

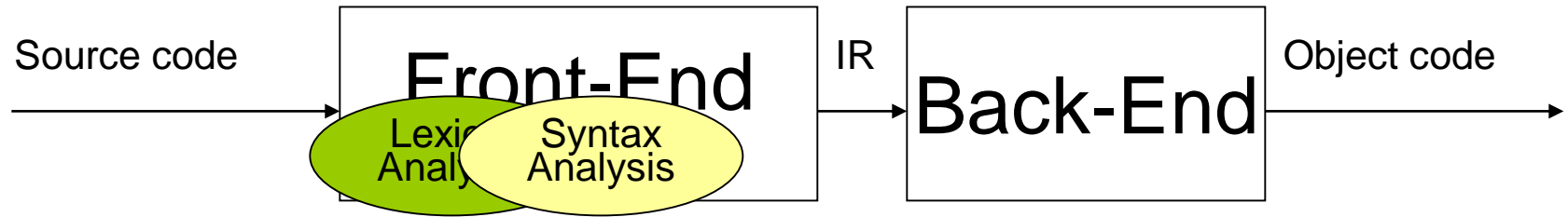
Compiler Design

Lecture 5: Syntax Analysis Top-Down Parsing

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Parsing (Syntax Analysis)



■ Syntax Analysis:

- Derivation and parse trees
- Top-down parsing
- Bottom-up parsing

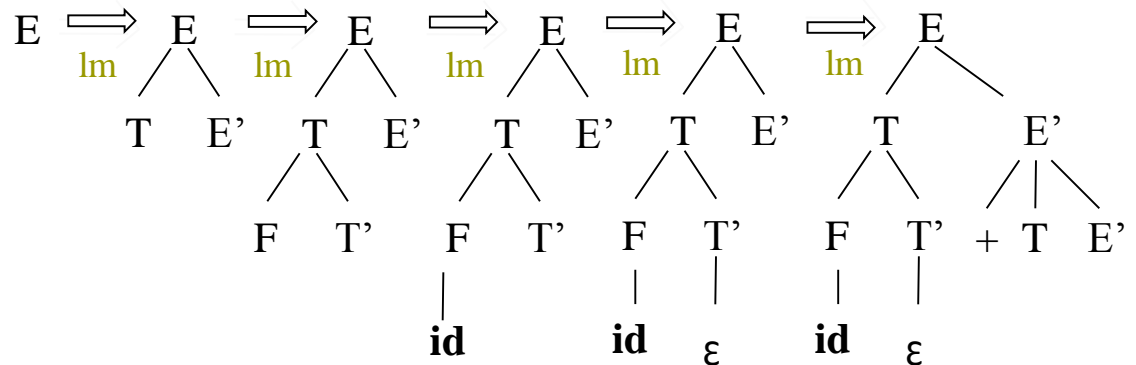
Outline

- **Introduction**
- Parsing as a Search
- Predictive Parsing

Introduction

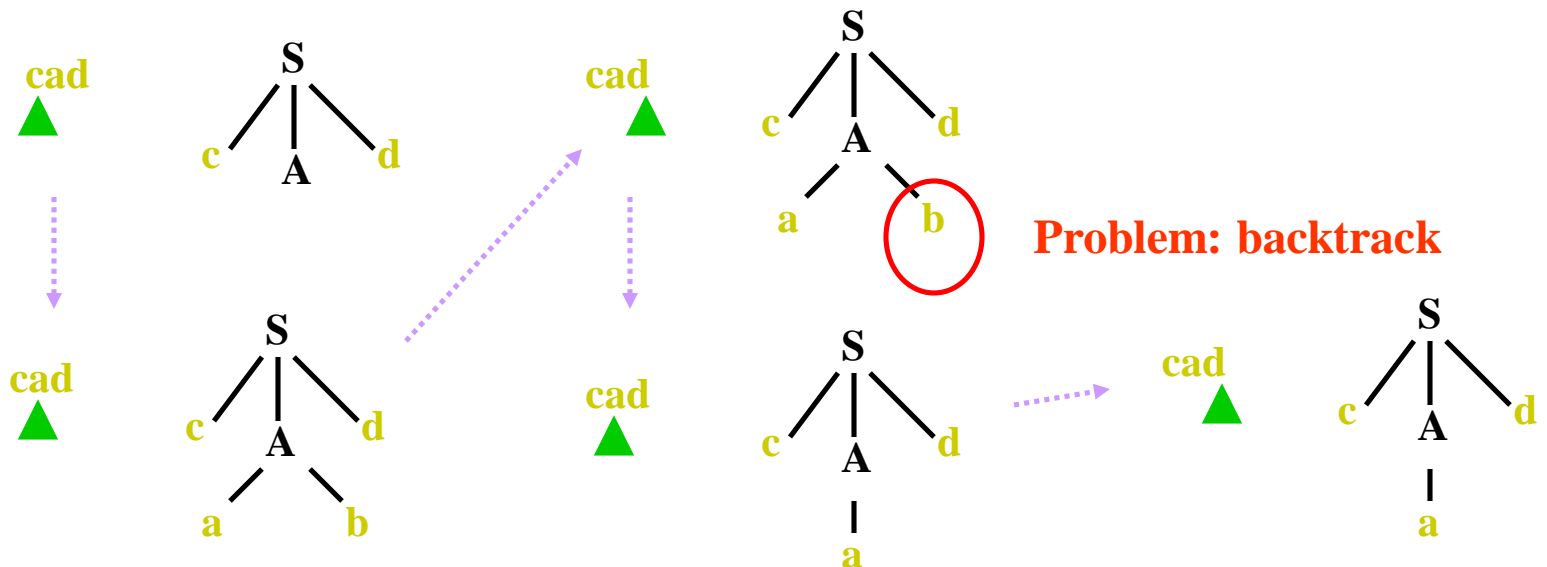
- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: id+id*id

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$



Top-Down Parsing

- Choose production rule based on input symbol
- May require backtracking to correct a wrong choice.
- Example:
$$\left. \begin{array}{l} S \rightarrow c A d \\ A \rightarrow ab \mid a \end{array} \right\} \text{input: } cad$$



Top-Down Recursive-Descent Parsing

- Construct the root with the starting symbol of the grammar.
- Repeat until the fringe of the parse tree matches the input string:
 - Assuming a node labelled A, select a production with A on its left-hand-side and, for each symbol on its right-hand-side, construct the appropriate child.
 - When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack.
 - Find the next node to be expanded.

