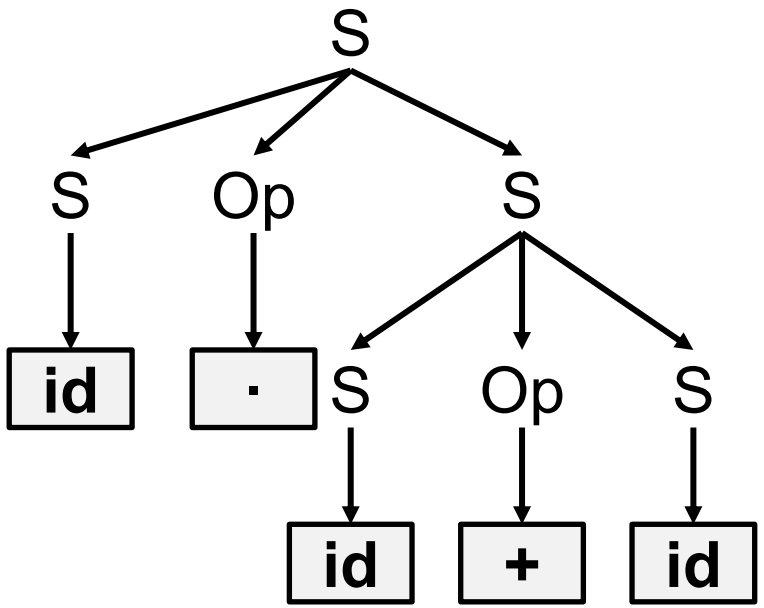
- **Tree representation of the derivation**
- **A parse tree has**
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- **An in-order traversal of the leaves is the original input**
- **Shows the order of operations**
  - Ex) Operation order, association …

Derivation Order and Parse Tree

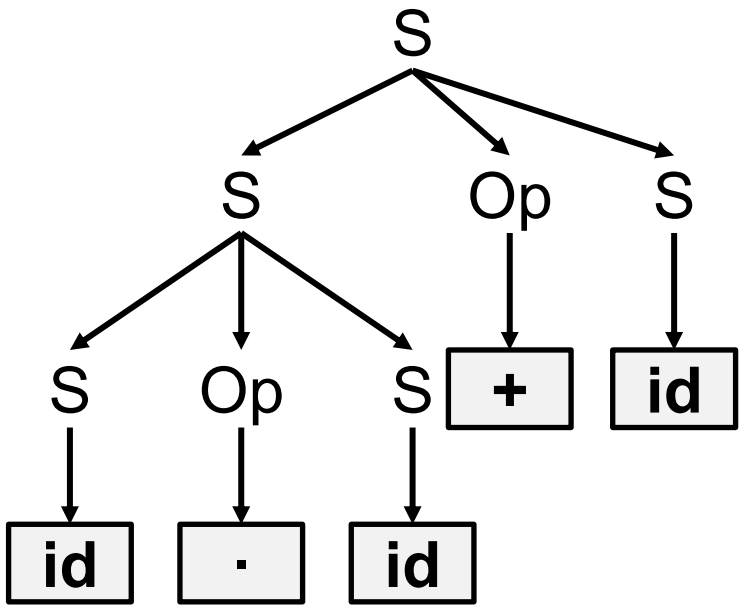
- **We can apply productions for the non-terminals in any order**
  - Left-most: Select the left-most non-terminal first
  - Right-most: Select the right-most non-terminal first

