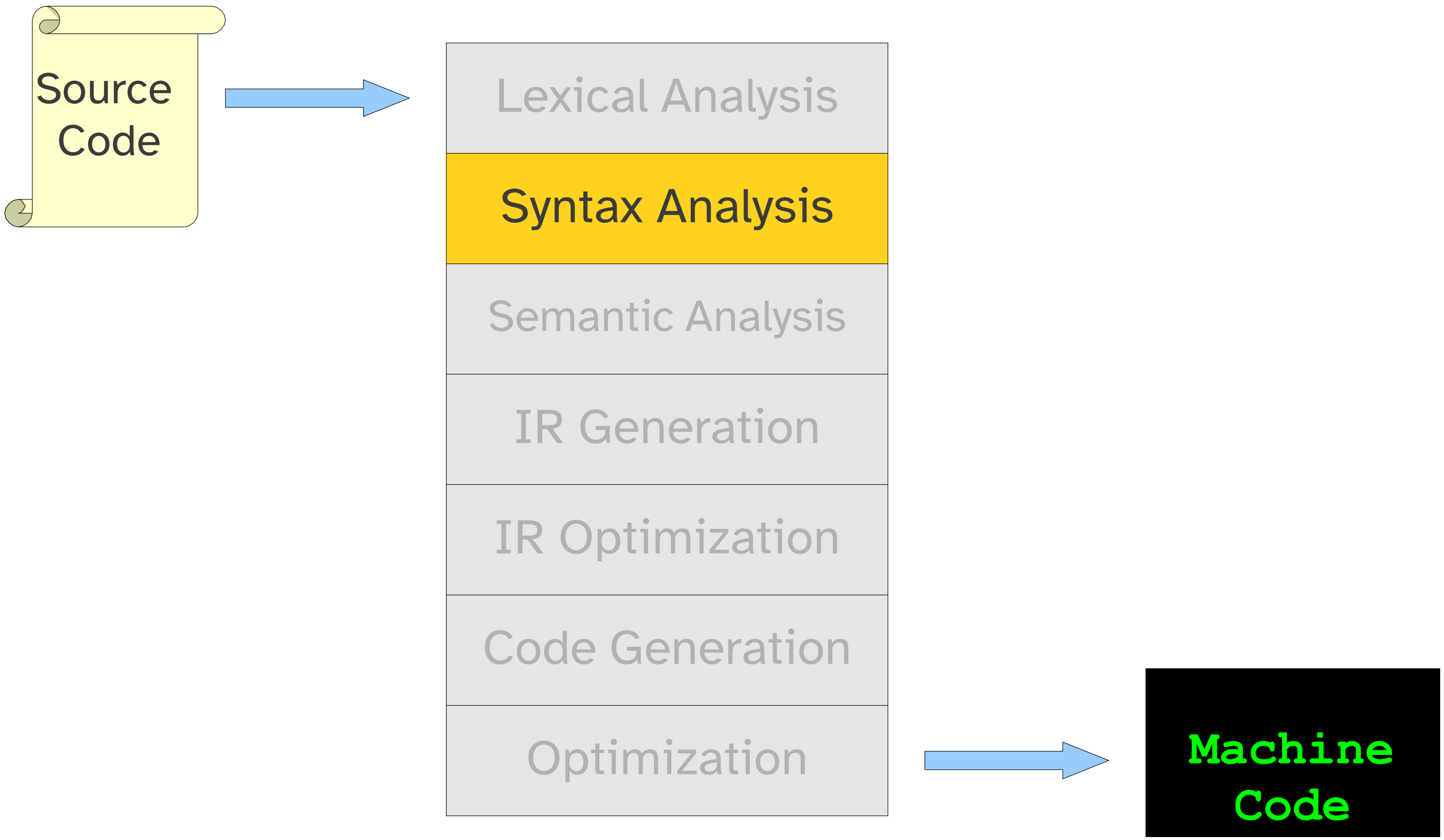
Top-Down Parsing

Where We Are


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

Review from Last Time

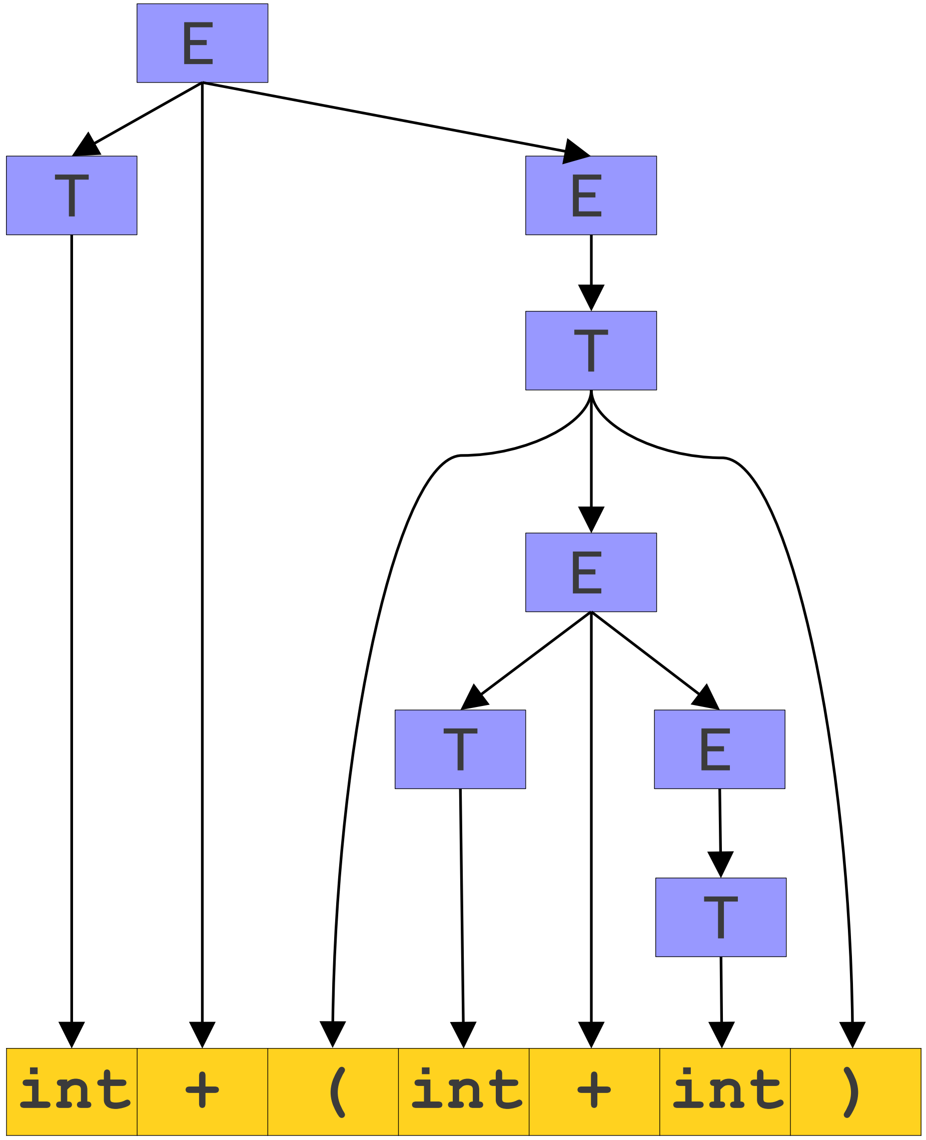
- Goal of syntax analysis: recover the intended structure of the program.
- First step: Use a **context-free grammar** to describe the language.
- Given a sequence of tokens, look for a **parse tree**. that generates those tokens.
- Recovering this syntax tree is called parsing

Different Types of Parsing

- **Top-Down Parsing**
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.
- **Bottom-Up Parsing**
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

Top-Down Parsing

E → T
E → T + E
T → int
T → (E)

