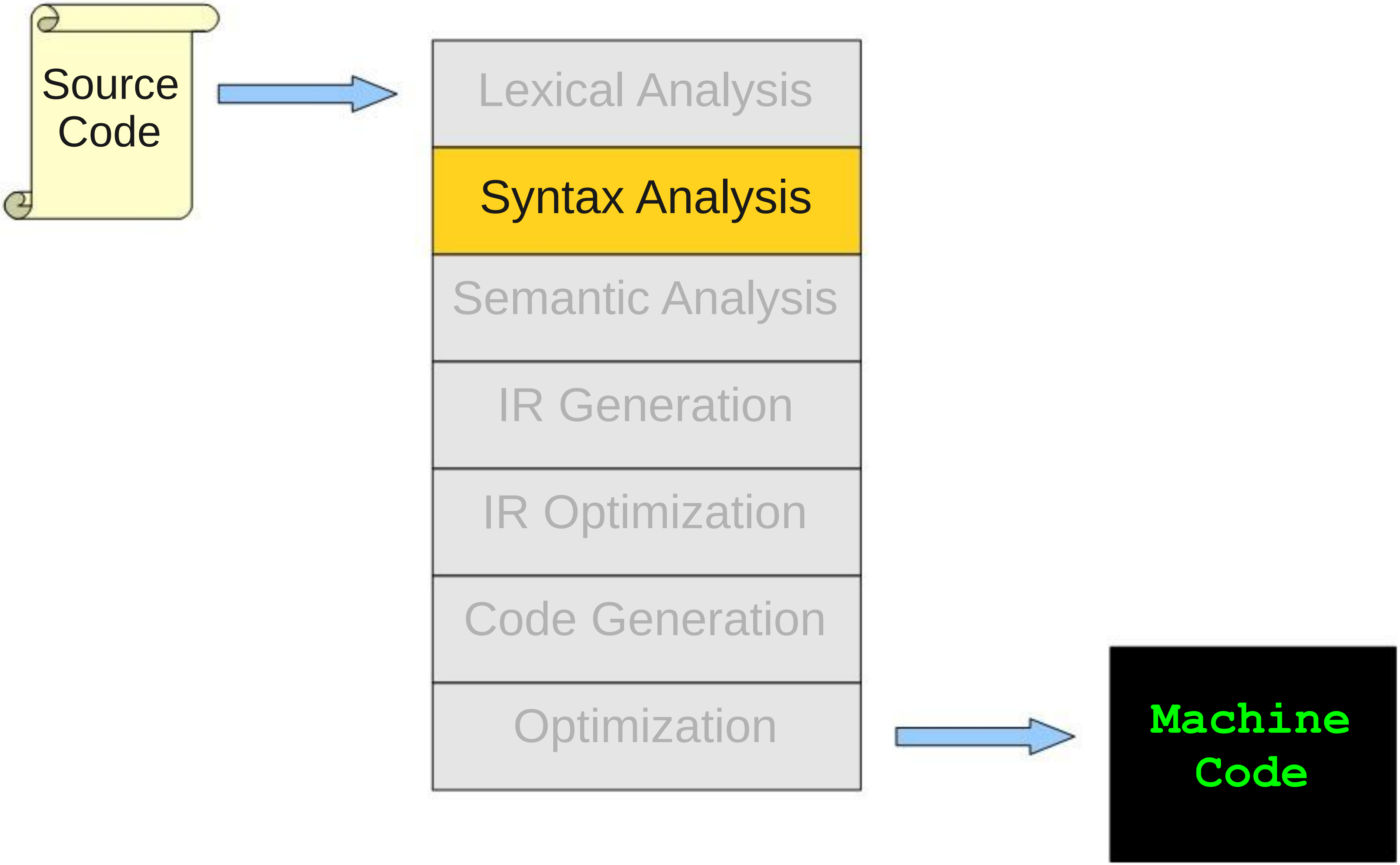
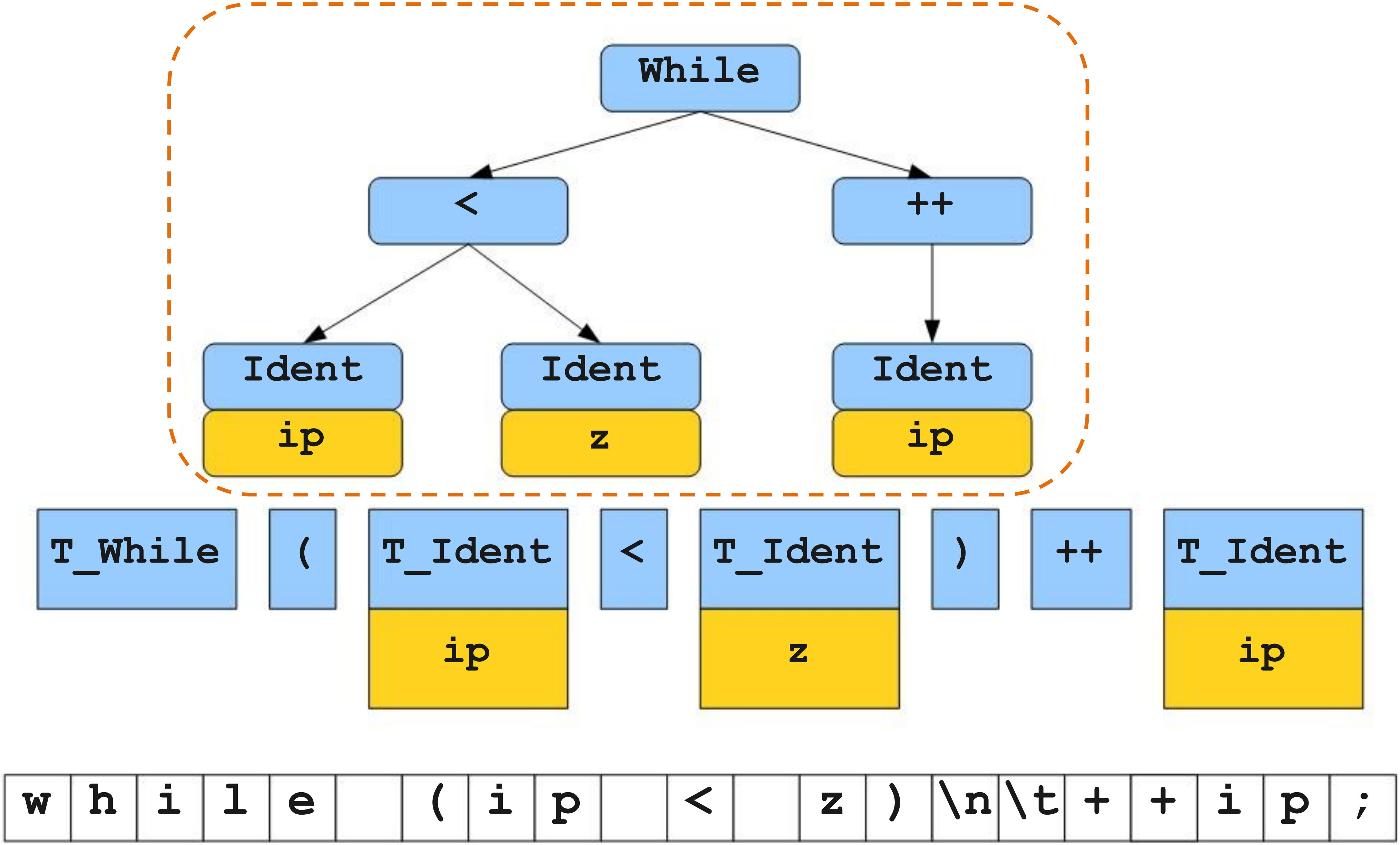
Syntax Analysis