id · id + id

S ➔ S Op S | id
Op ➔ + | ·



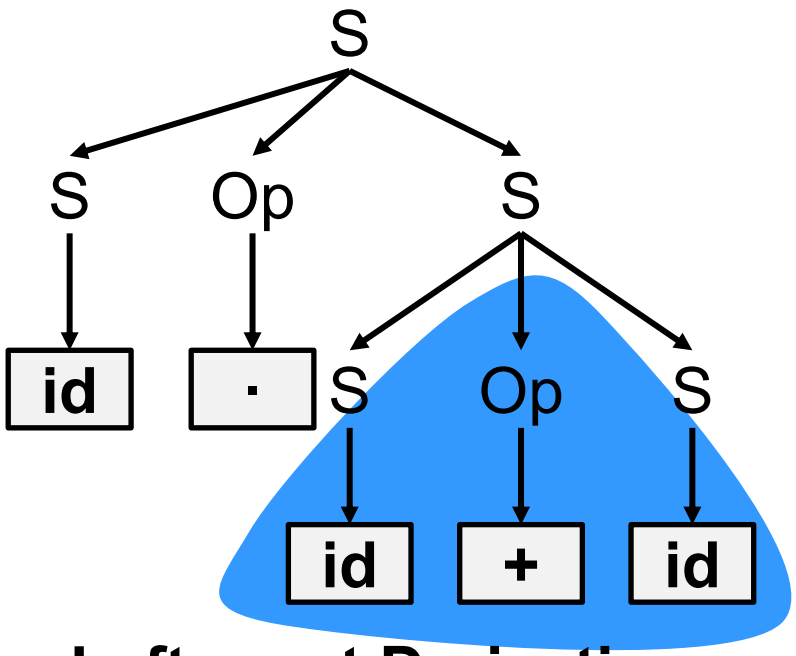
Left-most Derivation



Right-most Derivation

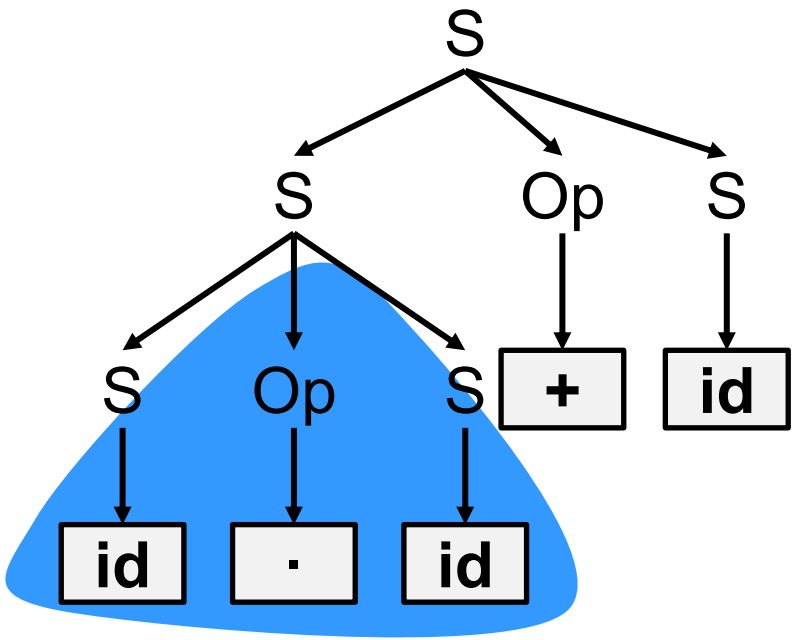
Ambiguous Grammars

- **A grammar (G) is ambiguous if it produces different parse trees depending on the order**
  - Ambiguous if the left and right-most derivation differs for some input



Left-most Derivation

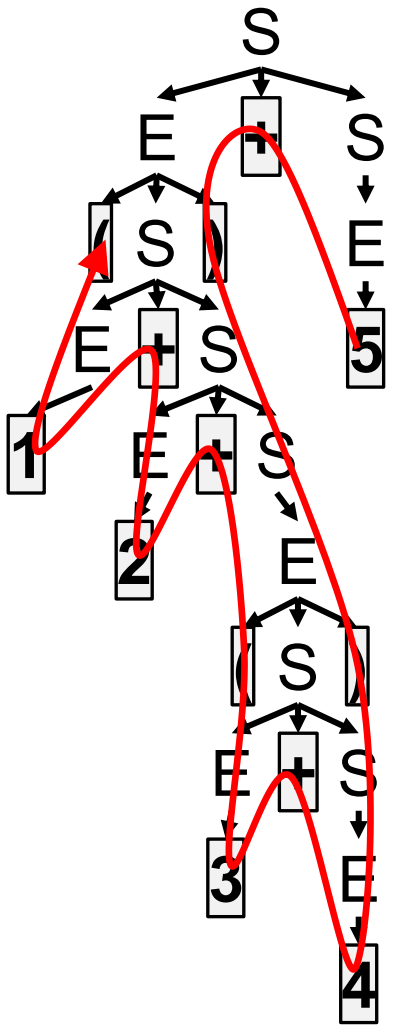
Which is correct?



Right-most Derivation

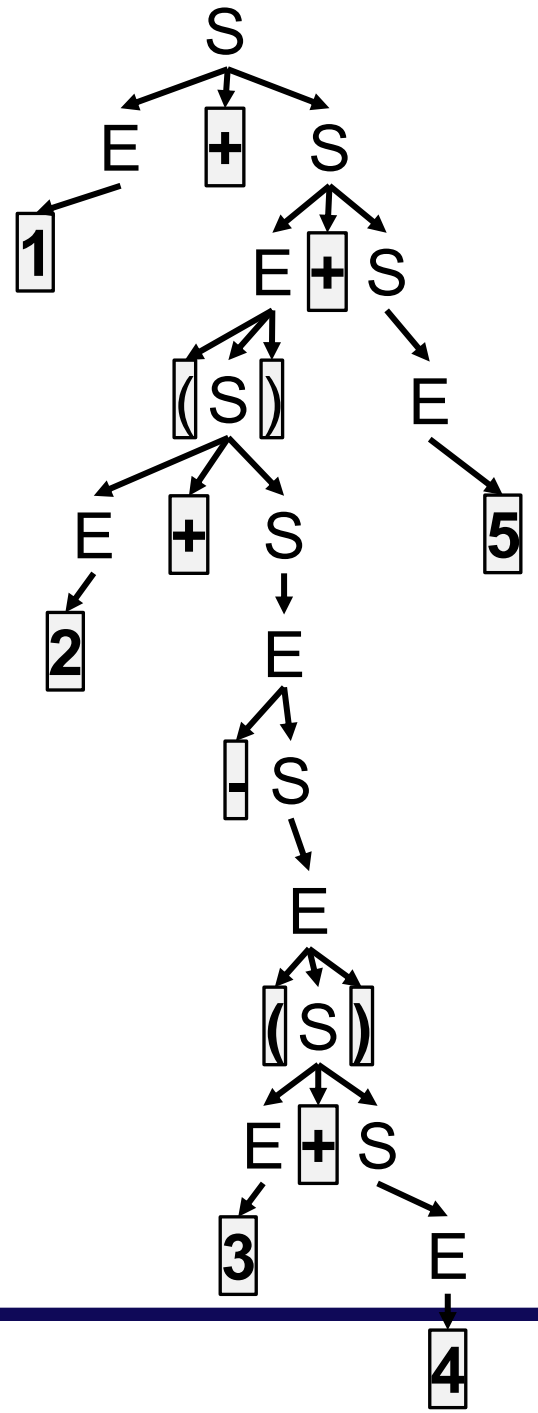
Derivation Example

- **S ➔ E + S | E**
- **E ➔ number | (S)**
- **Do the rightmost derivation of (1 + 2 + (3 + 4)) + 5**