The key is picking the right production in the first step: that choice should be guided by the input string.

Example: Top-Down Recursive-Descent Parsing

Example:

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr + Term$
3. / $Expr - Term$
4. / $Term$
5. $Term \rightarrow Term * Factor$
6. / $Term / Factor$
7. / $Factor$
8. $Factor \rightarrow number$
9. / id

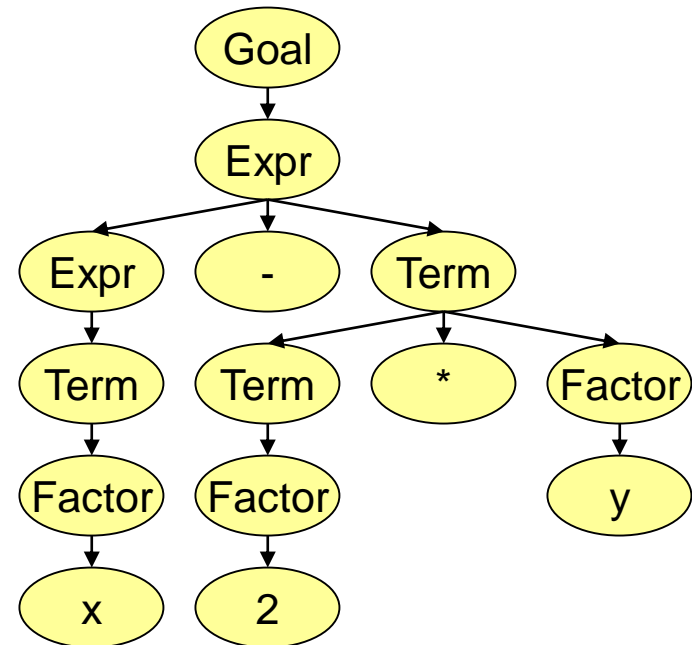
Parse $x-2*y$

Steps (one scenario from many)

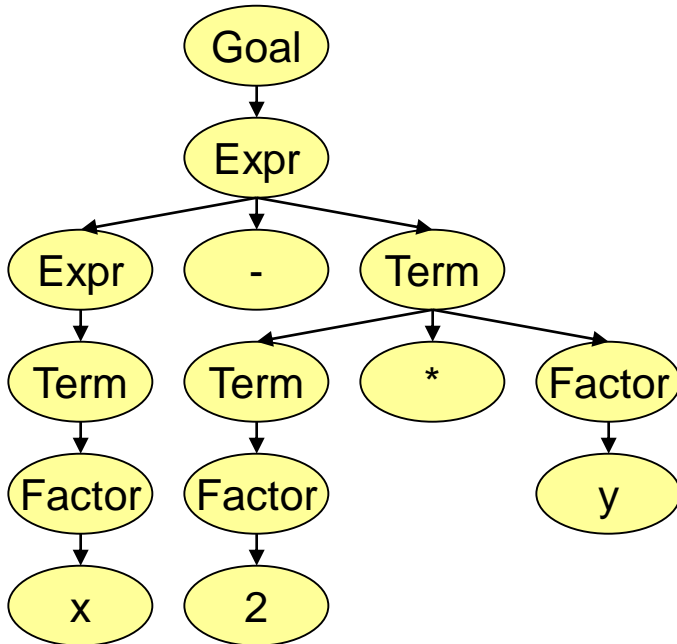
Rule	Sentential Form	Input
-	<i>Goal</i>	$x - 2*y$
1	<i>Expr</i>	$x - 2*y$
2	<i>Expr + Term</i>	$x - 2*y$
4	<i>Term + Term</i>	$x - 2*y$
7	<i>Factor + Term</i>	$x - 2*y$
9	<i>id + Term</i>	$x - 2*y$
Fail	<i>id + Term</i>	x $- 2*y$
Back	<i>Expr</i>	$x - 2*y$
3	<i>Expr - Term</i>	$x - 2*y$
4	<i>Term - Term</i>	$x - 2*y$
7	<i>Factor - Term</i>	$x - 2*y$
9	<i>id - Term</i>	$x - 2*y$
Match	<i>id - Term</i>	x - $2*y$
7	<i>id - Factor</i>	x - $2*y$
9	<i>id - num</i>	x - $2*y$
Fail	<i>id - num</i>	$x - 2$ $*y$
Back	<i>id - Term</i>	x - $2*y$
5	<i>id - Term * Factor</i>	x - $2*y$
7	<i>id - Factor * Factor</i>	x - $2*y$
8	<i>id - num * Factor</i>	x - $2*y$
match	<i>id - num * Factor</i>	$x - 2*$ y
9	<i>id - num * id</i>	$x - 2*$ y
match	<i>id - num * id</i>	$x - 2*y$

Example: Top-Down Recursive-Descent Parsing

Rule	Sentential Form	Input
-	<i>Goal</i>	x - 2*y
1	<i>Expr</i>	x - 2*y
2	<i>Expr + Term</i>	x - 2*y
4	<i>Term + Term</i>	x - 2*y
7	<i>Factor + Term</i>	x - 2*y
9	<i>id + Term</i>	x - 2*y
Fail	<i>id + Term</i>	x - 2*y
Back	<i>Expr</i>	x - 2*y
3	<i>Expr - Term</i>	x - 2*y
4	<i>Term - Term</i>	x - 2*y
7	<i>Factor - Term</i>	x - 2*y
9	<i>id - Term</i>	x - 2*y
Match	<i>id - Term</i>	x - 2*y
7	<i>id - Factor</i>	x - 2*y
9	<i>id - num</i>	x - 2*y
Fail	<i>id - num</i>	x - 2 *y
Back	<i>id - Term</i>	x - 2*y
5	<i>id - Term * Factor</i>	x - 2*y
7	<i>id - Factor * Factor</i>	x - 2*y
8	<i>id - num * Factor</i>	x - 2*y
match	<i>id - num * Factor</i>	x - 2* y
9	<i>id - num * id</i>	x - 2* y
match	<i>id - num * id</i>	x - 2*y



Example: Top-Down Recursive-Descent Parsing



Other choices for expansion are possible:

Rule	Sentential Form	Input
-	<i>Goal</i>	x - 2*y
1	<i>Expr</i>	x - 2*y
2	<i>Expr + Term</i>	x - 2*y
2	<i>Expr + Term + Term</i>	x - 2*y
2	<i>Expr + Term + Term + Term</i>	x - 2*y
2	<i>Expr + Term + Term + ... + Term</i>	x - 2*y

- Wrong choice leads to non-termination!
- This is a bad property for a parser!
- Parser must make the right choice!