Where We Are


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


While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e (i p < z) \n \t + + i p ;

```
while (ip < z)
    ++ip;
```

What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- Syntax Analysis: Find the **structure** (Syntax Tree) described by that series of tokens and report **errors** if those tokens do not properly encode a structure

Context-Free Grammars (CFGs)

- A tool for describing languages that is strictly more powerful than regular languages.

- A production form

  $A \rightarrow B_1 B_2 \ldots B_n$

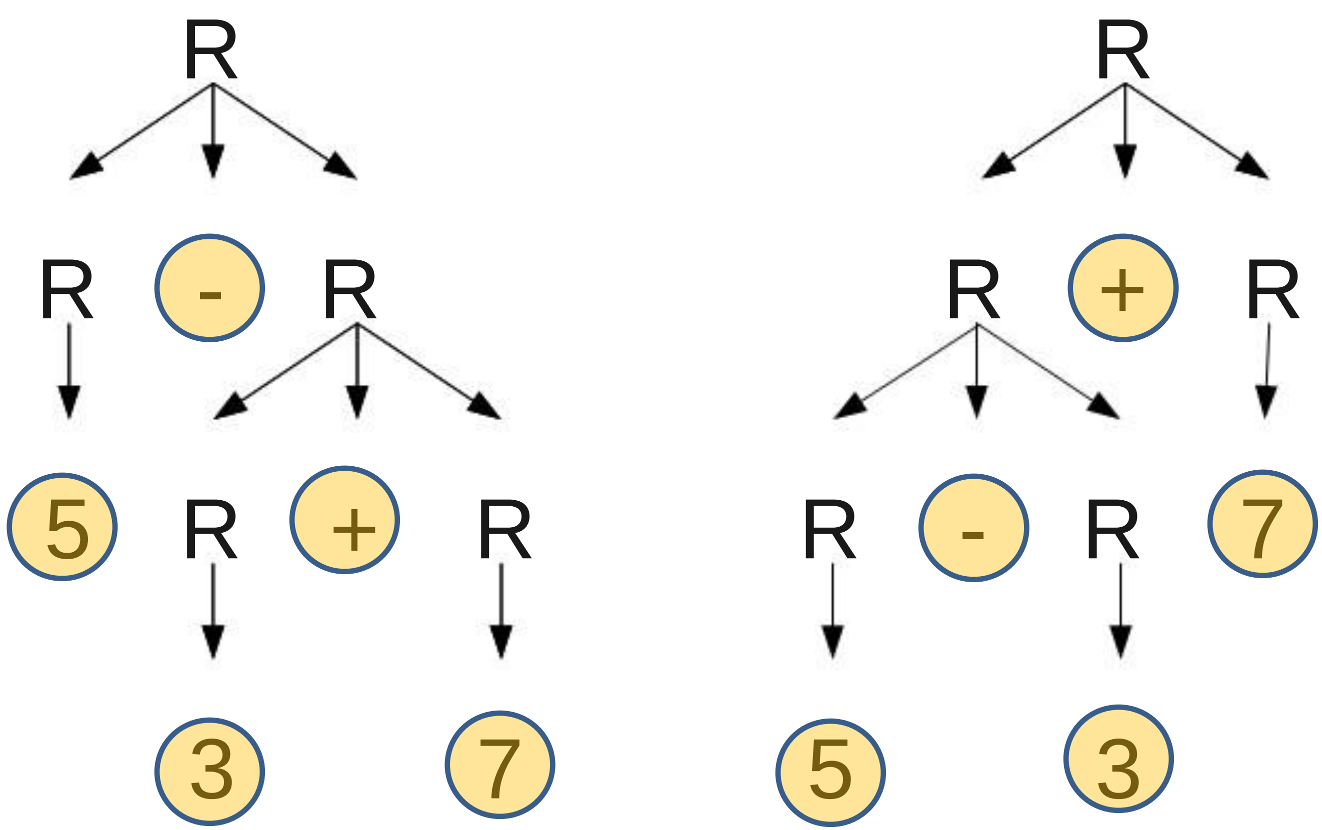
  A is a nonterminal
  B_i's are either terminals or nonterminals.