Class Exercise

- **S → E + S | E**
- **E → number | (S) | -S**
- **Do the rightmost derivation of 1 + (2 + -(3 + 4)) + 5**



Ambiguity in Grammar and Language

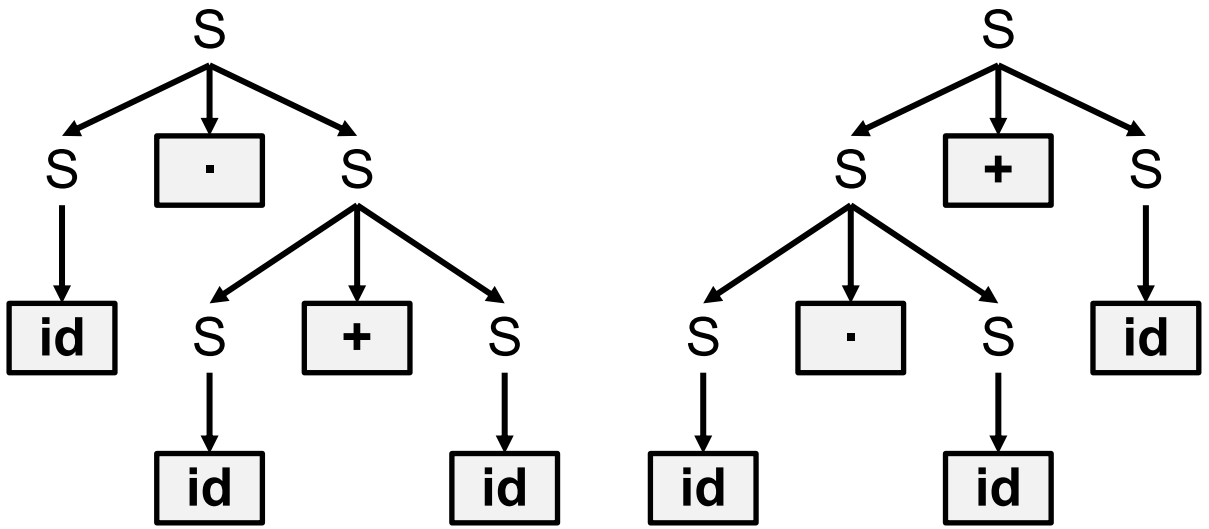
- **There are many grammars for a language**
  - A context-free language is unambiguous of all grammars are unambiguous and ambiguous if any of them are ambiguous
- **We need to choose an unambiguous grammar to construct a useful parser**
  - We need to eliminate an ambiguity in a grammar to construct a parser

Example of Ambiguity

- **Operators**
  - Precedence: some operators should precede other operators
    - Multiply and divide should precede addition and subtraction
  - Associativity: an operator is either left, right, or non associative
    - Left: a - b - c = (a - b) - c
    - Right: a ^ b ^ c = a ^ (b ^ c)
    - Non: a < b < c is invalid

Removing Ambiguity in Operator - 1

- **Precedence:**
  - The production at higher levels (closer to the root) will have operators with lower priorities (and vice versa)
  - We can insert non-terminals to enforce precedence



S ➔ S + S
| S · S | id

Ambiguous

S ➔ S + T | T
T ➔ T · id | id

Unambiguous

Removing Ambiguity in Operator - 2

- **Associativity:**
  - We should determine where to place recursion depending on the associativity
    - Left associative: place the recursion on the left
    - Right associative: place the recursion on the right
    - Non associative: do not use recursion

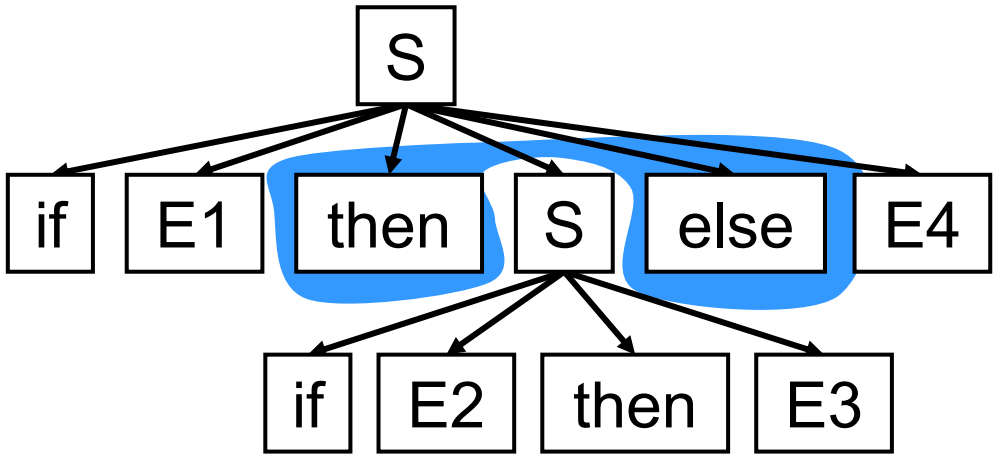
S → S - T | T T → id	S → T ^ S | T T → id	S → A < A | A A → id
Left	**Right**	**Non**