Challenges in Top-Down Parsing

- How can we know which productions to apply?
- In general, we can't.
 - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
 - If we have to guess, how do we do it?

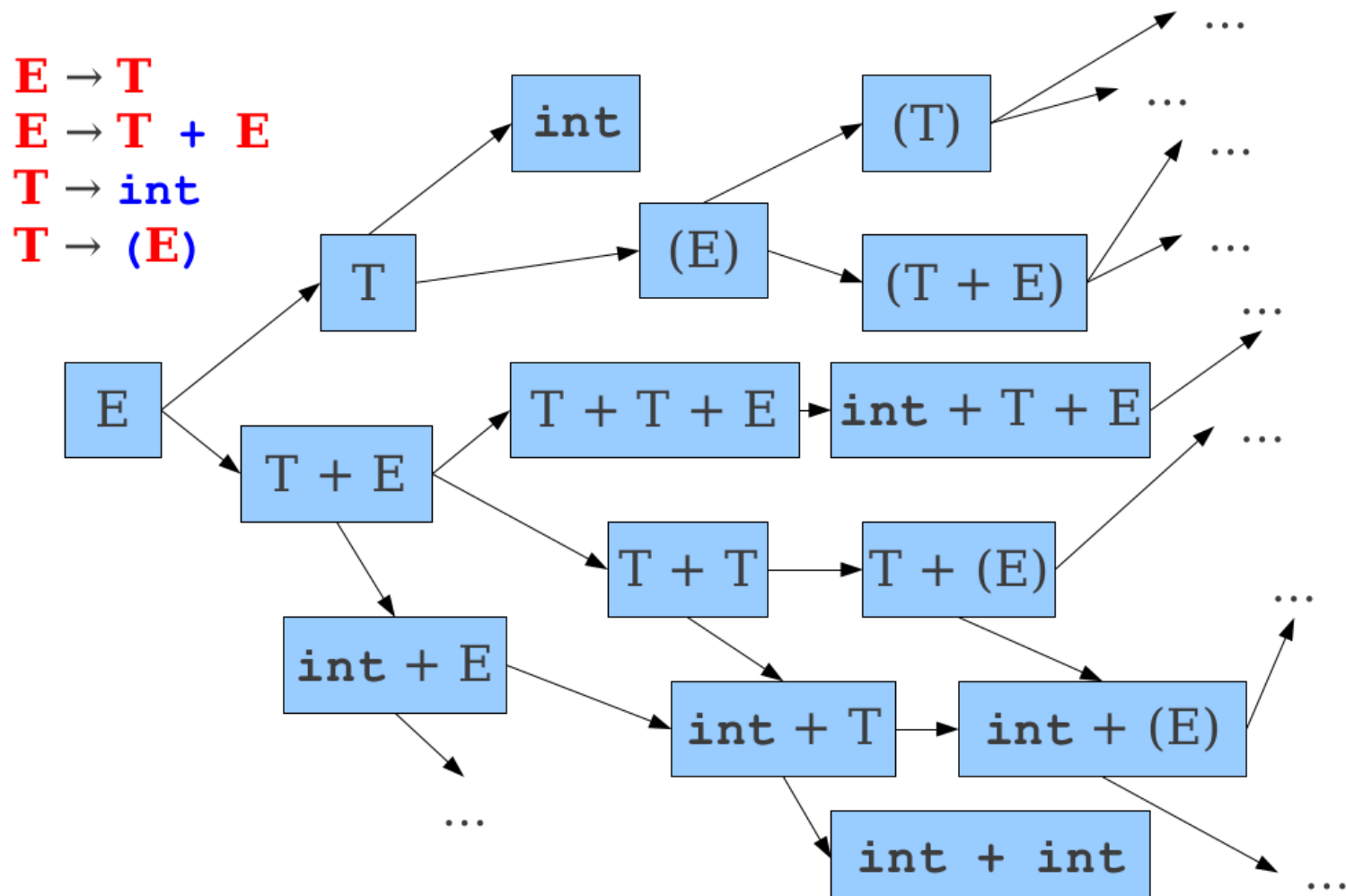
Outline

- Introduction
- **Parsing as a Search**
- Predictive Parsing

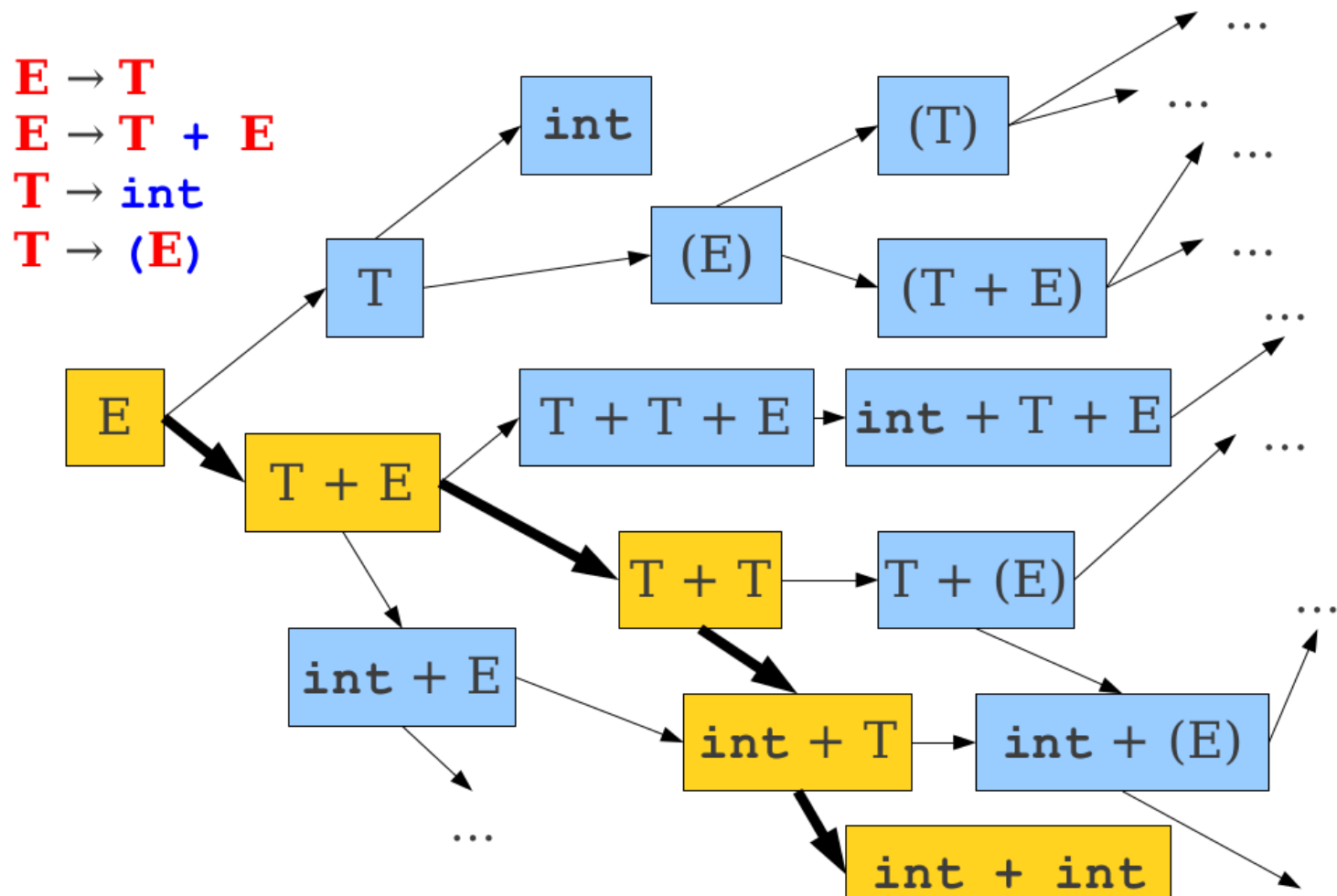
Parsing as a Search

- An idea: treat parsing as a graph search.
- Each node is a sentential form (a string of terminals and nonterminals derivable from the start symbol).
- There is an edge from node α to node β iff $\alpha \Rightarrow \beta$.

Parsing as a Search



Parsing as a Search



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.

Breadth-First Search Parsing

■ Example:



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

`int + int`

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.

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.

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

- Substantial improvement over naïve algorithm.
- Will always find a valid parse of a program if one exists.
- Can easily be modified to find if a program can't be parsed.
- But, there are still problems.

Leftmost Breadth-First Search Parsing

■ Example:



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

caaaaaaaaaa

Problems with Leftmost BFS

- Grammars like this can make parsing take exponential time.
- Also uses exponential memory.
- What if we search the graph with a different algorithm?

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

■ Example:

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

E

int + int

Problems with Leftmost DFS

■ Example:

A → **A****a** | **c**

A
Aa
Aaa
Aaaa
Aaaaa

c

Problems with Leftmost DFS

- Left Recursion
- A grammar is left-recursive if it has a left-recursive nonterminal symbol with the following derivation

$$A \Rightarrow + A\omega$$

for some string ω .

- A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
- Leftmost DFS may fail on left-recursive grammars.

Left-Recursive Grammars

- Fortunately, in many cases it is possible to eliminate left recursion.

- **Eliminating left-recursion:**

- In many cases, it is sufficient to replace

$$A \rightarrow Aa / b \quad \text{with} \quad A \rightarrow bA' \quad \text{and} \quad A' \rightarrow aA' / \varepsilon$$

- Example:

$$Sum \rightarrow Sum + number / number$$