Example

```
BLOCK → STMT
      |  { STMTS }

STMTS → ε
      |  STMT STMTS

STMT  → EXPR;
      |  if (EXPR) BLOCK
      |  while (EXPR) BLOCK
      |  do BLOCK while (EXPR);
      |  BLOCK
      |  ...

EXPR  → identifier
      |  constant
      |  EXPR + EXPR
      |  EXPR - EXPR
      |  EXPR * EXPR
```

Ambiguity

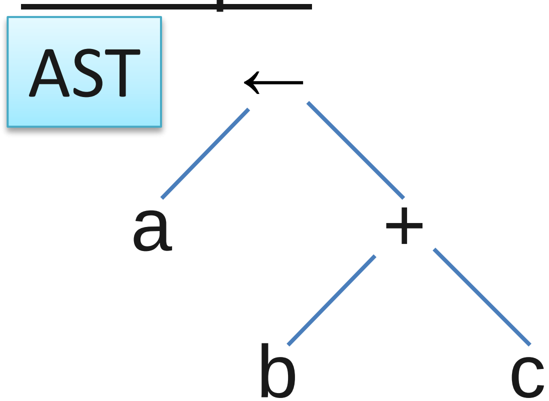
- A CFG is said to be **ambiguous** if there is at least one string with two or more derivations.
- R → **int** | R + R | R - R

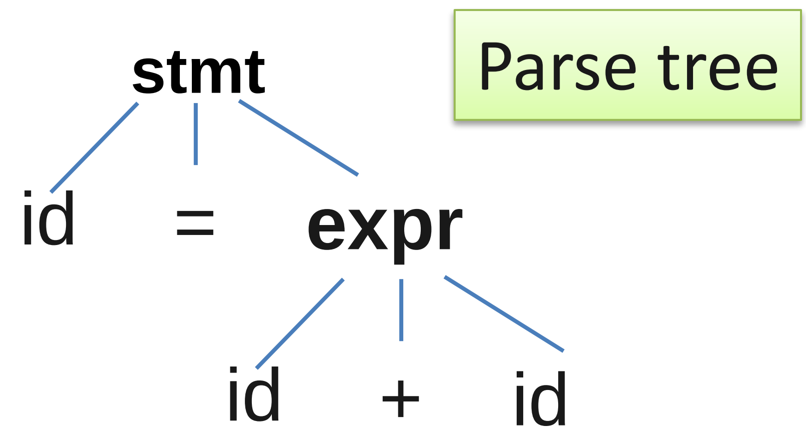


Abstract Syntax Trees (ASTs)

- Tree structure encoding the **logical structure** of a piece of code.

  Example a = b + c





- As the input is parsed, associate code creates AST with each production.
- This is called a **syntax-directed translation or semantic action**.

Syntax-directed translation (in Bison)

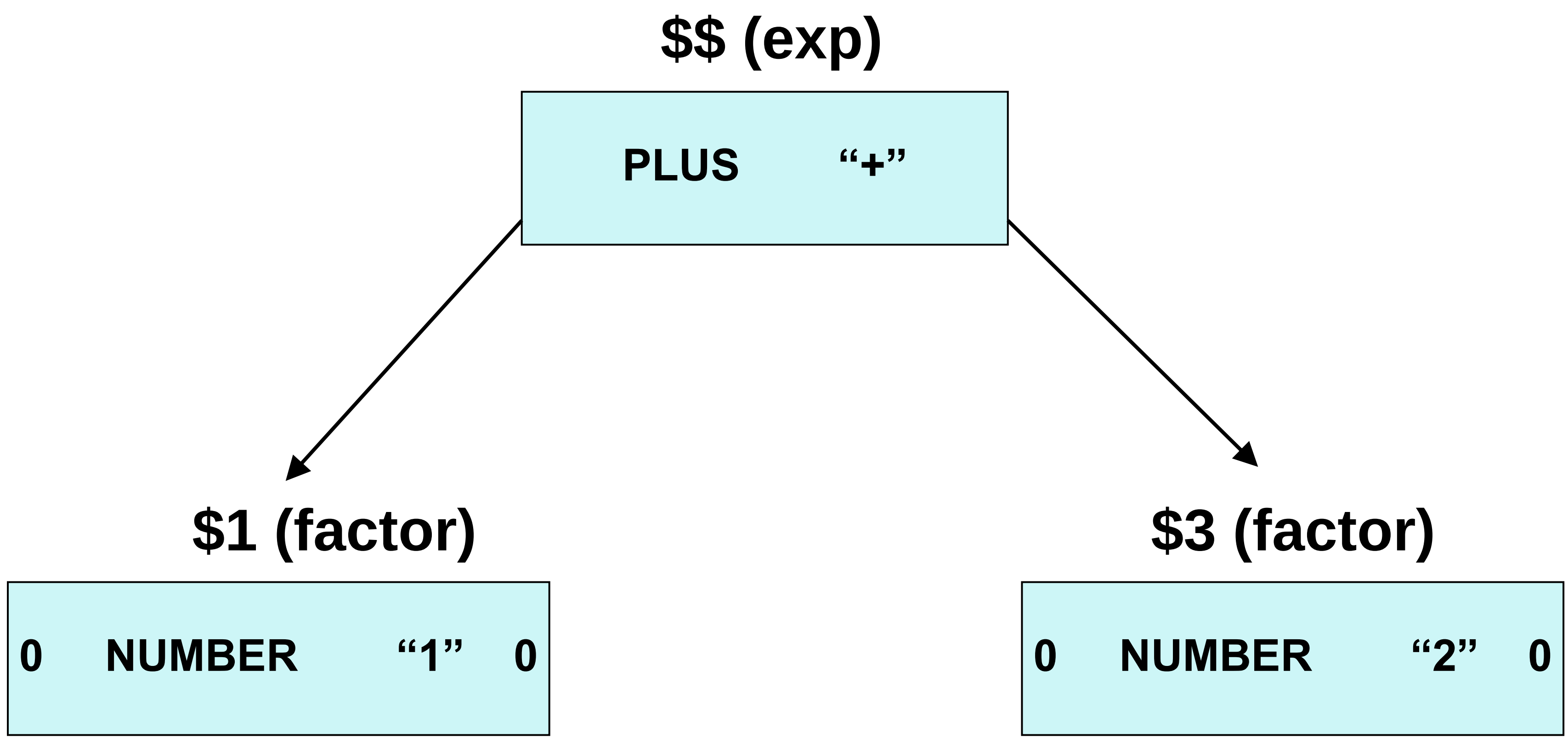
```
exp: factor                 { $$ = $1 }
   | factor PLUS factor
                            { $$ = makeNode($1, $3, PLUS, "+") }
   | factor MINUS factor
                            { $$ = makeNode($1, $3, MINUS, "-") }

factor : NUMBER
      { $$ = makeNode(0,0, NUMBER, (char *)yylval) }
```