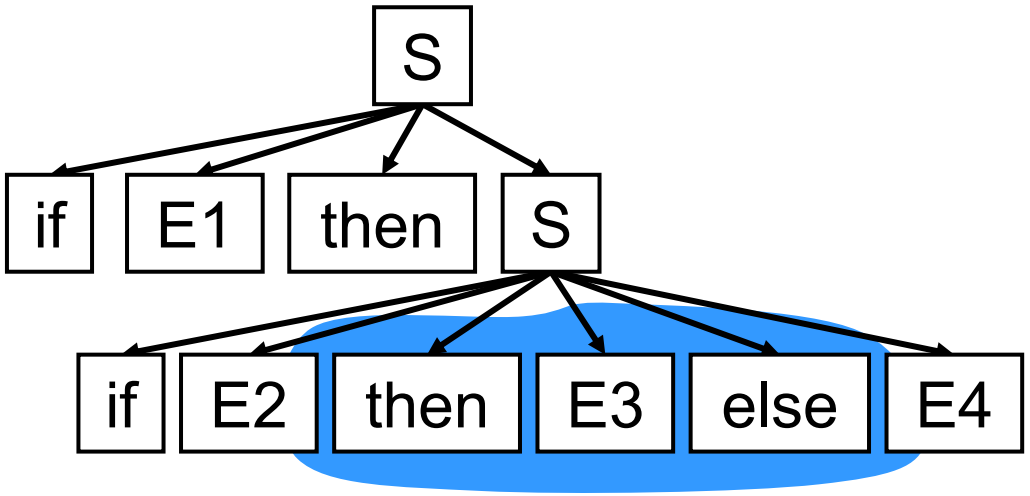
Class Exercise

- **Write an unambiguous grammar for - / · / ^ operators**
  - Consider both precedence and associativity
  - Ambiguous version : S ➔ S – S | S · S | S ^ S | id

Another Example

- **If/Then/Else**
  - We need to find the correct closure if there are both if/then/else and if/then
  - Consider “if E1 then if E2 then E3 else E4”
    - S ➔ if E then S | if E then S else S | E
    - We should match else to the closest then

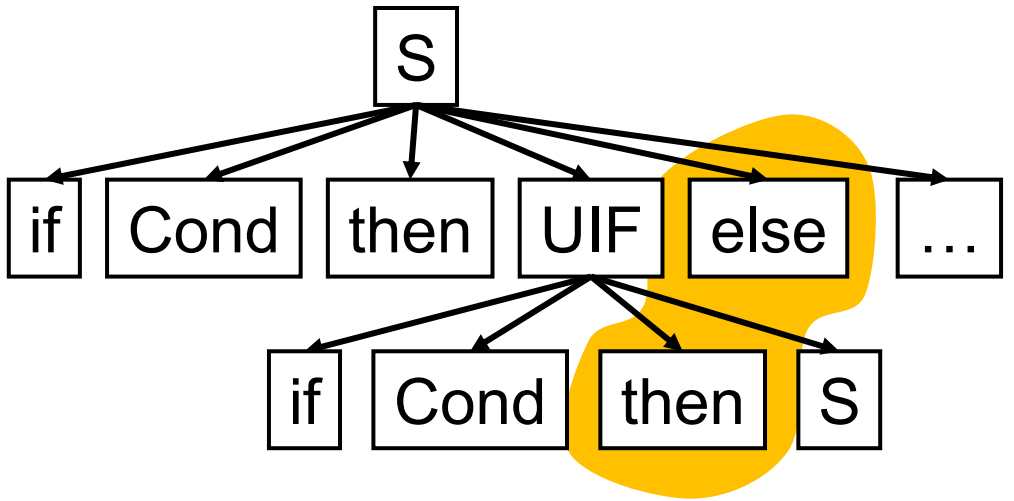




Removing Ambiguity in If/Then/Else

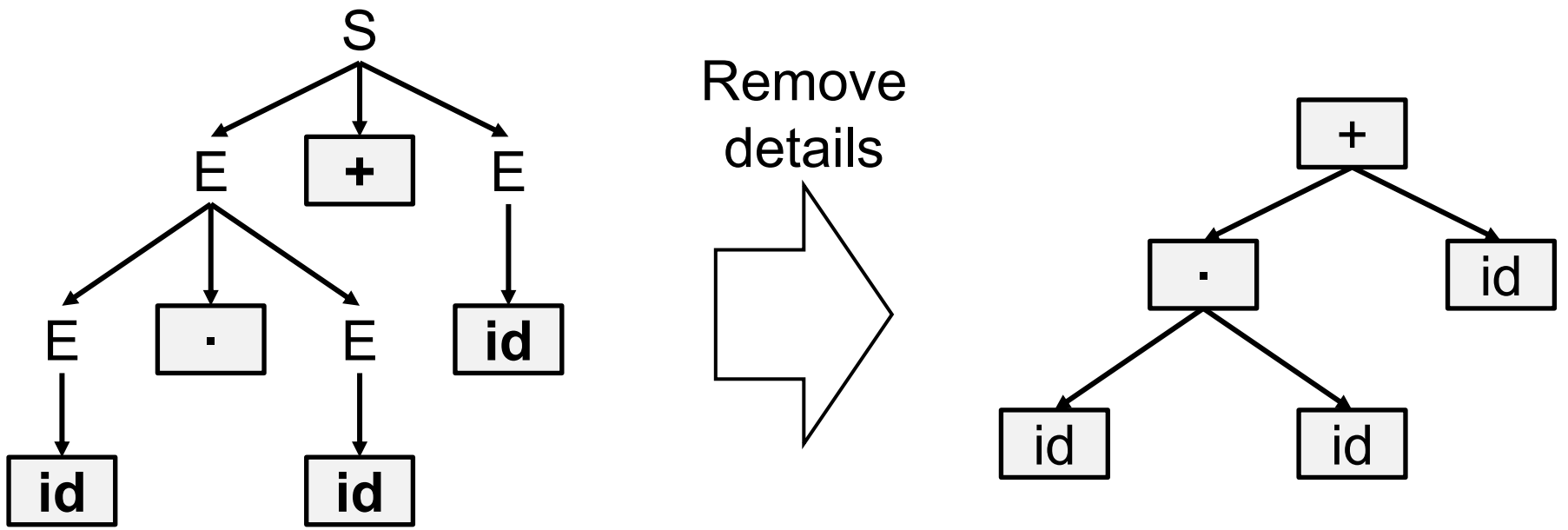
- **We should enforce "else" to match the closest then**

```
S ➔ MIF      // All "then"s are matched
  | UIF      // Some "then"s are unmatched

MIF ➔ if Cond then MIF else MIF
    | Others

UIF ➔ if Cond then S
    | if Cond then MIF else UIF

// Consider if Cond then UIF else S
```