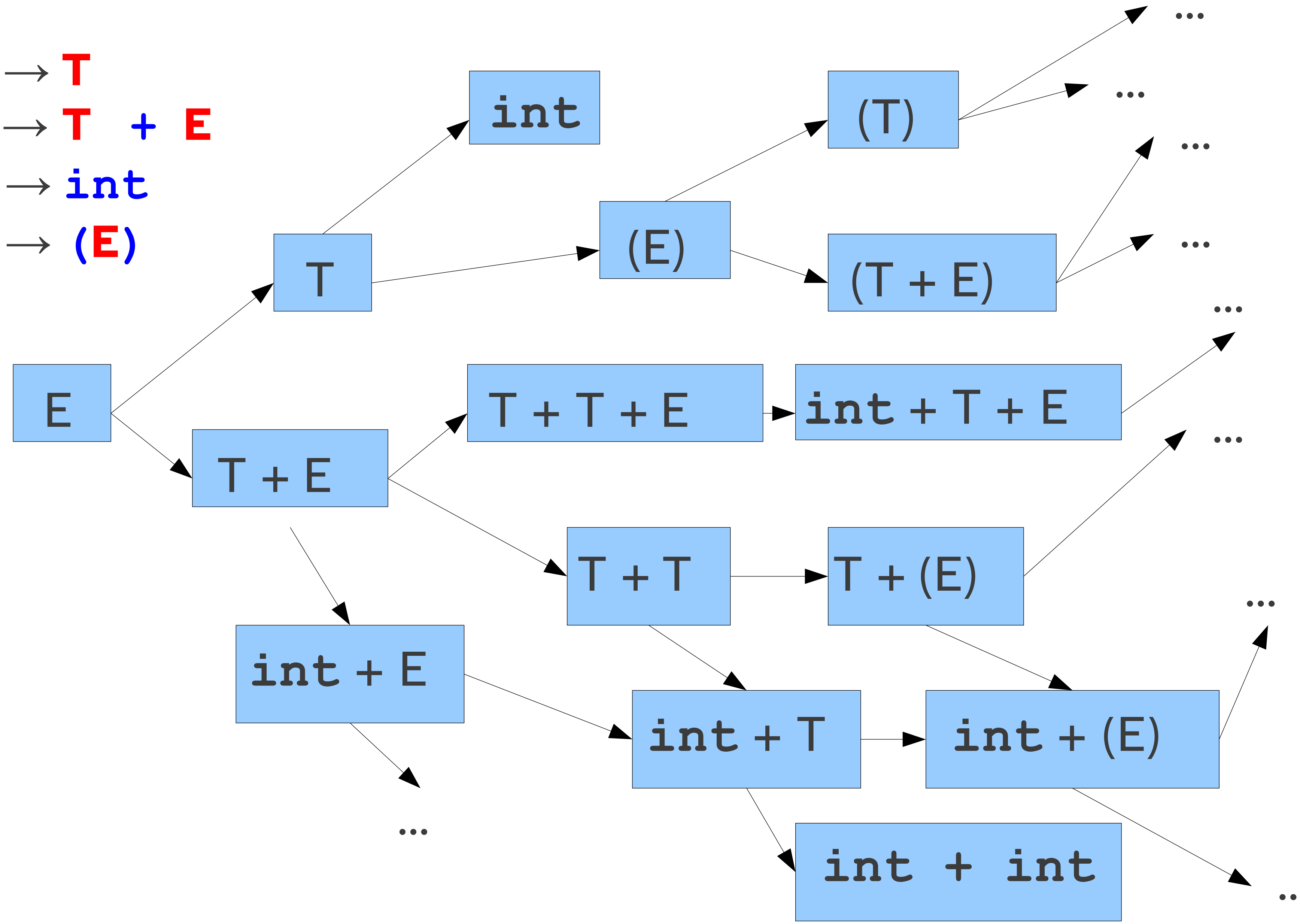


Parsing as a Search

- An idea: **treat parsing as a graph search**
- Each node is a **sentential form**, i.e., a string of terminals and nonterminals derivable from the start symbol.
- There is an edge from node $\boldsymbol{\alpha}$ to node $\boldsymbol{\beta}$ iff $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta}$.

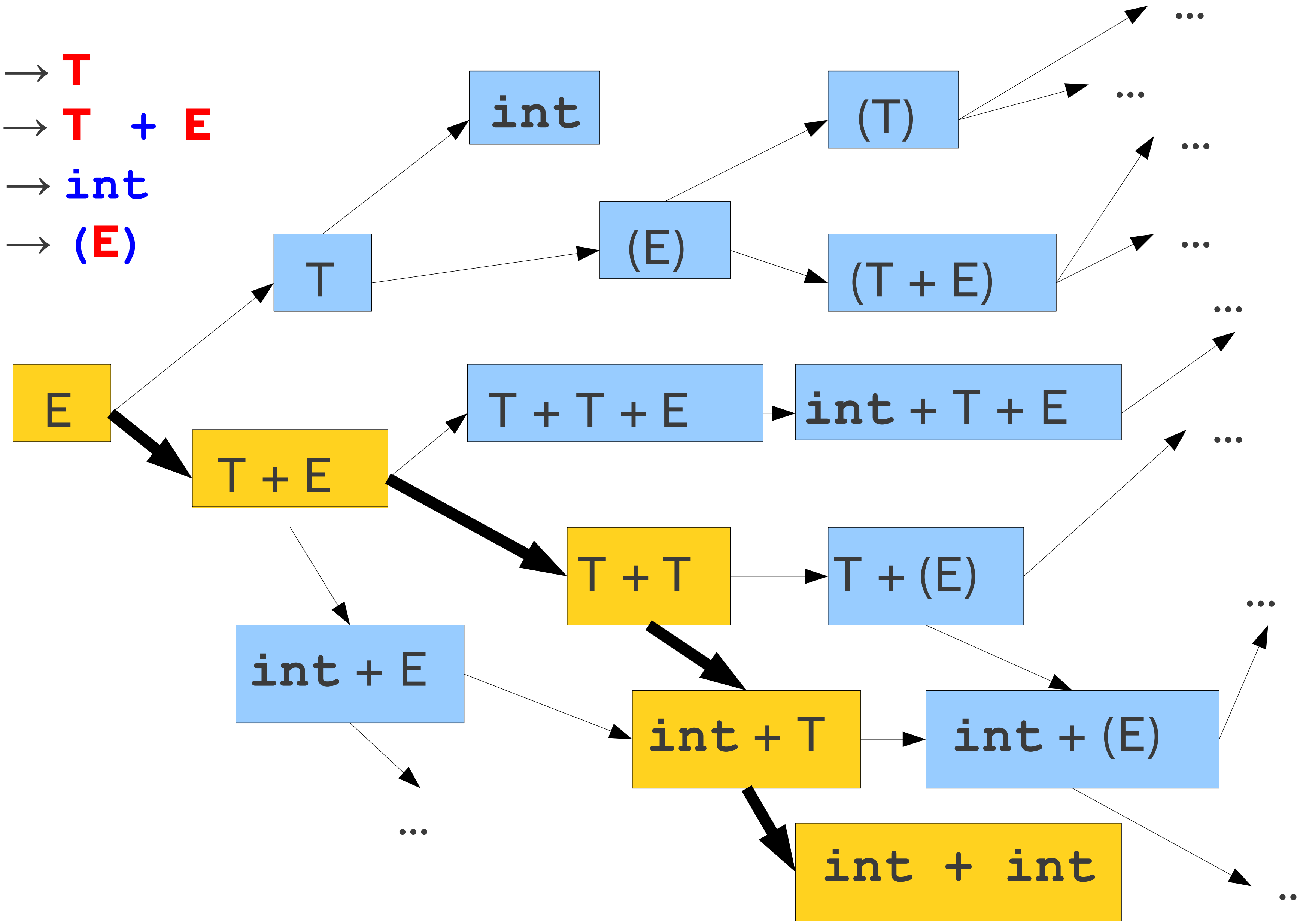
Parsing as a Search

E → T
E → T + E
T → int
T → (E)



Parsing as a Search

E → T
E → T + E
T → int
T → (E)



Our First Top-Down Algorithm

- **Breadth-First Search**
- 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.