would become:

$$Sum \rightarrow number \ Sum'$$

$$Sum' \rightarrow +number \ Sum' / \varepsilon$$

Eliminating Left Recursion

Example:

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr + Term$
3. $/ Expr - Term$
4. $/ Term$
5. $Term \rightarrow Term * Factor$
6. $/ Term / Factor$
7. $/ Factor$
8. $Factor \rightarrow number$
9. $/ id$

Eliminating Left Recursion

Example:

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr + Term$
3. $/ Expr - Term$
4. $/ Term$
5. $Term \rightarrow Term * Factor$
6. $/ Term / Factor$
7. $/ Factor$
8. $Factor \rightarrow number$
9. $/ id$

$$\begin{aligned} Expr &\rightarrow Term Expr' \\ Expr' &\rightarrow +Term Expr' \\ &\quad / -Term Expr' \\ &\quad / \varepsilon \\ Term &\rightarrow Factor Term' \\ Term' &\rightarrow *Factor Term' \\ &\quad / / Factor Term' \\ &\quad / \varepsilon \end{aligned}$$

remain unchanged:

$$\begin{aligned} Goal &\rightarrow Expr \\ Factor &\rightarrow number \\ &\quad / id \end{aligned}$$

Eliminating Left Recursion

- Problem: If left recursion is two-or-more levels deep, this is not enough

$$\left. \begin{array}{l} S \rightarrow Aa \mid b \\ A \rightarrow Ac \mid Sd \mid \epsilon \end{array} \right\} S \Rightarrow Aa \Rightarrow Sda$$

Left-Recursive Grammars

■ General algorithm

1. Arrange the non-terminal symbols in order: $A_1, A_2, A_3, \dots, A_n$
2. For $i=1$ to n do
 - for $j=1$ to $i-1$ do
 - I) replace each production of the form $A_i \rightarrow A_j \gamma$ with the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$
where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current A_j productions
 - II) eliminate the immediate left recursion among the A_i

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 without left recursion.
- Worst-case runtime is exponential.
- Worst-case memory usage is linear.
- Often used in a limited form as recursive descent.

Outline

- Introduction
- Parsing as a Search
- **Predictive Parsing**
 - LL(1) Parser

Predictive Parsing

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

Tradeoffs in Prediction

- Predictive parsers are fast.
 - Many predictive algorithms can be made to run in linear time.
 - Often can be table-driven for extra performance.
- Predictive parsers are weak.
 - Not all grammars can be accepted by predictive parsers.
- Trade expressiveness for speed.