Abstract Syntax Tree (AST)

- **AST discards unneeded information for syntax analysis (Removes non-terminals)**



Parse Tree **Abstract Syntax Tree**

Error Handling

- **Purpose of the compiler is error handling**
  - To detect non-valid programs
  - To translate the non-valid programs to the valid ones

- **Error handler should**
  - Report errors accurately and clearly
  - Recover quickly from an error
  - Not slow down compilation of valid code

Why Error Handling?

- **Back in the days, the compiler was extremely slow**
  - It may take a single day simply to fix a minor typo
  - It needed a complex error handling procedure
    - Identify as many mistakes as possible within a single compilation
    - Automatically correct straightforward errors during the compilation procedure

- **Nowadays, we do not need complex error handling procedure**
  - Quick recompilation
  - Users tend to correct few errors at once
  - Does not need a complex error recovery procedure

Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid strings of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

Specification, Recognition, and Automation in Parser

- **Specification: how to specify valid strings of tokens?**
  - Context-free grammars (CFG)
- **Recognition: how to recognize the specified patterns?**
  - Parse tree and abstract syntax tree (AST)
- **Automation: how to generate parse tree from CFG?**
  - Top-down and bottom-up parsing
  - Automatic generation tool (bison)

Top-Down Parsing

- **Construct a leftmost derivation of string while reading the token stream**

S ➔ E + S | E
E ➔ num | (S)

CFG

Partly-derived String	parsed part unparsed part
➔ E + S	(1+2+(3+4))+5
➔ (S) + S	(1+2+(3+4))+5
➔ (E+S)+S	(1+2+(3+4))+5
➔ (1+S)+S	(1+2+(3+4))+5
➔ (1+E+S)+S	(1+2+(3+4))+5
➔ (1+2+S)+S	(1+2+(3+4))+5
➔ (1+2+E)+S	(1+2+(3+4))+5
➔ (1+2+(S))+S	(1+2+(3+4))+5
➔ ...	

Derivation Order

Recursive Descent Parsing

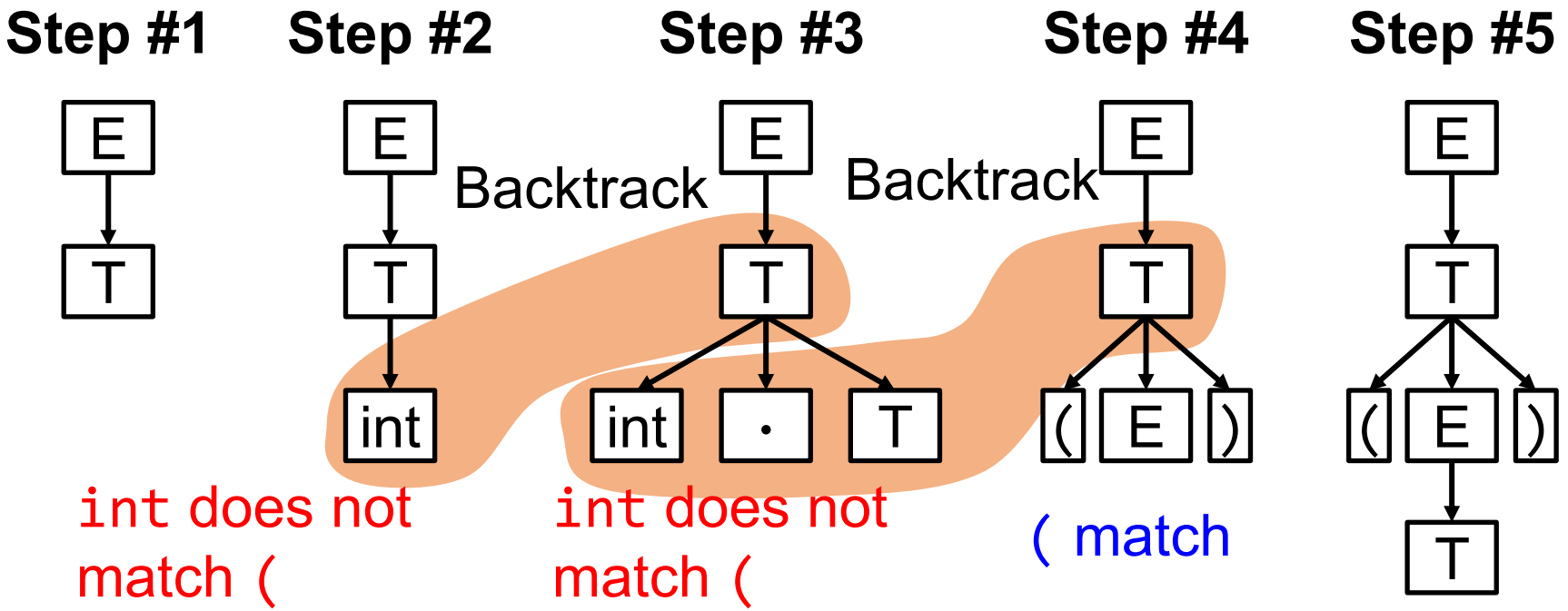
- **Try out rules in order and backtrack if the production does not generate proper token**