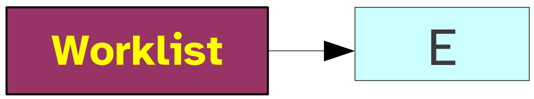
BFS is Slow

- 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.

Leftmost Derivations

- Recall: A **leftmost derivation** is one where we always expand the leftmost symbol first.
- Updated algorithm:
- Do a breadth-first search, **only considering leftmost derivations**.
  - Dramatically drops branching factor.
  - Increases likelihood that we get a prefix of nonterminals.
- Prune sentential forms that can't possibly match.
  - Avoids wasted effort.

Leftmost BFS

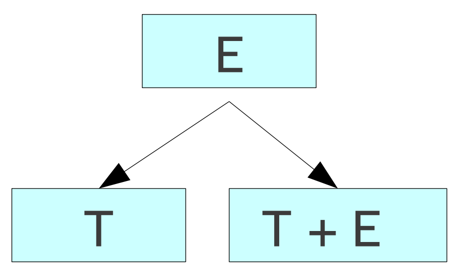


$\mathbf{E} \longrightarrow \mathbf{T}$
$\mathbf{E} \longrightarrow \mathbf{T} + \mathbf{E}$
$\mathbf{T} \longrightarrow \texttt{int}$
$\mathbf{T} \longrightarrow (\mathbf{E})$

`int + int`

Leftmost BFS

Worklist



$E \rightarrow T$

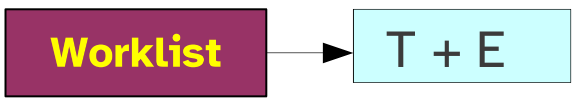
$E \rightarrow T + E$

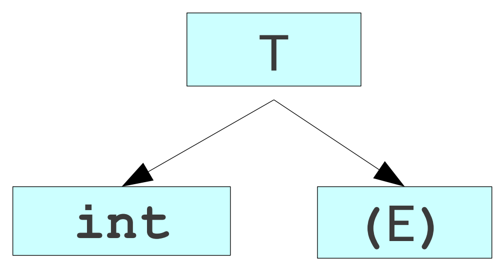
$T \rightarrow$ int

$T \rightarrow$ (E)