Node

***Left**	**Token**	**Lexeme**	***Right**

Syntax-directed translation (in Bison)

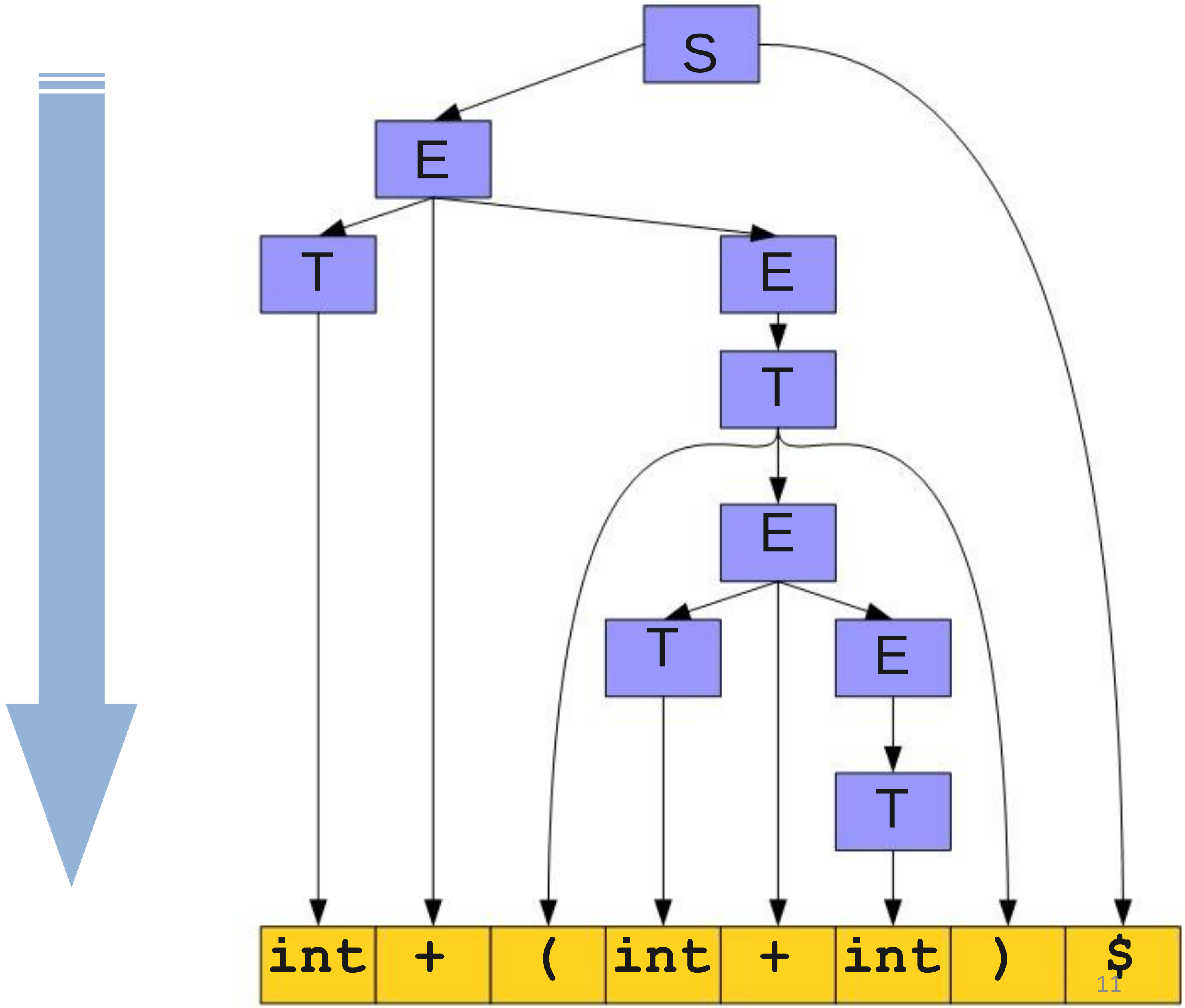
Example: AST for the input string “1+2”