CFG

```
E ➔ T|T+E
T ➔ int|int·T|(E)
```

Input Tokens

```
(int)
```



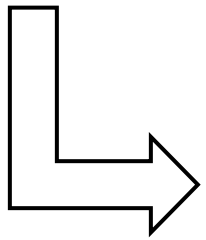
Formal Notation

- **There is a formal definition of the general top-down parsing**
- **`next` indicates a pointer to the input token**
- **If the derivation generates a terminal do the following**
  - `bool term(token) {return *next++ == token;}`
- **The set of productions for a non-terminal S and the nth element for the set are defined as**
  - `bool S() {…}`
  - `bool` S_n`() {…}`

Top-Down Parsing

CFG

```
E → T|T+E
T → int|int·T|(E)
```



Parsing Function

```
bool E1() {return T();}
bool E2() {return T() && term("+") && E();}
bool E() {
  Token *save = next;
  return (next = save, E1()) || (next = save, E2());}
bool T1() {return term("int");}
bool T2() {return term("int") && term("·") && T();}
bool T3() {return term("(") && E() && term(")");}
bool T() {
  Token *save = next;
  return (next = save, T1()) || (next = save, T2())
     || (next = save, T3());}
```

Backtracking

Backtracking

Limitation

- **Top-down parsing may fail to work properly if there can be multiple matches**

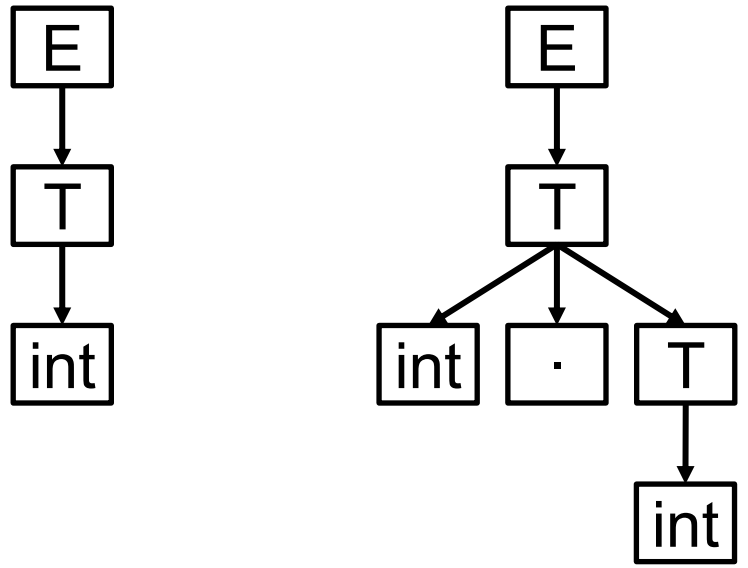
CFG

```
E ➔ T|T+E
T ➔ int|int·T|(E)
```

Input Stream

```
int · int
```

Parse Tree



Incorrect **Correct**

Making a Grammar LL(1)

- **LL(1) eliminates the multiple matches in top-down parsing**
  - Left-to-right scanning, Leftmost derivation, and 1 look-ahead symbol
    - We have to read one token to determine which production to apply

- **Left factoring: factor common prefix and add a non-terminal S'**
  - S ➔ E + S | E is converted as follows
  - S ➔ ES' and S' ➔ +S | ε

- **Convert left recursion to right recursion**
  - Example left recursion: S ➔ Sa (infinite derivation)
  - We use the term S ➔* Sa (for some a) to indicate the left recursion

Converting Left Recursion

- **Representative example: S generates all strings starting with a "b" followed by any number of "a"**
  - S ➔ Sa | b is converted to
  - S ➔ bS' and S' ➔ aS' | ε

- **General form: S generates all strings starting with one of b_1 ~ b_n followed by any number of a_1 ~ a_n**
  - S ➔ Sa_1 | Sa_2 | … | Sa_n | b_1 | b_2 | … | b_n is converted to
  - S ➔ b_1S' | b_2S' | … | b_nS' and S' ➔ a_1S' | a_2S' | … | a_nS' | ε

Class Exercise

1. A ➔ AAa | b

2. E ➔ E+E | ExE | a

Predictive Parsing

- **Predictive parsing applies a single production "without backtracking"**
  - The choice of production may affect the functionality ☹
- **LL(1) grammar can apply "at most a single production"**
  - We can apply predictive parsing without affecting the functionality

Predictive Parsing for LL(1)

- **Use a parsing table with non-terminals times terminals**

CFG
S ➔ ES’
S’ ➔ +S | ε
E ➔ num | (S)

Partly-derived String	parsed part unparsed part
➔ ES’	(1+2+(3+4))+5
➔ (S)S’	(1+2+(3+4))+5
➔ (ES’)S’	(1+2+(3+4))+5
➔ (1S’)S’	(1+2+(3+4))+5
➔ (1+S)S’	(1+2+(3+4))+5

	num	+	(	)	$ (EOF)
S	ES’		ES’		
S’		+S		ε	ε
E	num		(S)		

Parser Implementation

- **We can use the table to implement parsers**

- **In the example, we need to implement three functions:**
  - parse_S(), parse_S'(), and parse_E()

	num	**+**	**(**	**)**	**$ (EOF)**
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Recursive-Descent Parser

```
void parse_S() {
    switch (token) {
      case num: parse_E(); parse_S'(); return;
      case '(': parse_E(); parse_S'(); return;
      default: ParseError(); }}
```

	num	+	(	)	$ (EOF)
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Recursive-Descent Parser

```
void parse_S'() {
    switch (token) {
      case '+': token = input.next(); parse_S(); return;
      case ')': return;
      case EOF: return;
      default: ParseError(); }}
```

	num	+	(	)	$ (EOF)
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Recursive-Descent Parser

```
void parse_E() {
    switch (token) {
      case number: token = input.next(); return;
      case ‘(‘: token = input.next(); parse_S();
                if (token != ‘)’) ParseError();
                token = input.next(); return;
      default: ParseError(); }}
```

	num	+	(	)	$ (EOF)
S	ES’		ES’		
S’		+S		ε	ε
E	num		(S)		

How to Construct Parsing Tables?