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.

Implementing Predictive Parsing

- Predictive parsing is only possible if we can predict which production to use given some number of lookahead tokens.
 - Increasing the number of lookahead tokens increases the number of grammars we can parse, but complicates the parser.
 - Decreasing the number of lookahead tokens decreases the number of grammars we can parse, but simplifies the parser.

Predictive Parsing

■ Example

E

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

int	+	(int	+	int)
-----	---	---	-----	---	-----	---

Predictive Parsing

■ Example

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$
$\text{int} + (\text{int} + \text{int})$

int	+	(int	+	int)
-----	---	---	-----	---	-----	---

Parsing – Top-Down & Predictive

■ Top-Down Parsing:

Parse tree / derivation of a token string occurs in a top down fashion.

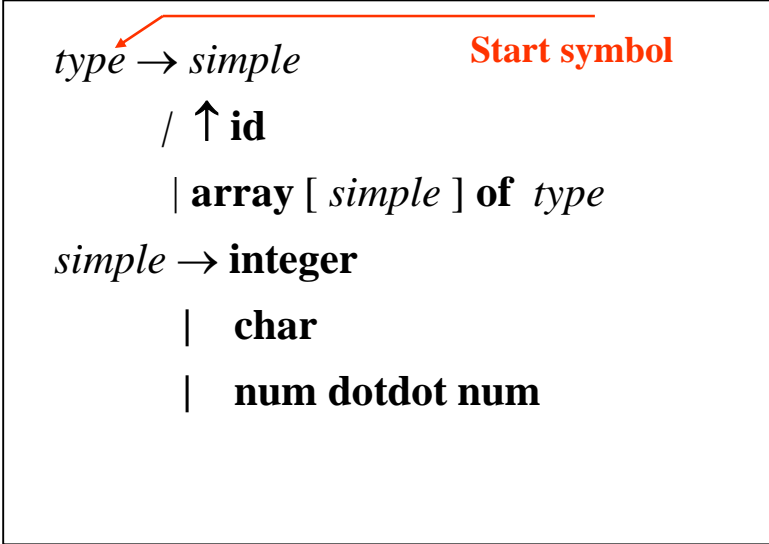
■ Example:

input:

array [num dotdot num] of integer

Parsing would begin with

type → ???



```
type → simple
      / ↑ id
      | array [ simple ] of type
simple → integer
        | char
        | num dotdot num
```

Start symbol

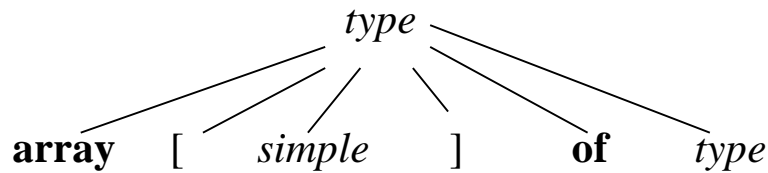
Top-Down Parse

Input : **array [num dotdot num] of integer**

Lookahead symbol

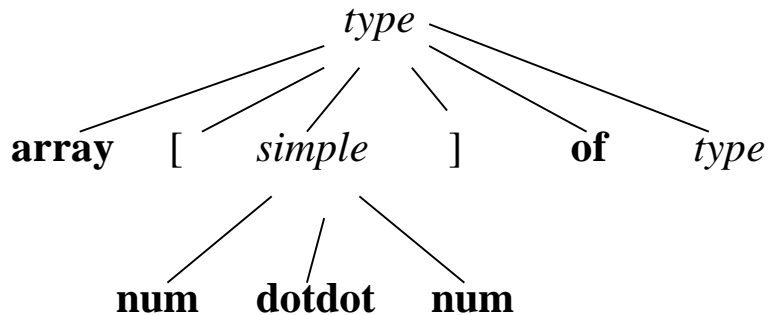
type

?



Input : **array [num dotdot num] of integer**

Lookahead symbol

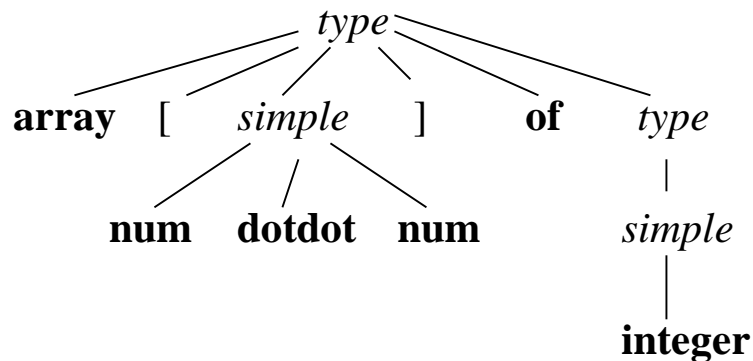
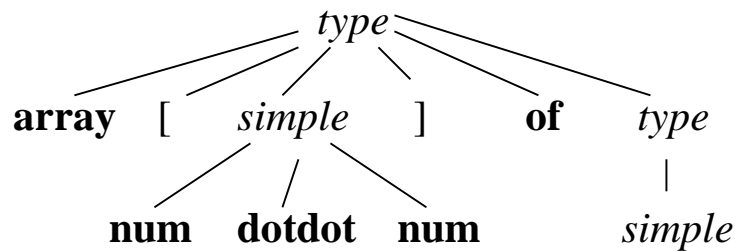


type → *simple* **Start symbol**
 / ↑ **id**
 | **array [simple] of type**
simple → **integer**
 | **char**
 | **num dotdot num**

Top-Down Parse

Lookahead symbol

Input : **array [num dotdot num] of integer**



type → *simple* **Start symbol**
 / ↑ **id**
 | **array [*simple*] of *type***
simple → **integer**
 | **char**
 | **num dotdot num**

Recursive Descent Top-Down Parsing

- Parser operates by attempting to match tokens in the input stream

array [num dotdot num] of integer

```
procedure match ( t : token ) ;  
begin  
    if lookahead = t then  
        lookahead := nexttoken  
    else error  
end ;
```

```
type → simple           Start symbol  
      / ↑ id  
      | array [ simple ] of type  
simple → integer  
      | char  
      | num dotdot num
```

