Parsing LL(1) Grammars

Announcements

- Final Project
  - Find a partner ASAP! E-mail me if you can't find one.
  - We will post additional resources on Piazza.
  - Parsing:
    - You will extend the parser, which is written in LALRPOP, a parser generator written for Rust.

Parsing with CFGs?

- Regexp -> DFA
- Linear time implementation
- Lexer/Scanner Generators

- CFG -> Pushdown Automata
- Best algorithm: O(n^3)
- Not feasible

Parsing with *Some* CFGs

- Carve out a subset of CFGs that *can* be implemented in linear time.
- Practical: Most programming languages fall into these categories, and that's no accident: many languages are *designed* with these restrictions in mind.

- Today: LL(1)
  - Fast, simple enough for hand-written parsers
  - Too weak for many practical languages

- Next Time: LR(1)
  - Fast, more expressive than LL(1)
  - Too complicated for hand-written, instead use parser generators

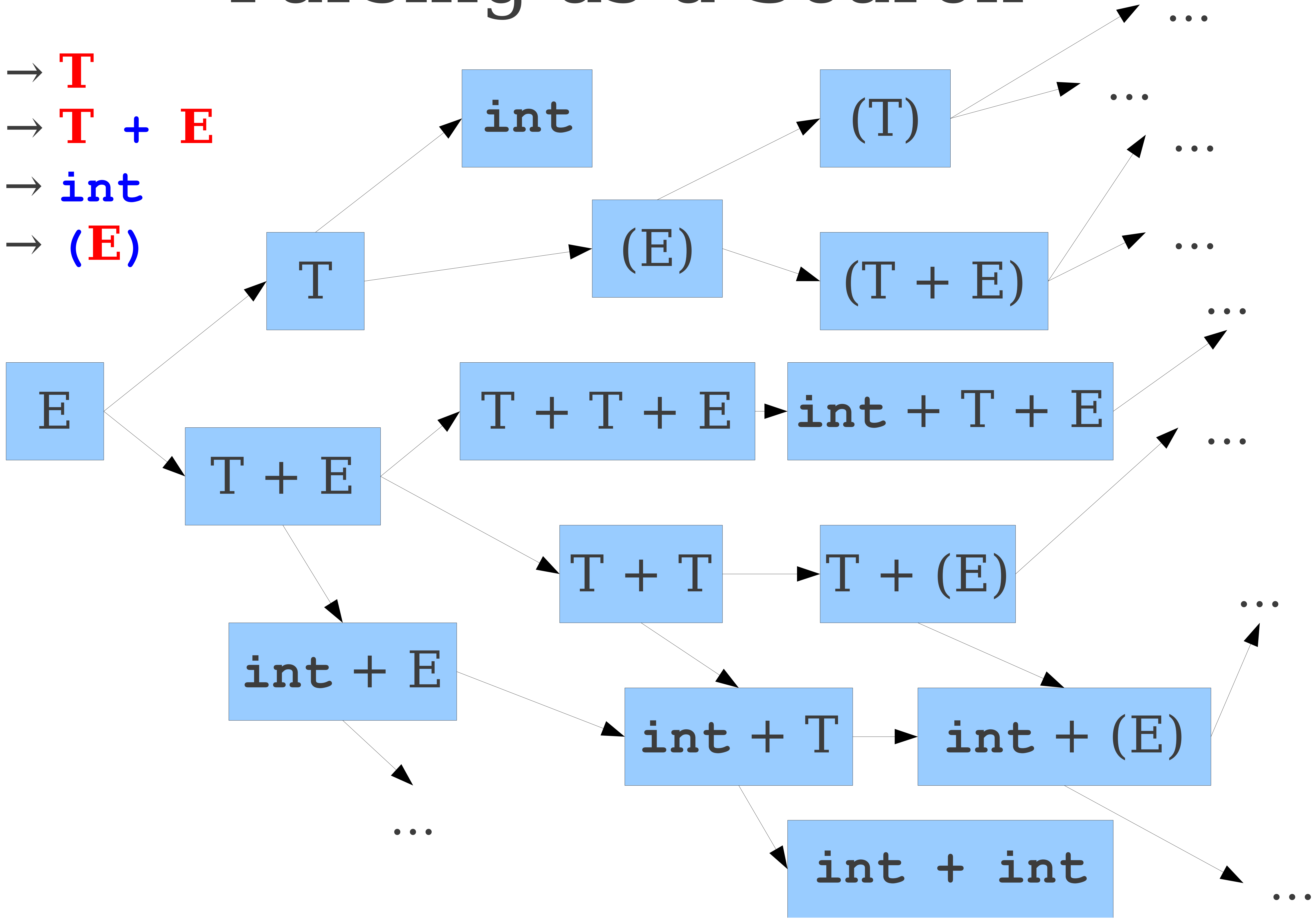
"Top-Down" Parsing

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 $\boldsymbol{\alpha}$ to node $\boldsymbol{\beta}$ iff $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta}$.