- **We need an automatic method to generate a predictive parse table from a grammar**

- **There are three important traits to generate parse tables**
  - x is Nullable if it can derive an empty string
    - If x ➔* ε exists
  - First(x) is a set of terminals that can be derived in the first position
    - First(x) = {t | x ➔* ty}
  - Follow(x) is a set of terminals that appears after α in at least one of the derivations
    - Follow(x) = {t | S ➔* yxtz} (S is the start symbol)

Computing Nullable

- **Nullable definition: {x | x ➔* ε}**
- **Identify two cases of nullability**
  - Case 1) ε is nullable
  - Case 2) For a production X ➔ $A_1A_2…A_n$ is nullable if $A_1A_2…A_n$ are nullable
    - As A_1 is nullable, X ➔* $A_2…A_n$
    - As A_2 is nullable, X ➔* $A_3…A_n$
    - As A_3 is nullable, X ➔* $A_4…A_n$
    - As A_4 is nullable, X ➔* $A_5…A_n$
    - …
    - As An is nullable, X ➔* ε

    - Ex) X is nullable if X ➔ ε

Computing First

- **First definition: First(x) = {t | x ➔* ty}**
- **Computing First involves two different steps**
  - Step 1) First(t) = { t } (t is a terminal)
  - Step 2) For a production x ➔ $A_1A_2…A_n$ where $A_1…A_{i-1}$ are nullable, First(x) += First(A_i)
    - As A_1 is nullable, x ➔* $A_2A_3…A_n$
    - …
    - As A_{i-1} is nullable, x ➔* $A_iA_{i+1}…A_n$
    - This implies that, x ➔* $tA_i'A_{i+1}…A_n$ for t in First(A_i) = {t | A_i ➔* tA_i'}
- **Repeat until there is no change**

Computing Follow

- **Follow definition: Follow(x) = {t | S ➔* yxtz} (S → start symbol)**
- **Computing Follow involves three different steps**
  - Step 1) Follow(S) = { $ } (S is the start symbol)
  - Step 2) For a production x ➔ $A_1A_2\ldots A_n$ where $A_{i+1}A_{i+2}\ldots A_n$ are nullable, Follow(A_i) += Follow(x)
    - As $A_{i+1}A_{i+2}\ldots A_n$ are nullable, x ➔* $A_1A_2\ldots A_i$
    - Follow(x) = {t | S ➔* yxtz} → {S ➔* y $A_1A_2\ldots A_i$ tz} → Follow(A_i) += Follow(x)
  - Step 3) For a production x ➔ $A_1A_2\ldots A_n$ where $A_{i+1}A_{i+2}\ldots A_{j-1}$ are nullable, Follow(A_i) += First(A_j)
    - As $A_{i+1}A_{i+2}\ldots A_{j-1}$ are nullable, x ➔* $A_1A_2\ldots \mathbf{A_iA_j}A_{j+1}\ldots A_n$
    - First(A_j) = {t | A_j ➔* ty} → x ➔* $A_1A_2\ldots \mathbf{A_i t}yA_{j+1}\ldots A_n$ → Follow(A_i) += First(A_j)

Combine Together

```
// Initialize
for each symbol X
      First(X) = {}, Follow(X) = {}, Nullable(X) = false
Follow(S) = {$}
for each terminal t
      First(t) = {t}
Nullable(ε) = true
// Iterate
Repeat
      for each production X ➔ A1A2…An
            if all Ai are Nullable:       Nullable(X) = True
            if A1…Ai-1 are Nullable:      First(X) += First(Ai)
            if Ai+1…An are Nullable:      Follow(Ai) += Follow(X)
            if Ai+1…Aj-1 are Nullable:    Follow(Ai) += First(Aj)
Until there is no change in First, Follow, and Nullable
```

Nullable, First, and Follow Example

CFG
S ➔ ES’
S’ ➔ +S | ε
E ➔ num | (S)

	First	Follow	Nullable
S			
S’			
E			

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

```
for each symbol X: First(X) = {}, Follow(X) = {},
Nullable[X] = false
Follow(S) = {$}
for each terminal t: First(t) = {t}
Nullable[ε] = true
```