Top-Down Parsing

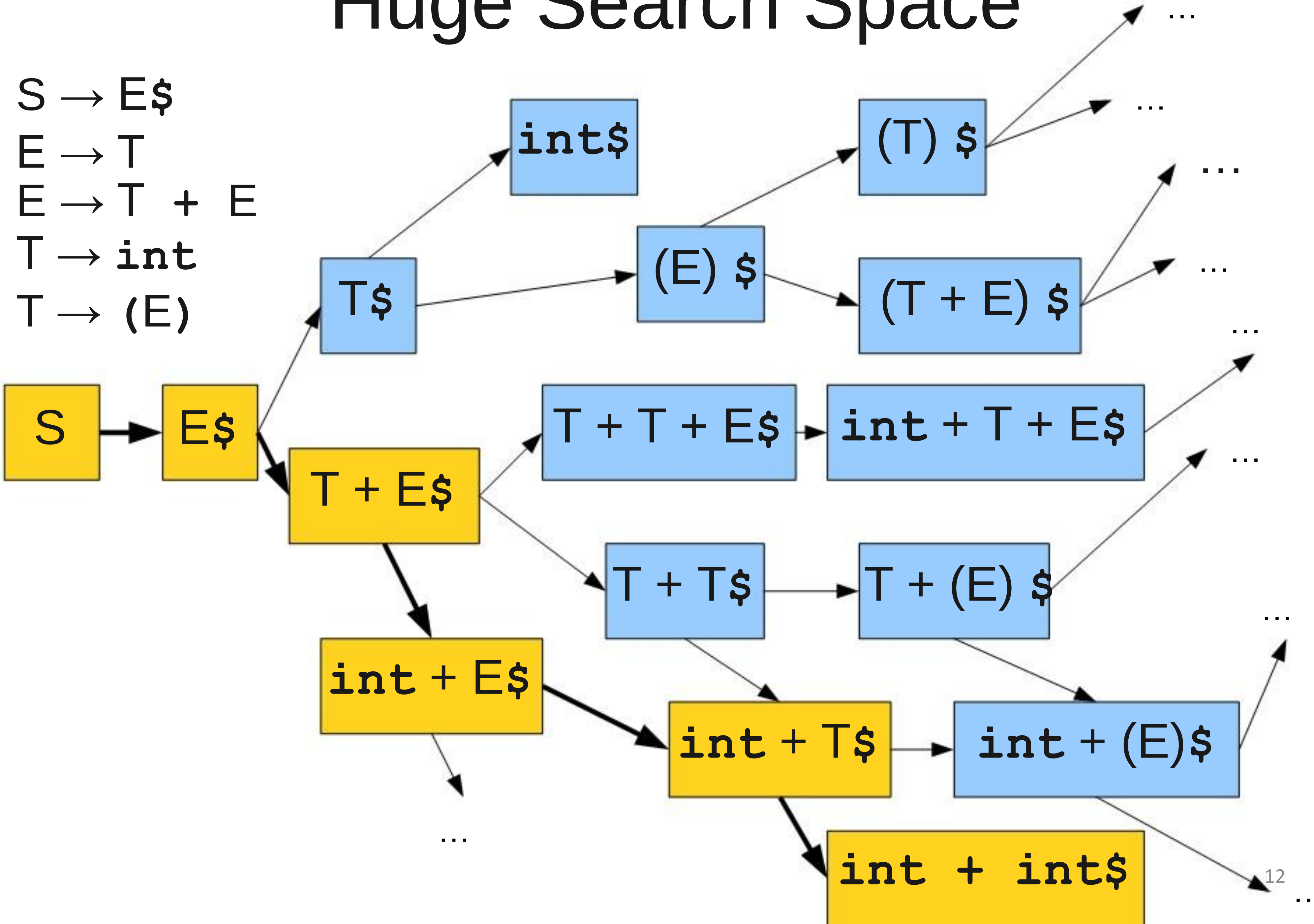
Top-Down Parsing

$S \rightarrow E\$$
$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**



Huge Search Space

S → E$
E → T
E → T + E
T → int
T → (E)



BFS is Slow

- **Enormous** time and memory usage:
  - Lots of **wasted effort**:
    - Generates a lot of sentential forms that couldn't possibly match.
  - High **branching factor**:
    - Each sentential form can expand in (potentially) many ways for each nonterminal it contains.

Predictive Parsing

- There is another class of parsing algorithms called **predictive** algorithms (no backtracking).

- Idea: **Lookahead tokens**.
- **LL(1) :** Top-down, predictive parsing
  - **L**: Left-to-right scan of the tokens
  - **L**: Leftmost derivation.
  - **(1)**: One token of lookahead

- When expanding a nonterminal, we predict the production to use by looking at the next token of the input.

LL(1) Parse Tables

$S \rightarrow E\$$
$E \rightarrow$ **int**
$E \rightarrow$ **(**E Op E**)**
$Op \rightarrow$ **+**
$Op \rightarrow$ *****

	int	(	)	+	*	$
S	E$	E$				
E	int	(E Op E)				
Op				+	*	

We will learn how to create the LL(1) parsing table soon.

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	**(int + (int * int))$**
Parsing stack	Remaining input

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

S	(int + (int * int))$

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int + (int * int))$
E$	(int + (int * int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	`(int + (int * int))$`
E$	`(int + (int * int))$`
(E Op E) $	`(int + (int * int))$`
E Op E) $	`int + (int * int))$`

Pop '(' out of the stack and advance to the next token

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	`(int + (int * int))$`
E$	`(int + (int * int))$`
(E Op E) $	`(int + (int * int))$`
E Op E) $	`int + (int * int))$`

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E`$`
2. E → `int`
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `*`

S	`(int + (int * int))$`
E`$`	`(int + (int * int))$`
`(`E Op E`)$`	`(int + (int * int))$`
E Op E`)$`	`int + (int * int))$`
`int` Op E`)$`	`int + (int * int))$`
Op E`)$`	`+ (int * int))$`

Pop 'int' out of the stack and advance to the next token

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$

Pop '+' out of the stack and advance to the next token

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$
(E Op E)) $	(int * int))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$
(E Op E)) $	(int * int))$
E Op E)) $	int * int))$

Pop '(' out of the stack and advance to the next token

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$
(E Op E)) $	(int * int))$
E Op E)) $	int * int))$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$
(E Op E)) $	(int * int))$
E Op E)) $	int * int))$
int Op E)) $	int * int))$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$

Pop ‘int’ out of the stack and advance to the next token

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$
(E Op E)) $	(int * int))$
E Op E)) $	int * int))$
int Op E)) $	int * int))$
Op E)) $	* int))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E) $	(int + (int * int))$
E Op E) $	int + (int * int))$
int Op E) $	int + (int * int))$
Op E) $	+ (int * int))$
+ E) $	+ (int * int))$
E) $	(int * int))$
(E Op E)) $	(int * int))$
E Op E)) $	int * int))$
int Op E)) $	int * int))$
Op E)) $	* int))$
* E)) $	* int))$
E)) $	int))$

Pop '*' out of the stack and advance to the next token

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$
$	$

Accepted !

Error Detection

S	(int (int))$

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int (int))$
E$	(int (int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

S	`(int (int))$`
E`$`	`(int (int))$`

1. S → E**`$`**
2. E → **`int`**
3. E → **`(`**E Op E**`)`**
4. Op → **`+`**
5. Op → **`*`**

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

Error Detection

S	(int (int))$
E$	(int (int))$
(E Op E)$	(int (int))$

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	`(int (int))$`
E$	`(int (int))$`
(E Op E)$	`(int (int))$`
E Op E)$	`int (int))$`

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int (int))$
E$	(int (int))$
(E Op E)$	(int (int))$
E Op E)$	int (int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	**(int (int))$**
E**$**	**(int (int))$**
(E Op E**)$**	**(int (int))$**
E Op E**)$**	**int (int))$**
int Op E**)$**	**int (int))$**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → *****

S	(int (int))$
E$	(int (int))$
(E Op E)$	(int (int))$
E Op E)$	int (int))$
int Op E)$	int (int))$
Op E)$	(int))$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → *

S	(int (int))$
E$	(int (int))$
(E Op E)$	(int (int))$
E Op E)$	int (int))$
int Op E)$	int (int))$
Op E)$	(int))$

Rejected !