Recursive Descent Top-Down Parsing

array [num dotdot num] of integer

procedure *simple* ;
begin

if *lookahead* = **integer** **then** *match* (**integer**);

else if *lookahead* = **char** **then** *match* (**char**);

else if *lookahead* = **num** **then begin**

match (**num**); *match* (**dotdot**); *match* (**num**)

end

else error

end ;

type → *simple* **Start symbol**
 / ↑ id
 | array [*simple*] of *type*
simple → **integer**
 | **char**
 | **num dotdot num**

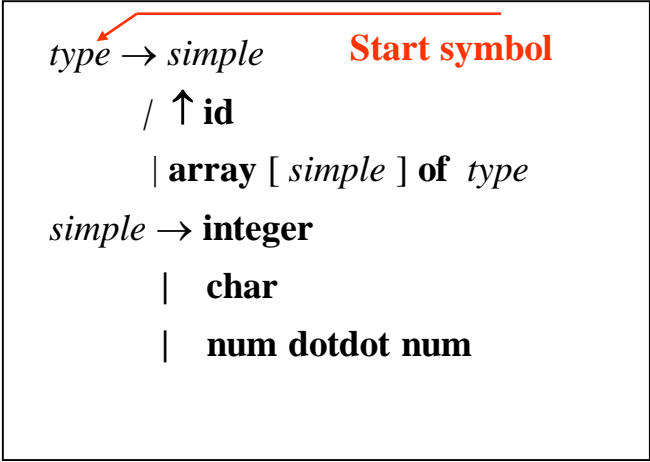
Recursive Descent Top-Down Parsing

array [num dotdot num] of integer

procedure *type* ;
begin

if *lookahead* is in { **integer**, **char**, **num** } **then** *simple*
 else if *lookahead* = '↑' **then begin** *match* ('↑') ; *match*(**id**) **end**
 else if *lookahead* = **array** **then begin**
 match(**array**) ; *match*('[') ; *simple* ; *match*(']') ; *match*(**of**) ; *type*
 end
 else *error*

end ;



type → *simple* **Start symbol**
 / ↑ **id**
 | **array** [*simple*] **of** *type*
simple → **integer**
 | **char**
 | **num dotdot num**

Outline

- Introduction
- Parsing as a Search
- Predictive Parsing
 - LL(1) Parser

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 Table

E → **int**

E → (**E Op E**)

Op → **+**

Op → *****

	int	()	+	*
E	int	(E Op E)			
Op				+	*

LL(1) Parsing

■ Predict step

- If the first symbol of our is a nonterminal. We then look at our parsing table to see what production to use.

■ Match step

- If the first symbol of our guess is now a terminal symbol. We thus match it against the first symbol of the string to parse.

Example: LL(1) Parsing

- (1) **E** \rightarrow **int**
- (2) **E** \rightarrow (**E Op E**)
- (3) **Op** \rightarrow **+**
- (4) **Op** \rightarrow *****

E\$	(int + (int * int))\$
-----	-----------------------

	int	()	+	*
E	1	2			
Op				3	4

Example: LL(1) Parsing

- (1) **E** \rightarrow **int**
- (2) **E** \rightarrow (**E Op E**)
- (3) **Op** \rightarrow **+**
- (4) **Op** \rightarrow *****

	int	()	+	*
E	1	2			
Op				3	4

E\$ (int + (int * int))\$

The \$ symbol is the end-of-input marker and is used by the parser to detect when we have reached the end of the input. It is not a part of the grammar.

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

	int	()	+	*
E	1	2			
Op				3	4

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))\$
)\$)\$
)\$)\$
\$	\$

LL(1) Error Detection

(1) **E** → **int**

(2) **E** → (**E Op E**)

(3) **Op** → +

(4) **Op** → *

E\$	int + int\$
-----	-------------

	int	()	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

- (1) **E** \rightarrow **int**
- (2) **E** \rightarrow (**E Op E**)
- (3) **Op** \rightarrow **+**
- (4) **Op** \rightarrow *****

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

LL(1) Error Detection

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

	int	()	+	*
E	1	2			
Op				3	4

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

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 $T[A, t]$.

A Simple LL(1) Grammar

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

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

TERM → id
 | constant

A Simple LL(1) Grammar

```

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

EXPR   →  TERM -> id      id -> id;
          |  zero? TERM      while not zero? id
          |  not EXPR         do --id;
          |  ++ id
          |  -- id            if not zero? id then
                              if not zero? id then
                                constant -> id;

TERM   →  id
          |  constant

```

Constructing LL(1) Parse Tables

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

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

TERM \rightarrow **id** (9)
 | **constant** (10)

	if	then	while	do	zero?	not	++	--	\rightarrow	id	const	;
STMT												
EXPR												
TERM												

Constructing LL(1) Parse Tables

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

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

TERM	\rightarrow id	(9)
	constant	(10)

	if	then	while	do	zero?	not	++	--	\rightarrow	id	const	;
STMT												
EXPR												
TERM										9	10	

Constructing LL(1) Parse Tables

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

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

TERM \rightarrow **id** (9)
 | **constant** (10)

	if	then	while	do	zero?	not	++	--	\rightarrow	id	const	;
STMT												
EXPR					5	6	7	8				
TERM										9	10	

Constructing LL(1) Parse 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)

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

Constructing LL(1) Parse 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)

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

Constructing LL(1) Parse 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)

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

Constructing LL(1) Parse 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)

	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	

Constructing LL(1) Parse Tables

- 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 .

Selecting the Appropriate Production Rule

■ Basic Tools:

- First: Let α be a string of grammar symbols. $\text{First}(\alpha)$ is the set that includes every terminal that appears leftmost in α or in any string originating from α .

NOTE: If $\alpha \Rightarrow \epsilon$, then ϵ is $\text{First}(\alpha)$.

- Follow: Let A be a non-terminal. $\text{Follow}(A)$ is the set of terminals a that can appear directly to the right of A in some sentential form. ($S \Rightarrow \alpha A a \beta$, for some α and β).

NOTE: If $S \Rightarrow \alpha A$, then $\$$ is $\text{Follow}(A)$.