int + int

Leftmost BFS





E → **T**

E → **T + E**

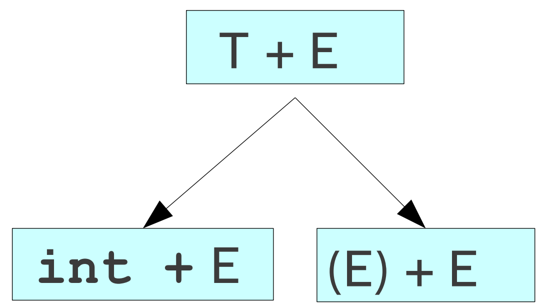
T → `int`

T → `(E)`

`int + int`

Leftmost BFS

Worklist



$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Leftmost BFS Has Problems

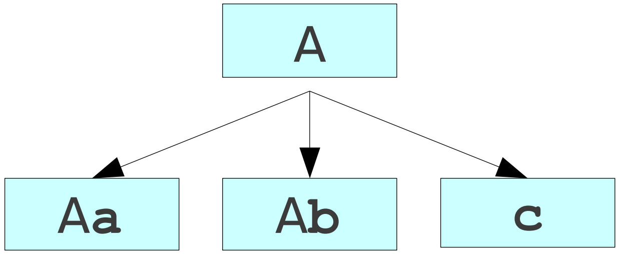
Worklist

$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaaa

Leftmost BFS Has Problems

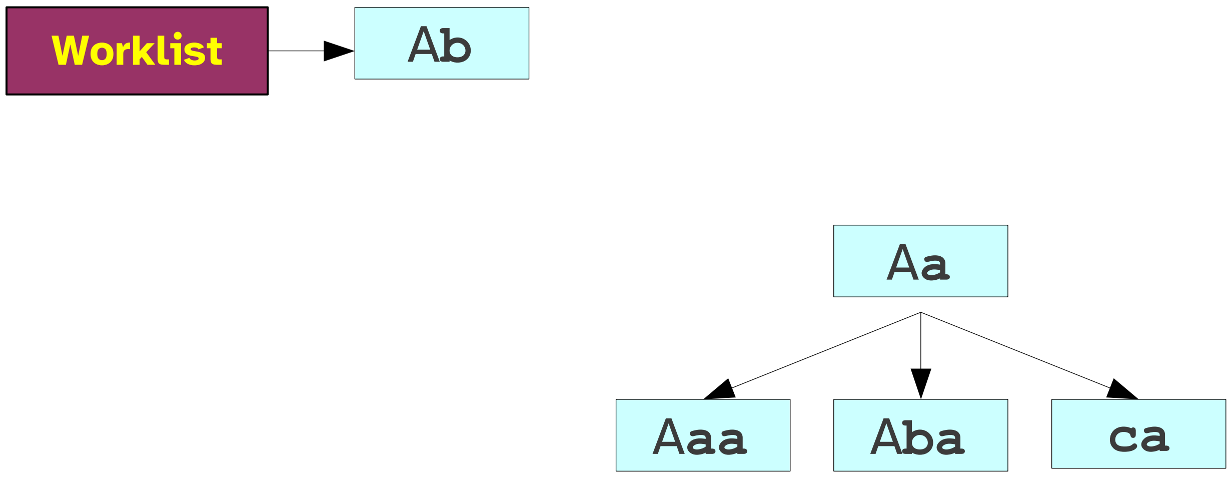
Worklist



$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaa

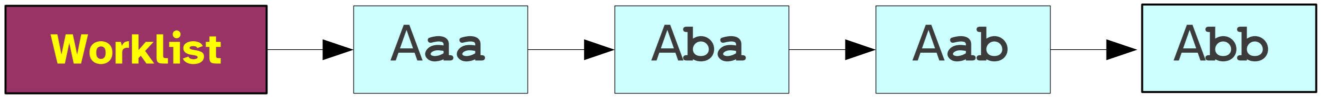
Leftmost BFS Has Problems



$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaa

Leftmost BFS Has Problems



$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaa

Leftmost DFS

- Idea: Use **depth-first** search.
- Advantages:
  - Lower memory usage: Only considers one branch at a time.
  - High performance: On many grammars, runs very quickly.
  - Easy to implement: Can be written as a set of mutually recursive functions.

Leftmost DFS

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow$ `int`
$T \rightarrow$ `(E)`

E
T + E
int + E
int + T
int + int

`int + int`

Problems with Leftmost DFS

A → **Aa** | **c**

A
Aa
Aaa
Aaaa
Aaaaa

c

Summary of Leftmost BFS/DFS

Leftmost BFS

- Works on all grammars.
- Worst-case runtime is exponential.
- Worst-case memory usage is exponential.
- Rarely used in practice.

Leftmost DFS

- Works on grammars <u>without</u> left recursion.
- Worst-case runtime is exponential.
- Worst-case memory usage is linear.
- Often used in a limited form as **recursive descent**.

Recursive Descent Parsing

- Define a function for every nonterminal
- Every function works as follows
  - Find applicable production rule
  - Terminal function checks match with next input token (if no match reports error)
  - Nonterminal function calls (recursively) other functions
- If there are several applicable productions for a nonterminal, use lookahead

Matching Tokens

```
match(token t) {
  if (current == t)
    current = next_token()
  else
    error
}
```

- Variable **`current`** holds the current input token

Functions for nonterminals

E → LIT | (E OP E) |
not E

LIT → true | false
OP → and | or | xor

```
E() {
  if (current ∈ {TRUE, FALSE}) // E → LIT
    LIT();
  else if (current == LPAREN)  // E → ( E OP E )
   match(LPAREN); E(); OP(); E(); match(RPAREN);
  else if (current == NOT)    // E → not E
   match(NOT); E();
  else
   error;
}
```

```
LIT() {
  if (current == TRUE) match(TRUE);
  else if (current == FALSE) match(FALSE);
  else error;
}
```

Predictive Parsing

Predictive Parsing

- The leftmost DFS/BFS algorithms are **backtracking** algorithms.
  - Guess which production to use, then back up if it doesn't work.
  - Try to match a prefix by sheer dumb luck.
- There is another class of parsing algorithms called **predictive** algorithms.
  - Based on remaining input, predict (*without backtracking*) which production to use.

Exploiting Lookahead

- Given just the start symbol, how do you know which productions to use to get to the input program?
- Idea: Use **lookahead tokens**.
- When trying to decide which production to use, look at some number of tokens of the input to help make the decision.

Predictive Parsing

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

int	+	(	int	+	int	)

Predictive Parsing

E → T
E → T + E
T → int
T → (E)

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)
int + (int + int)

int	+	(	int	+	int	)

A Simple Predictive Parser: LL(1)

- Top-down, predictive parsing:
  - **L**: Left-to-right scan of the tokens
  - **L**: Leftmost derivation.
  - **(1)**: One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
  - When expanding a nonterminal, we predict the production to use by looking at the next token of the input. **The decision is forced.**

LL(1) Parse Tables

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

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

LL(1) Parsing

`(int + (int * int))`

(1) $E \longrightarrow$ `int`
(2) $E \longrightarrow$ `(`E Op E`)`
(3) $Op \longrightarrow$ `+`
(4) $Op \longrightarrow$ `*`

Example with tables format

LL(1) Error Detection

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

```
int + int$
```

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

E$	int + int$
int $	int + int$
$	+ int$

???