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Self review

The LL(1) Algorithm

- Given an LL(1) parsing table T and input w:
- Initialize a stack containing S.
- Repeat until the stack is just **$**:
  - Let the next character of *w* be *c*.
  - If the top of the stack is a terminal *t*:
    - If *c* and *t* don't match, report an error.
    - Otherwise consume the character *c* and pop *t* from the stack.
  - Otherwise, the top of the stack is a nonterminal *A*:
    - If T[A, *c*] is undefined, report an error.
    - Replace the top of the stack with T[A, c].

LL(1) Grammar

- In order to use LL(1) algorithm, our CFG grammar must be LL(1).
- We can determine which production to be used by looking at only the current token.

Example:

```
STMT → if EXPR then STMT
     | while EXPR do STMT
     | EXPR ;

EXPR → TERM → id
     | zero? TERM
     | not EXPR
     | ++ id
     | -- id

TERM → id
     | constant
```

Constructing LL(1) Parsing Table

Table T

		Token c	
Var. A		**T[A, c]**	

- T[A, c] should be a production

  $A \longrightarrow A_1\ A_2\ \ldots\ A_n$

  if A_1 ultimately derives something starting with c.

- The systematic way to filling in table entries is to use notions of **First sets** and **Follow sets**.

FIRST Sets

- Definition: $\text{FIRST}(A) = \{\ t \mid A \rightarrow^{*} tw\ \}$
  - The set of tokens that appear first in the production of A.

- Set $T[A, c] = A_1\ A_2\ \ldots\ A_n$ if $c \in \text{FIRST}(A_1)$

  and $A \rightarrow A_1 A_2 \ldots A_n$

Computing FIRST Sets

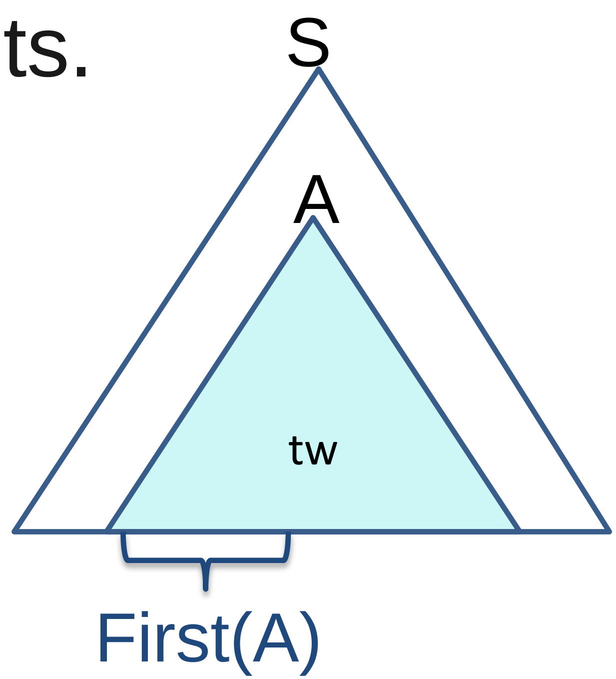
- Initially, for each production $A \to tw$,

  FIRST(A) = { t | $A \to tw$ } *// t is a terminal symbol*

- Then, for each $A \to Bw$, iteratively compute

  FIRST(A) = FIRST(A) ∪ FIRST(B)

- When no changes occur, the resulting sets are the FIRST sets.



Iterative FIRST Computations

STMT → **`if`** EXPR **`then`** STMT
| **`while`** EXPR **`do`** STMT
| EXPR **`;`**

EXPR → TERM → **`id`**
| **`zero?`** TERM
| **`not`** EXPR
| **`++ id`**
| **`-- id`**

TERM → **`id`**
| **`constant`**

STMT	EXPR	TERM

Iterative FIRST Computations

STMT → **if** EXPR **then** STMT
| **while** EXPR **do** STMT
| EXPR ;

EXPR → TERM → **id**
| **zero?** TERM
| **not** EXPR
| **++ id**
| **-- id**

TERM → **id**
| **constant**

STMT	EXPR	TERM

`if`
`while`
	`zero?`
	`not`
	`++`
	`--`
		`id`
		`constant`

Iterative FIRST Computations

STMT → **if** EXPR **then** STMT
| **while** EXPR **do** STMT
| EXPR **;**

EXPR → TERM → **id**
| **zero?** TERM
| **not** EXPR
| **++ id**
| **-- id**

TERM → **id**
| **constant**

STMT	EXPR	TERM
if while zero? not ++ --	zero? not ++ --	id constant

Iterative FIRST Computations

STMT → **if** EXPR **then** STMT
| **while** EXPR **do** STMT
| EXPR ;

EXPR → TERM → **id**
| **zero?** TERM
| **not** EXPR
| **++ id**
| **-- id**

TERM → **id**
| **constant**

STMT	EXPR	TERM
if	zero?	id
while	not	constant
zero?	++	
not	--	
++	id	
--	constant	

EXPR has been changed.
Recompute First(STMT) again.

Iterative FIRST Computations

STMT → `if` EXPR `then` STMT
    | `while` EXPR `do` STMT
    | EXPR `;`

EXPR → TERM → `id`
    | `zero?` TERM
    | `not` EXPR
    | `++ id`
    | `-- id`

TERM → `id`
    | `constant`

STMT	EXPR	TERM
if	zero?	id
while	not	constant
zero?	++	
not	--	
++	id	
--	constant	
id		
constant		

Done !