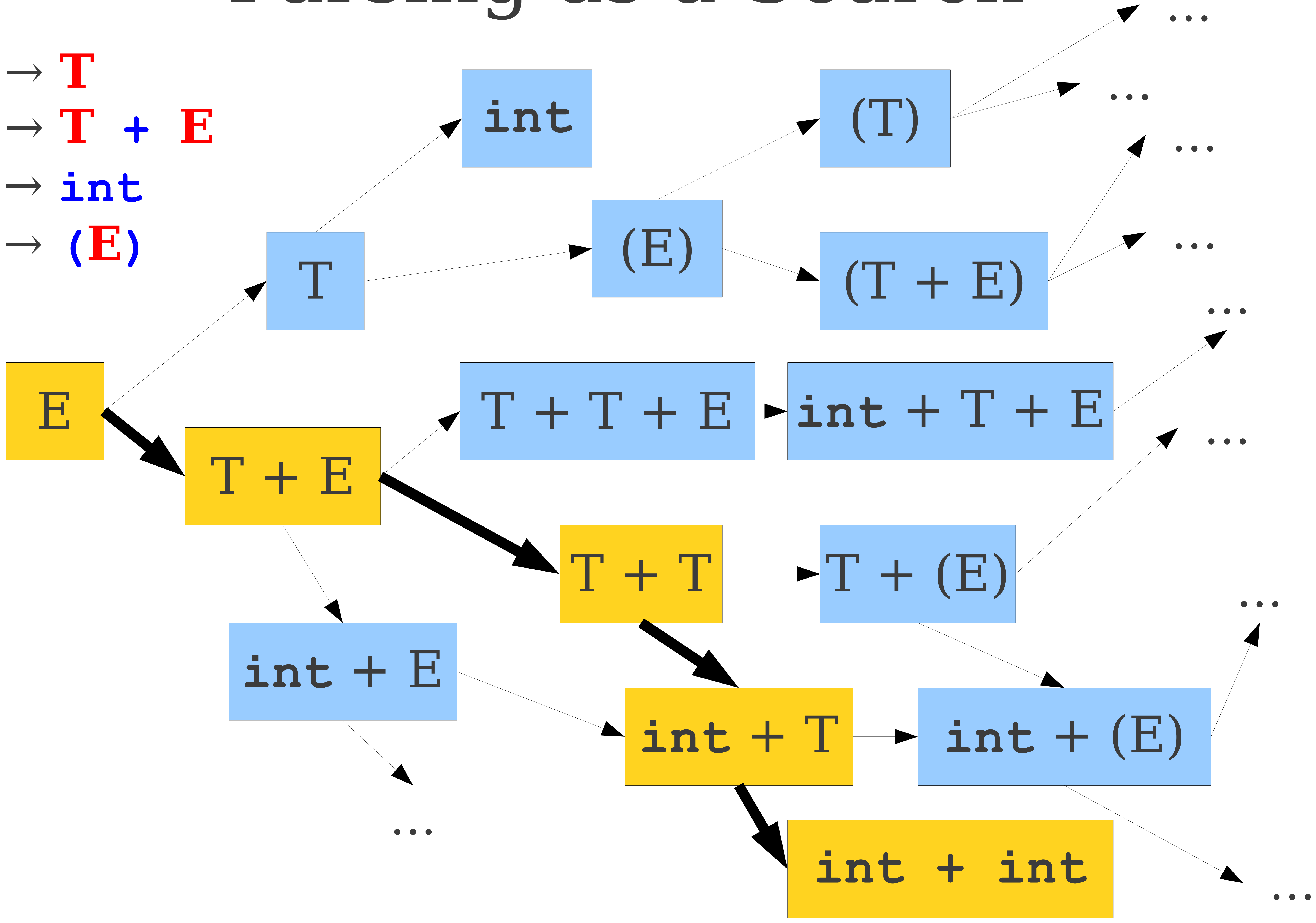
Parsing as a Search

E → **T**
E → **T + E**
T → `int`
T → **(E)**



Parsing as a Search

E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`



Recursive Descent

- Pseudo-algorithm:
  - For each non-terminal A, define a function matchA
    - non-deterministically choose a production A → ω
    - traverse the RHS ω from left to right
      - Terminal b: If the next character is a b, pop it off, otherwise fail
      - NonTerminal B: make a recursive call to matchB
  - Run matchS where S is the start symbol, if the input is all consumed, succeed.

LL(1) Grammars

- Special class of CFGs that can be implemented by recursive descent with no non-determinism
  - Also called "predictive parsing"
- LL(1)
  - L: Left-to-right
  - L: Leftmost derivation
  - 1: 1-token lookahead

LL(1) Grammars

- Intuitively, LL(1) grammars ensure we can remove the non-determinism by using **lookahead**
    - Look at **1** token (without consuming it) to determine which production to choose
    - Not every grammar can be implemented this way

LL(1) Grammars

- Weaknesses
  - LL(1) grammars are inherently unambiguous
  - LL(1) grammars are restrictive
- Strengths
  - FAST
  - Easy to implement manually

Recursive Descent for LL(1)

- Pseudo-algorithm:
    - For each non-terminal A, define a function matchA
        - Look at the first token (treat end of string as a special $ token),
            - At most one production A → ω can possibly match. If none do, fail
            - traverse the RHS ω from left to right
                - Terminal b: If the next character is a b, pop it off, otherwise fail
                - NonTerminal B: make a recursive call to matchB
    - Run matchS where S is the start symbol. If the entire string is consumed, succeed

LL(1)

- Formal definition:
  - Whenever we have two distinct rules A -> alpha | beta
    - First(alpha) disjoint First(beta)
    - Nullable(alpha) exclusive or Nullable(beta)
    - (If Nullable(alpha) then First(beta) disjoint Follow(A)) and vice-versa

LL(1) Parse Tables

LL(1) Parse Tables

E → `int`

E → `(`**E Op E**`)`

Op → `+`

Op → `*`

LL(1) Parse Tables

E → `int`
E → `(`**E Op E**`)`
Op → `+`
Op → `*`

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

LL(1) Parsing

`(int + (int * int))`

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

LL(1) Parsing

E	(int + (int * int))

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

LL(1) Parsing

E	(int + (int * int))

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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.

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

The first symbol of our guess is a nonterminal. We then look at our parsing table to see what production to use.

This is called a **predict** step.

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

This is called a match step.

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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))$

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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))$