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

`(int (int))$`

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

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

???

	int	(	)	+	*
E	1	2			
Op				3	4

The LL(1) Algorithm

- Suppose a grammar has start symbol **S** and LL(1) parsing table T. We want to parse string ω
- Initialize a stack containing **S**`$`.
- Repeat until the stack is empty:
  - Let the next character of ω be `t`.
  - If the top of the stack is a terminal `r`:
    - If `r` and `t` don't match, report an error.
    - Otherwise consume the character `t` and pop `r` from the stack.
  - Otherwise, the top of the stack is a nonterminal **A**:
    - If T[**A**, `t`] is undefined, report an error.
    - Replace the top of the stack with T[**A**, `t`].

Can we find an algorithm for constructing LL(1) parse tables?

Filling in Table Entries

- Intuition: The next character should uniquely identify a production, so we should pick a production that ultimately starts with that character.
- T[**A**, `t`] should be a production **A** $\rightarrow \omega$ iff ω derives something starting with `t`.

In what follows, assume that our grammar does not contain any ε-productions.

We'll relax this restriction later.

FIRST Sets

- We want to tell if a particular nonterminal **A** derives a string starting with a particular nonterminal **`t`**.
- We can formalize this with **FIRST sets**.
  - FIRST(**A**) = { **`t`** | **A** $\Rightarrow^*$ **`t`**ω for some ω }
  - Intuitively, FIRST(**A**) is the set of terminals that can be at the start of a string produced by **A**.
- If we can compute FIRST sets for all nonterminals in a grammar, we can efficiently construct the LL(1) parsing table.

Computing FIRST Sets

- Initially, for all nonterminals **A**, set

  FIRST(**A**) = { **t** | **A** → **t**ω for some ω }

- Then, repeat the following until no changes occur:
  - For each nonterminal **A**, for each production **A** → **B**ω, set

    FIRST(**A**) = FIRST(**A**) ∪ FIRST(**B**)

- This is known a **fixed-point iteration** or a **transitive closure algorithm**.

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`	`zero?`	`id`
`while`	`not`	`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`
| `constan`
`t`

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

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`	

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`		

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												

ε-Free LL(1) Parse Tables

- The following algorithm constructs an LL(1) parse table for a grammar with no ε-productions.
- Compute the FIRST sets for all nonterminals in the grammar.
  - For each production $\mathbf{A} \to \mathtt{t}\omega$, set $T[\mathbf{A}, \mathtt{t}] = \mathtt{t}\omega$.
  - For each production $\mathbf{A} \to \mathbf{B}\omega$, set $T[\mathbf{A}, \mathtt{t}] = \mathbf{B}\omega$ for each $\mathtt{t} \in \text{FIRST}(\mathbf{B})$.

Exercise

- Write a CFG for integers without ε-productions

- FIRST sets?

FIRST Sets with ε

Num	→	Sign Digits
Sign	→	+ | − | ε
Digits	→	Digit More
More	→	Digits | ε
Digit	→	0 | 1 | 2 | … | 9

Num	Sign	Digit	Digits	More

FIRST Sets with ε

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

Num	Sign	Digit	Digits	More
+ −	+ −	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9

FIRST Sets with ε

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

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

FIRST Sets with ε

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

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

FIRST and ε

- When computing FIRST sets in a grammar with ε-productions, we often have to “look through” nonterminals.
- Rationale: Might have a derivation like this:

$$\mathbf{A} \Rightarrow \mathbf{B}\mathtt{t} \Rightarrow \mathtt{t}$$

So $\mathtt{t} \in$ FIRST($\mathbf{A}$).

FIRST Computation with ε

- Initially, for all nonterminals **A**, set

  $$\text{FIRST}(\mathbf{A}) = \{\ \mathtt{t} \mid \mathbf{A} \to \mathtt{t}\omega \text{ for some } \omega\ \}$$

- For all nonterminals **A** where $\mathbf{A} \to \varepsilon$ is a production, add ε to FIRST(**A**).
- Repeat the following until no changes occur:
  - For each production $\mathbf{A} \to \alpha$, where α is a string of nonterminals whose FIRST sets all contain ε, set $\text{FIRST}(\mathbf{A}) = \text{FIRST}(\mathbf{A}) \cup \{\ \varepsilon\ \}$.
  - For each production $\mathbf{A} \to \alpha\mathtt{t}\omega$, where α is a string of nonterminals whose FIRST sets all contain ε, set $\text{FIRST}(\mathbf{A}) = \text{FIRST}(\mathbf{A}) \cup \{\ \mathtt{t}\ \}$
  - For each production $\mathbf{A} \to \alpha\mathbf{B}\omega$, where α is string of nonterminals whose FIRST sets all contain ε, set $\text{FIRST}(\mathbf{A}) = \text{FIRST}(\mathbf{A}) \cup (\text{FIRST}(\mathbf{B}) - \{\ \varepsilon\ \})$.