From FIRST Sets to LL(1) Tables

STMT →		**if** EXPR **then** STMT	(1)
	|	**while** EXPR **do** STMT	(2)
	|	EXPR ;	(3)
EXPR →		TERM → **id**	(4)
	|	**zero?** TERM	(5)
	|	**not** EXPR	(6)
	|	**++ id**	(7)
	|	**-- id**	(8)
TERM →		**id**	(9)
	|	**constant**	(10)

STMT	EXPR	TERM
if **while** **zero?** **not** **++** **--** **id** **constant**	**zero?** **not** **++** **--** **id** **constant**	**id** **constant**

	if	**then**	**while**	**do**	**zero?**	**not**	**++**	**--**	→	**id**	**const**	;
STMT												
EXPR												
TERM												

From FIRST Sets to LL(1) Tables

STMT →		**if** EXPR **then** STMT	(1)
	|	**while** EXPR **do** STMT	(2)
	|	EXPR ;	(3)
EXPR →		TERM → **id**	(4)
	|	**zero?** TERM	(5)
	|	**not** EXPR	(6)
	|	**++ id**	(7)
	|	**-- id**	(8)
TERM →		**id**	(9)
	|	**constant**	(10)

STMT	EXPR	TERM
if	**zero?**	**id**
while	**not**	**constant**
zero?	**++**	
not	**--**	
++	**id**	
--	**constant**	
id		
constant		

	if	**then**	**while**	**do**	**zero?**	**not**	**++**	**--**	→	**id**	**const**	**;**
STMT	1		2		3	3	3	3		3	3	
EXPR												
TERM												

From FIRST Sets to LL(1) Tables

STMT → **if** EXPR **then** STMT (1)
| **while** EXPR **do** STMT (2)
| EXPR ; (3)

EXPR → TERM → **id** (4)
| **zero?** TERM (5)
| **not** EXPR (6)
| **++ id** (7)
| **-- id** (8)

TERM → **id** (9)
| **constant** (10)

STMT	EXPR	TERM
if **while** **zero?** **not** **++** **--** **id** **constant**	**zero?** **not** **++** **--** **id** **constant**	**id** **constant**

	if	**then**	**while**	**do**	**zero?**	**not**	**++**	**--**	→	**id**	**const**	**;**
STMT	1		2		3	3	3	3		3	3	
EXPR					5	6	7	8		4	4	
TERM												

From FIRST Sets to LL(1) Tables

STMT →		**if** EXPR **then** STMT	(1)
	|	**while** EXPR **do** STMT	(2)
	|	EXPR ;	(3)
EXPR →		TERM → **id**	(4)
	|	**zero?** TERM	(5)
	|	**not** EXPR	(6)
	|	**++ id**	(7)
	|	**-- id**	(8)
TERM →		**id**	(9)
	|	**constant**	(10)

STMT	EXPR	TERM
if **while** **zero?** **not** **++** **--** **id** **constant**	**zero?** **not** **++** **--** **id** **constant**	**id** **constant**

	if	**then**	**while**	**do**	**zero?**	**not**	**++**	**--**	→	**id**	**const**	;
STMT	1		2		3	3	3	3		3	3	
EXPR					5	6	7	8		4	4	
TERM										9	10	

FIRST Sets with ε

Number → Sign Digits
Digits → Digit | Digit Digits
Digit → **0 | 1 | 2 | … | 9**
Sign → **+ | – | ε**

Thing gets complicated if we allow ε-production in our grammar.

Number	Digits	Digit	Sign
		0 1 2 3 4 5 6 7 8 9	+ – ε

FIRST Sets with ε

Number → Sign Digits
Digits → **Digit** | **Digit** Digits
Digit → 0 | 1 | 2 | … | 9
Sign → + | – | ε

Number	Digits	Digit	Sign
	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9	+ – **ε**

FIRST Sets with ε

Number → **Sign** Digits
Digits → Digit | Digit Digits
Digit → **0 | 1 | 2 | … | 9**
Sign → **+ | – | ε**

Number	Digits	Digit	Sign
+	0	0	+
–	1	1	–
	2	2	ε
	3	3	
	4	4	
	5	5	
	6	6	
	7	7	
	8	8	
	9	9	

FIRST Sets with ε

Number → Sign **Digits**
Digits → Digit | Digit Digits
Digit → **0 | 1 | 2 | … | 9**
Sign → **+ | − | ε**

Since **Sign** can be empty, First(**Digits**) must be also included in First(**Number**).

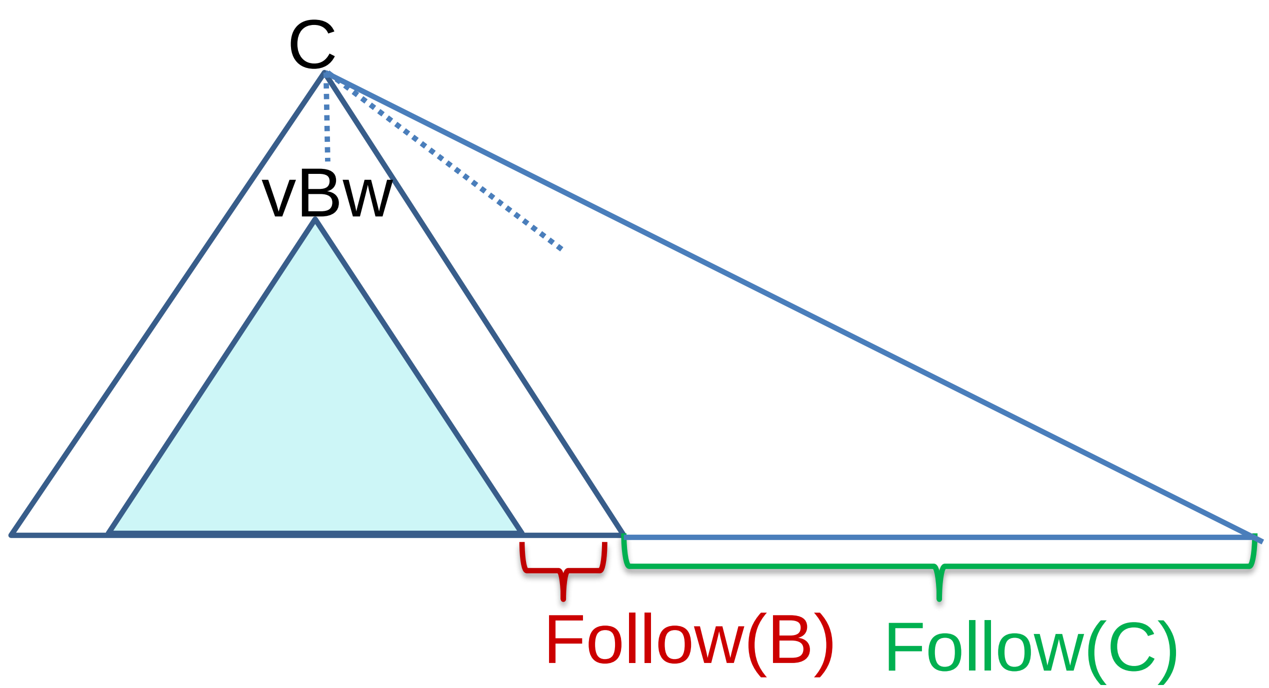
Number	Digits	Digit	Sign
+	0	0	+
−	1	1	−
0	2	2	**ε**
1	3	3	
2	4	4	
3	5	5	
4	6	6	
5	7	7	
6	8	8	
7	9	9	
8			
9			