	First	Follow	Nullable
S	{}	{$}	False
S'	{}	{}	False
E	{}	{}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{} +First(E) = {}	{$}	False
S'	{}	{} + Follow(S) = {$}	False
E	{}	{} + First(S') = {}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{} + First(E) = {}	{$} + Follow(S') = {$}	False
S'	{} + First(+) = {+}	{$} + Follow(S) = {$}	True
E	{}	{} + First(S') = {+}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{} + First(E) = {num, (}	{$} + Follow(S') + First()) = {$,)}	False
S'	{+} + First(+) = {+}	{$} + Follow(S) = {$,)}	True
E	{} + First(num) + First(() = {num, (}	{+} + First(S') = {+}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{num, (} + First(E) = {num, (}	{$,)} + Follow(S') + First()) = {$,)}	False
S'	{+} + First(+) = {+}	{$,)} + Follow(S) = {$,)}	True
E	{num, (} + First(num) + First(() = {num, (}	{+} + First(S') + Follow(S) = {$,), +}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{num, (} + First(E) = {num, (}	{$,)} + Follow(S') + First()) = {$,)}	False
S'	{+} + First(+) = {+}	{$,)} + Follow(S) = {$,)}	True
E	{num, (} + First(num) + First(() = {num, (}	{$,), +} + First(S') + Follow(S) = {$,), +}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{num, (} + First(E) = {num, (}	{$,)} + Follow(S') + First()) = {$,)}	False
S'	{+} + First(+) = {+}	{$,)} + Follow(S) = {$,)}	True
E	{num, (} + First(num) + First(() = {num, (}	{$,), +} + First(S') + Follow(S) = {$,), +}	False

Nullable, First, and Follow Example

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
S	{num, (}	{$,)}	False
S'	{+}	{$,)}	True
E	{num, (}	{$,), +}	False

Putting It Together

- **We can use First/Follow to generate the table (i.e., we should search for the lookahead)**
  - For a production in X ➔ $A_1A_2…A_n$, for each terminal t ∈ First($A_1A_2…A_n$), table[X, t] = $A_1A_2…A_n$
  - How to calculate First($A_1A_2…A_n$) ?
    - If Nullable(A_1) == False, return First(A_1)
    - If Nullable(A_1) == True, return First(A_1) + First($A_2A_3…A_n$)
  - If all A_i is nullable, for each t ∈ Follow(X) do, table[X, t] = $A_1A_2…A_n$
    - Given that the partly derived string is XtY
    - If we apply X ➔ $A_1…A_n$, the string becomes $A_1…A_n$ tY
    - As Ai are nullable, the string becomes tY and we have t at the left-most terminal

Putting It Together

	First	Follow	Nullable
S	{num, (}	{$,)}	False
S'	{+}	{$,)}	True
E	{num, (}	{$,), +}	False

CFG
S ➔ ES'
S' ➔ +S | ε
E ➔ num | (S)

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG
E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

	First	Follow	Nullable
E	{}	{$}	False
T	{}	{}	False
X	{}	{}	False
Y	{}	{}	False

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG

E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

for a production X ➔ $A_1A_2…A_n$

if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
E	{} + First(T) = {}	{$}	False
T	{}	{} + First(X) = {}	False
X	{}	{} + Follow(E) = {$}	False
Y	{}	{}	False

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG
E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
E	{} + First(T) = {(, int}	{$} + First()) = {$,)}	False
T	{} + First(() + First(int) = {(, int}	{} + First(X) = {}	False
X	{}	{$} + Follow(E) = {$,)}	False
Y	{}	{} + Follow(T) = {}	False

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG
E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
E	{(, int} + First(T) = {(, int}	{$,)} + First()) + Follow(X) = {$,)}	False
T	{(, int} + First(() + First(int) = {(, int}	{} + First(X) = {+}	False
X	{} + First(+) = {+}	{$,)} + Follow(E) = {$,)}	True
Y	{}	{} + Follow(T) = {+}	False

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG
E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
E	{(, int} + First(T) = {(, int}	{$,)} + First()) + Follow(X) = {$,)}	False
T	{(, int} + First(() + First(int) = {(, int}	{+} + First(X) + Follow(Y) = {+}	False
X	{+} + First(+) = {+}	{$,)} + Follow(E) = {$,)}	True
Y	{} + First(*) = {*}	{+} + Follow(T) = {+}	True

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG

E ➔ TX

T ➔ (E) | int Y

X ➔ + E | ε

Y ➔ * T | ε

for a production X ➔ $A_1A_2…A_n$

if all Ai are Nullable: Nullable(X) = True

if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)

if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)

if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
E	{(, int} + First(T) = {(, int}	{$,)} + First()) + Follow(X) = {$,)}	False
T	{(, int} + First(() + First(int) = {(, int}	{+} + First(X) + Follow(Y) + Follow(E) = {$,), +}	False
X	{+} + First(+) = {+}	{$,)} + Follow(E) = {$,)}	True
Y	{*} + First(*) = {*}	{+} + Follow(T) = {$,), +}	True

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG
E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

for a production X ➔ $A_1A_2…A_n$
if all Ai are Nullable: Nullable(X) = True
if $A_1…A_{i-1}$ are Nullable: First(X) += First(A_i)
if $A_{i+1}…A_n$ are Nullable: Follow(A_i) += Follow(X)
if $A_{i+1}…A_{j-1}$ are Nullable: Follow(A_i) += First(A_j)

	First	Follow	Nullable
E	{(, int} + First(T) = {(, int}	{$,)} + First()) + Follow(X) = {$,)}	False
T	{(, int} + First(() + First(int) = {(, int}	{$,), +} + First(X) + Follow(Y) + Follow(E) = {$,), +}	False
X	{+} + First(+) = {+}	{$,)} + Follow(E) = {$,)}	True
Y	{*} + First(*) = {*}	{+} + Follow(T) = {$,), +}	True

Class Exercise

- **Compute First / Follow and generate parse table entries**

CFG
E ➔ TX
T ➔ (E) | int Y
X ➔ + E | ε
Y ➔ * T | ε

	First	Follow	Nullable
E	{(, int}	{$,)}	False
T	{(, int}	{+, $,)}	False
X	{+}	{$,)}	True
Y	{*}	{+, $,)}	True

	(	)	int	+	*	$
E	TX		TX			
T	(E)		int Y			
X		ε		+E		ε
Y		ε		ε	*T	ε