A Notational Diversion

- Once we have computed the correct FIRST sets for each nonterminal, we can generalize our definition of FIRST sets to strings.
- Define FIRST*(ω) as follows:
  - FIRST*(ε) = { ε }
  - FIRST*($\mathbf{t}\omega$) = { $\mathbf{t}$ }
  - If $\varepsilon \notin$ FIRST($\mathbf{A}$):
    - FIRST*($\mathbf{A}\omega$) = FIRST($\mathbf{A}$)
  - If $\varepsilon \in$ FIRST($\mathbf{A}$):
    - FIRST*($\mathbf{A}\omega$) = (FIRST($\mathbf{A}$) - { ε }) ∪ FIRST*(ω)

FIRST Computation with ε

- Initially, for all nonterminals **A**, set

  $$\text{FIRST}(\mathbf{A}) = \{\ \mathtt{t} \mid \mathbf{A} \to \mathtt{t}\omega \text{ for some } \omega\ \}$$

- For all nonterminals **A** where $\mathbf{A} \to \varepsilon$ is a production, add ε to FIRST(**A**).
- Repeat the following until no changes occur:
  - For each production $\mathbf{A} \to \alpha$, set

    $$\text{FIRST}(\mathbf{A}) = \text{FIRST}(\mathbf{A}) \cup \text{FIRST}^*(\alpha)$$

LL(1) Tables with ε

Num	→	**Sign Digits**
Sign	→	**+** | **–** | **ε**
Digits	→	**Digit More**
More	→	**Digits** | **ε**
Digit	→	**0** | **1** | **2** | ... | **9**

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

	+	–	#
Num			
Sign			
Digits			
More			
Digit			

LL(1) Tables with ε

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

Num $	- 5 $
Sign Digits $	- 5 $
- Digits $	- 5 $
Digits $	5 $
Digit More $	5 $
5 More $	5 $
More $	$

	+	−	#
Num			
Sign			
Digits			
More			
Digit			

?????

ε is Complicated

- When constructing LL(1) tables with ε-productions, we need to have an extra column for **$**.

	+	**-**	**#**	**$**
Num				
Sign				
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **–** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | **2** | ... | **9**

Num $	5 $
Sign Digits $	5 $

?????

	+	–	#	$
Num				
Sign				
Digits				
More				
Digit				

FOLLOW Sets

- With ε-productions in the grammar, we may have to "look past" the current nonterminal to what can come after it.
- The **FOLLOW set** represents the set of terminals that might come after a given **nonterminal**. Formally:
  - FOLLOW(**A**) = { $\mathtt{t}$ | **S** $\Rightarrow^* \alpha \mathbf{A} \mathtt{t} \omega$ for some α, ω } where **S** is the start symbol of the grammar.
- Informally, every terminal that can ever come after **A** in a derivation.

Computation of FOLLOW Sets

- Initially, for each nonterminal **A**, set FOLLOW(**A**) = { **t** | **B** $\rightarrow \alpha$**At**ω is a production }
- Add **$** to FOLLOW(**S**), where **S** is the start symbol.
- Repeat the following until no changes occur:
  - If **B** $\rightarrow \alpha$**A**ω is a production, set
    FOLLOW(**A**) = FOLLOW(**A**) ∪ FIRST*(ω) - { **ε** }.
  - If **B** $\rightarrow \alpha$**A**ω is a production and **ε** ∈ FIRST*(ω), set
    FOLLOW(**A**) = FOLLOW(**A**) ∪ FOLLOW(**B**).

The Final LL(1) Table Algorithm

- Compute FIRST(**A**) and FOLLOW(**A**) for all nonterminals **A**.
- For each rule **A** $\rightarrow \omega$, for each terminal **t** $\in$ FIRST*(ω), set T[**A**, **t**] = ω.
    - Note that **ε** is not a terminal.
- For each rule **A** $\rightarrow \omega$, if **ε** $\in$ FIRST*(ω), set T[**A**, **t**] = ω for each **t** $\in$ FOLLOW(**A**).

Final LL(1) Table

Num → **Sign Digits**
Sign → + | – | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | ... | 9

FIRST				
Num	Sign	Digit	Digits	More
+ – #	+ – ε	#	#	# ε

FOLLOW				
Num	Sign	Digit	Digits	More
$	#	# $	$	$

	+	–	#	$
Num	Sign Digits	Sign Digits	Sign Digits	
Sign	+	-	ε	
Digits			Digits More	
More			Digits	ε
Digit			#	

Exercise

Num	→	**Digits Frac**	
Digits	→	**Digit More**	
Frac	→	. **Digits**	| ε
More	→	**Digits**	| ε
Digit	→	0 | 1 | 2 | … | 9	

FIRST				
Num	Digits	Frac	More	Digit

FOLLOW				
Num	Digits	Frac	More	Digit

	.	#	$
Num			
Digits			
Frac			
More			
Digit			

The Limits of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:
  - $A \rightarrow Ab \mid c$
  - FIRST(A) = {c}
- However, we cannot build a valid LL(1) parse table.
- **Why?**

Eliminating Left Recursion

- In general, left recursion can be converted into **right recursion** by a mechanical transformation.
- Consider the grammar

$$\mathbf{A} \rightarrow \mathbf{A}\omega \mid \alpha$$

- This will produce α followed by some number of ω's.
- Can rewrite the grammar as

$$\mathbf{A} \rightarrow \alpha\mathbf{B}$$

$$\mathbf{B} \rightarrow \varepsilon \mid \omega\mathbf{B}$$

Algorithm to Remove Left Recursion

- Arrange **n** nonterminals in some order
- **for** i: 1 .. n {

  for j: 1 .. i-1 {

  ‘inline’ all $A_j \rightarrow \boldsymbol{\delta}$ into $A_i \rightarrow A_j\boldsymbol{\gamma}$

  i.e., will have by $A_i \rightarrow \boldsymbol{\delta\gamma}$ productions

  }

  eliminate immediate left recursion among A_i productions

  }
- Example:

  $S \rightarrow A\ a \mid b$

  $A \rightarrow A\ c \mid S\ d \mid \varepsilon$

Note: this CFG is not LL(1). Removing left recursion doesn’t make it LL(1)

Another Non-LL(1) Grammar

- Consider the following grammar:

  E → **T**

  E → **T** **+** **E**

  T → **`int`**

  T → **`(`E`)`**

- FIRST(**E**) = { **`int`**, **`(`** }
- FIRST(**T**) = { **`int`**, **`(`** }
- Why is this grammar not LL(1)?