Updated FIRST Set Computation

- If $A \rightarrow A_1 A_2 \ldots A_n$ and A_1 cannot produce ε, FIRST(A) contains FIRST(A_1)
- If $A \rightarrow A_1 A_2 \ldots A_n$ and A_1 can produce ε, FIRST(A) contains both FIRST(A_1) and FIRST(A_2)
- If $A \rightarrow A_1 A_2 \ldots A_n$ and *all* A_i can produce ε, FIRST(A) contains all FIRST(A_i) and ε.

FOLLOW Sets

- Can be computed iteratively:
  - Initially, set FOLLOW(B) = FIRST(*w*) - {ε} for all rules C → *v*B*w*. *// we don't consider ε in follow sets*
  - FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(C) if *w* can derive ε.



Example1

First sets

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

E	E'	T	T'	F
id	ε	id	ε	id
(	+	(	*	(

For all rule $C \rightarrow vBw$

FOLLOW(B) = FIRST(w) - {ε}

FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(C)
if w can derive ε.

Example1

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

First sets

E	E'	T	T'	F
id	ε	id	ε	id
(	+	(	*	(

Follow sets

E	E'	T	T'	F
$	Follow(E)	+	Follow(T)	*
)		Follow(E)		Follow(T)
Follow(E')		Follow(T')		

Start symbol E automatically contains $ in its follow set.

Consider non-terminals on the Right-hand Side of the production

Example1

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

First sets

E	E’	T	T’	F
id	ε	id	ε	id
(	+	(	*	(

Follow sets

E	E’	T	T’	F
$	Follow(E)	+	Follow(T)	*
)		Follow(E)		Follow(T)
~~Follow(E’)~~		~~Follow(T’)~~		

We need to break two infinite cycles.

Example1

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

First sets

E	E'	T	T'	F
id	ε	id	ε	id
(	+	(	*	(

Follow sets

E	E'	T	T'	F
$	$	+	+	*
)	)	Follow(E)	Follow(E)	Follow(T)

Example1

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id \mid (E)$

First sets

E	E’	T	T’	F
id	ε	id	ε	id
(	+	(	*	(

Follow sets

E	E’	T	T’	F
$	$	+	+	*
)	)	$	$	+
		)	)	$
				)

Example2

statement → if **expr** then **statement** else **statement**

| if **expr** then **statement**

| id = **expr**

| id (**expr**)

expr → id | **expr** + id

First sets

statement	**expr**
if	id
id	

Follow sets

statement	**expr**

Example2

statement → if **expr** then **statement** else **statement**
| if **expr** then **statement**
| id = **expr**
| id (**expr**)
expr → id | **expr** + id

First sets

statement	**expr**
if	id
id	

Follow sets

statement	**expr**
$	
else	
Follow(statement)	

Example2

statement → if **expr** then **statement** else **statement**

| if **expr** then **statement**

| id = **expr**

| id (**expr**)

expr → id | **expr** + id

First sets

statement	**expr**
if	id
id	

Follow sets

statement	**expr**
$	
else	
~~Follow(statement)~~	

Example2

statement → if **expr** then **statement** else **statement**

| if **expr** then **statement**

| id = **expr**

| id (**expr**)

expr → id | **expr** + id

First sets

statement	**expr**
if	id
id	

Follow sets

statement	**expr**
$	then
else	Follow(statement)
	)
	+

Example2

statement → if **expr** then **statement** else **statement**

| if **expr** then **statement**

| id = **expr**

| id (**expr**)

expr → id | **expr** + id

First sets

statement	expr
if	id
id	

Follow sets

statement	expr
$	then
else	$
	else
	)
	+

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement** **else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**
$	Follow(statement)	Follow(if-stmt)	)
First(else-part)			
Follow(if-stmt)			
Follow(else-part)			

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**
$	Follow(statement)	Follow(if-stmt)	)
else			
Follow(if-stmt)			
Follow(else-part)			

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**
$	Follow(statement)	Follow(if-stmt)	)
else			
Follow(if-stmt)			
Follow(else-part)			

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**
$	Follow(statement)	Follow(if-stmt)	)
else			
~~Follow(if-stmt)~~			
Follow(else-part)			

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**
$	Follow(statement)	Follow(if-stmt)	)
else			
~~Follow(if-stmt)~~			
~~Follow(else-part)~~			

Example3

statement → **if-stmt** | other

if-stmt → if (**exp**) **statement else-part**

else-part → else **statement** | ε

exp → 0 | 1

First sets

statement	if-stmt	else-part	exp
if	if	else	0
other		ε	1

Follow sets

statement	**if-stmt**	**else-part**	**exp**
$	$	$	)
else	else	else	

Next Time

- A More Elaborate Example

- Bottom-Up Parsing
  - Shift/reduce parsing
  - LR(0)
  - LR(1)