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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))$

LL(1) Parsing

(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) Parsing

(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) Parsing

(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) Parsing

(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** → **`*`**

`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 → *

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** → *****

E$	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 $	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 $	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))$

	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	(	)	+	*
E	1	2			
Op				3	4

E$	(int (int))$

LL(1) Error Detection, Part II

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

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

	`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	(	)	+	*
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	(	)	+	*
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))$

	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

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

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

TERM → `id`
  | `constant`

```
id -> id;
```

```
while not zero? id
   do --id;
```

```
if not zero? id then
   if not zero? id then
       constant -> id;
```

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												
TERM												

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												
TERM												

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												
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												
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				
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				
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												
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									
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				
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

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	

The Limits of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

	b	c
A		**A** → **A**`b` **A** → `c`

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **Ab** | **c**

- FIRST(**A**) = {**c**}
- However, we cannot build an LL(1) parse table.
- **Why?**

	b	c
A		**A** → **Ab** **A** → **c**

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

Eliminating Left Recursion

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

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

- This will produce $\boldsymbol{\alpha}$ followed by some number of $\boldsymbol{\omega}$'s.
- Can rewrite the grammar as

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

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

Summary

- CFGs are too general for efficient parsing algorithms
  - Instead, add practical restrictions
- LL(1)
  - Parseable by recursive descent algorithm
  - LL(1) restriction ensures no backtracking, only 1-token lookahead needed
  - Fast, and simple to implement, but weak