FIRST Sets

- If a particular nonterminal A derives a string starting with a particular terminal t , we can formalize this with FIRST sets.

$$\text{FIRST}(A) = \{ t \mid A \Rightarrow^* t\omega \text{ for some } \omega \}$$

- Intuitively, $\text{FIRST}(A)$ 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

$$\text{FIRST}(A) = \{ t \mid A \rightarrow t\omega \text{ for some } \omega \}$$

- Then, repeat the following until no changes occur:
 - For each nonterminal A , for each production $A \rightarrow B\omega$, set

$$\text{FIRST}(A) = \text{FIRST}(A) \cup \text{FIRST}(B)$$

- 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 \rightarrow id
 |
 constant

STMT	EXPR	TERM
if while		

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 ++ --	

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 ++ --	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 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	

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

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2									
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	zero?	id constant
while	not	
zero?	++	
not	--	
++	id	
--	constant	
id		
constant		

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

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					5	6	7	8		4	4	
TERM										9	10	

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 $A \rightarrow t\omega$, set $T[A, t] = t\omega$.
- For each production $A \rightarrow B\omega$, set $T[A, t] = B\omega$ for each $t \in \text{FIRST}(B)$.

Expanding our Grammar

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

Expanding our Grammar

STMT →	if EXPR then STMT	(1)	id → id ;
	while EXPR do STMT	(2)	
	EXPR ;	(3)	while not zero? id do --id ;
EXPR →	TERM -> id	(4)	if not zero? id then
	zero? TERM	(5)	if not zero? id then
	not EXPR	(6)	constant → id ;
	++ id	(7)	
	-- id	(8)	
TERM →	id	(9)	
	constant	(10)	
BLOCK →	STMT	(11)	
	{ STMTS }	(12)	
STMTS →	STMT STMTS	(13)	
	ε	(14)	

Expanding our Grammar

STMT	→	if EXPR then BLOCK	(1)	id → id;
		while EXPR do BLOCK	(2)	
		EXPR ;	(3)	while not zero? id do --id;
EXPR	→	TERM -> id	(4)	if not zero? id then
		zero? TERM	(5)	if not zero? id then
		not EXPR	(6)	constant → id;
		++ id	(7)	
		-- id	(8)	
TERM	→	id	(9)	
		constant	(10)	
BLOCK	→	STMT	(11)	
		{ STMTS }	(12)	
STMTS	→	STMT STMTS	(13)	
		ε	(14)	

Expanding our Grammar

STMT	→	if EXPR then BLOCK while EXPR do BLOCK EXPR ;	(1) id → id ; (2) (3) while not zero? id do --id;
EXPR	→	TERM -> id zero? TERM not EXPR ++ id -- id	(4) if not zero? id then (5) if not zero? id then (6) constant → id; (7) (8) if zero? id then
TERM	→	id constant	while zero? id do { (9) constant → id; (10) constant → id;
BLOCK	→	STMT { STMTS }	(11) } (12)
STMTS	→	STMT STMTS ε	(13) (14)

LL(1) with ϵ -Productions

- Computation of FIRST is different.
 - What if the first nonterminal in a production can produce ϵ ?

- Building the table is different.
 - What action do you take if the correct production produces the empty string?

FIRST Sets with ϵ

Num \rightarrow **Sign Digits**
Sign \rightarrow **+** | **-** | **ϵ**
Digits \rightarrow **Digit More**
More \rightarrow **Digits** | **ϵ**
Digit \rightarrow **0** | **1** | **2** | ... | **9**

Num	Sign	Digit	Digits	More

FIRST Sets with ϵ

Num \rightarrow **Sign Digits**
Sign \rightarrow + | - | ϵ
Digits \rightarrow **Digit More**
More \rightarrow **Digits** | ϵ
Digit \rightarrow 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
	+ -	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 1 6 2 7 3 8 4 9		

FIRST Sets with ϵ

Num \rightarrow Sign Digits
 Sign $\rightarrow + \mid - \mid \epsilon$
Digits \rightarrow **Digit More**
 More \rightarrow Digits $\mid \epsilon$
 Digit $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

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	

FIRST Sets with ϵ

Num \rightarrow Sign Digits
 Sign $\rightarrow + \mid - \mid \epsilon$
 Digits \rightarrow Digit More
More \rightarrow **Digits** $\mid \epsilon$
 Digit $\rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

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

FIRST Sets with ϵ

Num \rightarrow **Sign Digits**
Sign \rightarrow **+** | **-** | ϵ
Digits \rightarrow **Digit More**
More \rightarrow **Digits** | ϵ
Digit \rightarrow **0** | **1** | **2** | ... | **9**

Num	Sign	Digit	Digits	More
+ -	+ - 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

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:

$$A \Rightarrow Bt \Rightarrow t$$

- So $t \in \text{FIRST}(A)$.

FIRST Computation with ε

- Initially, for all nonterminals A , set

$$\text{FIRST}(A) = \{ t \mid A \rightarrow t\omega \text{ for some } \omega \}$$

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

$$\text{FIRST}(A) = \text{FIRST}(A) \cup \{ t \}$$

- For each production $A \rightarrow \alpha B \omega$, where α is string of nonterminals whose FIRST sets contain ε , set

$$\text{FIRST}(A) = \text{FIRST}(A) \cup (\text{FIRST}(B) - \{ \varepsilon \}).$$

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 $\text{FIRST}(\omega)$ as follows:
 - $\text{FIRST}(\varepsilon) = \{ \varepsilon \}$
 - $\text{FIRST}(t\omega) = \{ t \}$
 - If $\varepsilon \notin \text{FIRST}(A)$:
 - $\text{FIRST}(A\omega) = \text{FIRST}(A)$
 - If $\varepsilon \in \text{FIRST}(A)$:
 - $\text{FIRST}(A\omega) = (\text{FIRST}(A) - \{ \varepsilon \}) \cup \text{FIRST}(\omega)$

FIRST Computation with ε

- Initially, for all nonterminals A , set

$$\text{FIRST}(A) = \{ t \mid A \rightarrow t\omega \text{ for some } \omega \}$$

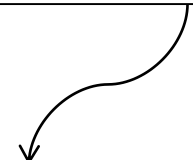
- For all nonterminals A where $A \rightarrow \varepsilon$ is a production, add ε to $\text{FIRST}(A)$.
- Repeat the following until no changes occur:
 - For each production $A \rightarrow \alpha$, set

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

LL(1) Tables with ϵ

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

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



	+	-	#	\$
Num				
Sign				
Digits				
More				
Digit				

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

Num → **Sign Digits**

Sign → **+** | **-** | **ϵ**

Digits → **Digit More**

More → **Digits** | **ϵ**

Digit → **0** | **1** | ... | **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	

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

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

Num → **Sign Digits**

Sign → **+** | **-** | ϵ

Digits → **Digit More**

More → **Digits** | ϵ

Digit → **0** | **1** | ... | **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	

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

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

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

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

LL(1) Tables with ϵ

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

	+	-	#	\$
Num	Sign Digits	Sign Digits	Sign Digits	
Sign	+	-	ϵ	
Digits			Digits More	
More			Digits	ϵ
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:

$$\text{FOLLOW}(A) = \{ t \mid S \Rightarrow^* \alpha A t \omega \text{ for some } \alpha, \omega \}$$

where S is the start symbol of the grammar.