Left-Factoring

- A grammar of the form

$$\mathbf{A} \rightarrow \alpha\, \boldsymbol{\beta} \mid \alpha\, \boldsymbol{\gamma} \mid \omega$$

Can be transformed as

$$\mathbf{A} \rightarrow \alpha\, \mathbf{B} \mid \omega$$

$$\mathbf{B} \rightarrow \boldsymbol{\beta} \mid \boldsymbol{\gamma}$$

Left-Factoring

$$\mathtt{E} \longrightarrow \mathtt{T}$$
$$\mathtt{E} \longrightarrow \mathtt{T} + \mathtt{E}$$
$$\mathtt{T} \longrightarrow \mathtt{int}$$
$$\mathtt{T} \longrightarrow (\mathtt{E})$$

A Formal Characterization of LL(1)

- A grammar G is LL(1) iff for any productions $\mathbf{A} \rightarrow \omega_1$ and $\mathbf{A} \rightarrow \omega_2$, the sets
- FIRST(ω_1) and FIRST(ω_2) are disjoint

  and

- if $\varepsilon \in$ FIRST(ω_1): FIRST(ω_2) and FOLLOW($\mathbf{A}$) are disjoint.
- Likewise, if $\varepsilon \in$ FIRST(ω_2)

This condition is equivalent to saying that there are no conflicts in the table.

Exercise: Are they LL(1)? If not, fix

- term = id | indexed_elem .
  indexed_elem = id “[“ expr “]” .
- $S \rightarrow A\ a\ b$
  $A \rightarrow a \mid \varepsilon$
- $E \rightarrow E - T \mid T$

The Strengths of LL(1)

LL(1) is Straightforward

- Can be implemented quickly with a table- driven design.
- Can be implemented by **recursive descent**:
  - Define a function for each nonterminal.
  - Have these functions call each other based on the lookahead token.

LL(1) & Ambiguity

- Ground truth: **if a grammar is LL(1), it is not ambiguous**
- Consider:

  if a grammar is not LL(1), it is ambiguous

  and

  if a grammar is ambiguous, it is not LL(1)

- Example:

  $\mathbf{S} \longrightarrow \mathbf{A}\ x\ y \mid \mathbf{B}\ x\ z$

  $\mathbf{A} \longrightarrow a$

  $\mathbf{B} \longrightarrow a$

Summary

- **Top-down parsing** tries to derive the user's program from the start symbol.
- **Leftmost BFS** is one approach to top-down parsing; it is mostly of theoretical interest.
- **Leftmost DFS** is another approach to top-down parsing that is uncommon in practice.
- **LL(1)** parsing scans from left-to-right, using one token of lookahead to find a leftmost derivation.
- **FIRST sets** contain terminals that may be the first symbol of a production.
- **FOLLOW sets** contain terminals that may follow a nonterminal in a production.
- **Left recursion** and **left factorability** cause LL(1) to fail and can be mechanically eliminated in some cases.

Recap: FIRST calculation

- FIRST($\boldsymbol{\omega}$) is the set of terminals that begin all strings given by $\boldsymbol{\omega}$, including $\boldsymbol{\varepsilon}$ iif $\boldsymbol{\omega} \Rightarrow \boldsymbol{\varepsilon}$
- $\boldsymbol{\omega}$ is either:
  - terminal a FIRST(a)={a}
  - nonterminal **X**
    - repeat
      - for each rule **X** $\rightarrow$ $\mathbf{Y_1Y_2 \ldots Y_k}$
        - Add a to FIRST(**X**) if a $\in$ FIRST($\mathbf{Y_1}$) **or** a $\in$ FIRST($\mathbf{Y_n}$) and $\mathbf{Y_1 \ldots Y_{n-1}} \Rightarrow^* \varepsilon$
        - If $\mathbf{Y_1 \ldots Y_k} \Rightarrow^* \varepsilon$ then add ε to FIRST(**X**)
    - until no changes
  - sentential form $\boldsymbol{\alpha}$
      - for each symbol $\mathbf{Y_1Y_2 \ldots Y_k}$ in $\boldsymbol{\alpha}$
        - Add a to FIRST($\boldsymbol{\alpha}$) if a $\in$ FIRST($\mathbf{Y_1}$) **or** a $\in$ FIRST($\mathbf{Y_n}$) and $\mathbf{Y_1 \ldots Y_{n-1}} \Rightarrow^* \varepsilon$
        - If $\mathbf{Y_1 \ldots Y_k} \Rightarrow^* \varepsilon$ then add ε to FIRST($\boldsymbol{\alpha}$)