- Informally, every nonterminal that can ever come after A in a derivation.

Computation of FOLLOW Sets

- Initially, for each nonterminal A, set

$$\text{FOLLOW}(A) = \{ t \mid B \rightarrow \alpha A t \omega \text{ is a production} \}$$

- Add \$ to FOLLOW(S), where S is the start symbol.

- Repeat the following until no changes occur:

- If $B \rightarrow \alpha A \omega$ is a production, set

$$\text{FOLLOW}(A) = \text{FOLLOW}(A) \cup \text{FIRST}(\omega) - \{ \varepsilon \}$$

- If $B \rightarrow \alpha A \omega$ is a production and $\varepsilon \in \text{FIRST}(\omega)$, set

$$\text{FOLLOW}(A) = \text{FOLLOW}(A) \cup \text{FOLLOW}(B).$$

The Final LL(1) Table Algorithm

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

An Egregious Abuse of Notation

- Compute $\text{FIRST}(A)$ and $\text{FOLLOW}(A)$ for all nonterminals A
- For each rule $A \rightarrow \omega$, for each terminal $t \in \text{FIRST}(\omega\text{FOLLOW}(A))$, set $T[A, t] = \omega$

LL(1) Construction

- The Limits of LL(1)
 - Some grammars are Not LL(1)
- The Strengths of LL(1)
 - LL(1) is straightforward
 - LL(1) is fast

LL(1) Limitation

- Left-recursive grammars:

- Example:

$$A \rightarrow Ab \mid c$$

- FIRST(A) ?
- Parse table ?

LL(1) Limitation

■ Left-recursive grammars:

■ Example:

$$A \rightarrow Ab \mid c$$

● $\text{FIRST}(A) = \{c\}$

● Parse table:

	b	c
A		$A \rightarrow Ab$ $A \rightarrow c$

- Cannot uniquely predict production!
- This is called a FIRST/FIRST conflict.

LL(1) Limitation

■ Left-recursive grammars:

■ Example:

$$A \rightarrow Ab \mid c$$

● $\text{FIRST}(A) = \{c\}$

● Parse table:

	b	c
A		$A \rightarrow Ab$ $A \rightarrow c$

- Cannot uniquely predict production!
- This is called a FIRST/FIRST conflict.

**Solution:
Eliminating
Left Recursion**

LL(1) Limitation

■ Left-recursive grammars:

■ Example:

$$E \rightarrow T$$

$$E \rightarrow T + E$$

$$T \rightarrow \text{int}$$

$$T \rightarrow (E)$$

- $\text{FIRST}(E) = \{ \text{int}, (\}$

- $\text{FIRST}(T) = \{ \text{int}, (\}$

LL(1) Limitation

■ Left-recursive grammars:

■ Example:

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

Can not predict
which rule to use

- $\text{FIRST}(E) = \{ \text{int}, (\}$
- $\text{FIRST}(T) = \{ \text{int}, (\}$

Solution:
Left Factoring

LL(1) Limitation

- Left-recursive grammars:

- Example:

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

LL(1) Limitation

- Left-recursive grammars:

- Example:

$$E \rightarrow T \textcolor{red}{\epsilon}$$

$$E \rightarrow T + E$$

$$T \rightarrow \text{int}$$

$$T \rightarrow (E)$$

LL(1) Limitation

- Left-recursive grammars:

- Example:

$$E \rightarrow T \textbf{Y}$$
$$T \rightarrow \text{int}$$
$$T \rightarrow (E)$$

LL(1) Limitation

- Left-recursive grammars:

- Example:

$$E \rightarrow T \textbf{Y}$$
$$T \rightarrow \text{int}$$
$$T \rightarrow (E)$$
$$\textbf{Y} \rightarrow + \textbf{E}$$
$$\textbf{Y} \rightarrow \epsilon$$

Example: LL(1) with Left Factoring

E → TY	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
FOLLOW		
E	T	Y

Example: LL(1) with Left Factoring

E → TY	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
	int (
FOLLOW		
E	T	Y

Example: LL(1) with Left Factoring

E → T Y	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
	int (+ ε
FOLLOW		
E	T	Y

Example: LL(1) with Left Factoring

E → TY	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y

Example: LL(1) with Left Factoring

E → T Y	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y
\$)		

Example: LL(1) with Left Factoring

E → TY	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y
\$)	+	

Example: LL(1) with Left Factoring

E → T Y	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y
\$)	+	\$)

Example: LL(1) with Left Factoring

E → T Y	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y
\$)	+ \$)	\$)



Example: LL(1) with Left Factoring

E → TY	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y
\$)	+ \$)	\$)

	int	()	+	\$
E					
T					
Y					

Example: LL(1) with Left Factoring

E → T Y 	1
T → int	2
T → (E)	3
Y → + E	4
Y → ε 	5

FIRST		
E	T	Y
int (int (+ ε
FOLLOW		
E	T	Y
\$)	+ \$)	\$)

	int	()	+	\$
E	1	1			
T	2	3			
Y			5	4	5

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.

The Strengths of LL(1)

- LL(1) is Fast
- Both table-driven LL(1) and recursive descent-powered LL(1) are fast.
- Can parse in $O(n |G|)$ time, where n is the length of the string and $|G|$ is the size of the grammar.

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.

Reading

- Aho2, Sections 4.1; 4.2; 4.3.1; 4.3.2; (see also pp.56-60)
- Aho1, pp. 160-175
- Hunter, pp. 21-44
- Grune pp.34-40; 110-115
- Cooper, pp.73-89.

Question?