Compiler Design

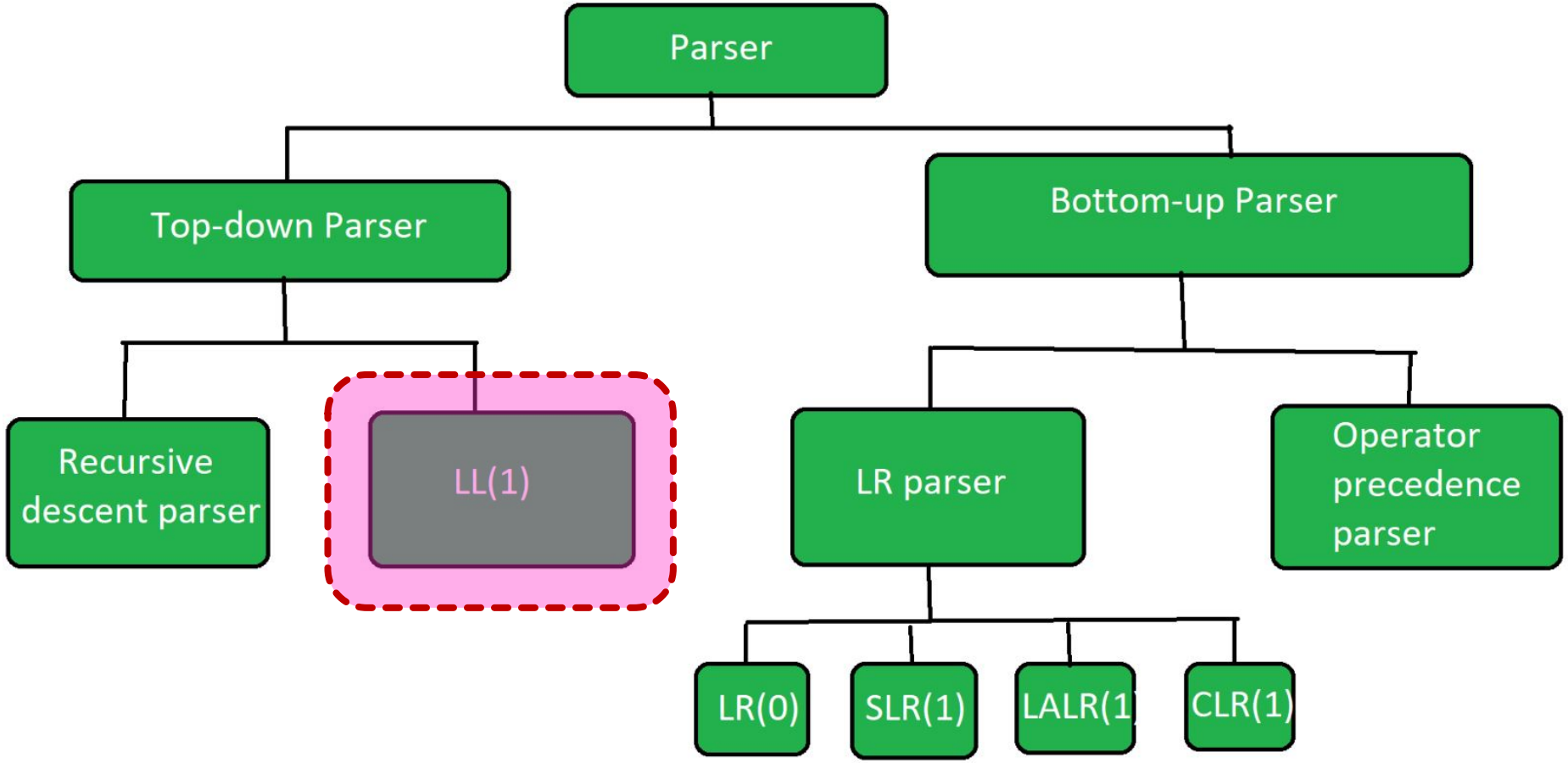
Lecture 7: Syntax Analysis III (Parsing)

Sahar Selim

Agenda

- Types of Parsing
- Top-Down Parsing
  - LL(1) Parsing
  - Parse Table

Types of Parsers



Top-down Parser

- Top-down parser is the parser which generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals
  - It starts from the start symbol and ends on the terminals.
- It uses left most derivation.
- It is classified into 2 types:
  A. Recursive descent parser
  B. Non-recursive descent parser

Top-down Parser

A. Recursive descent parser:

- It is also known as Brute force parser or the with backtracking parser.
- It basically generates the parse tree by using brute force and backtracking.

Top-down Parser

B. Non-recursive descent parser:

- It is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser.
- It uses parsing table to generate the parse tree instead of backtracking

Bottom-up Parser

- It generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol.
- It uses reverse of the right most derivation.
- Bottom-up parser is classified into 2 types:
  A. LR parser
  B. Operator precedence parser

Bottom-up Parser

A. LR parser:

- It is the bottom-up parser which generates the parse tree for the given string by using unambiguous grammar.
- It follows **reverse of right most derivation**.
- LR parser is of 4 types: LR(0), SLR(1), LALR(1), and CLR(1)

Bottom-up Parser

B. Operator precedence parser:

- It builds a parse tree for a grammar that doesn't contain epsilon productions and does not contain two adjacent non-terminals on R.H.S. of any production.

Example

- S → aABe
- A → Abc | b
- B → d

- String: abbcde

Example

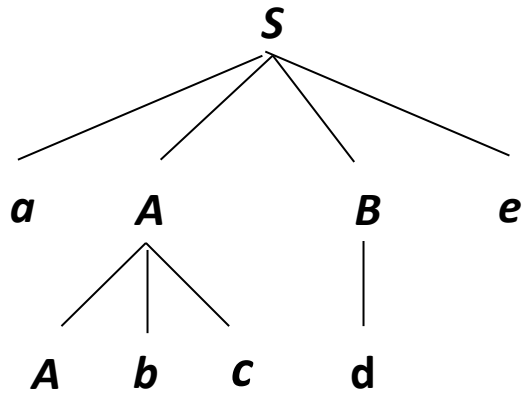
- S → aABe
- A → Abc | b
- B → d

- String: abbcde

Left Most Derivation

What to use

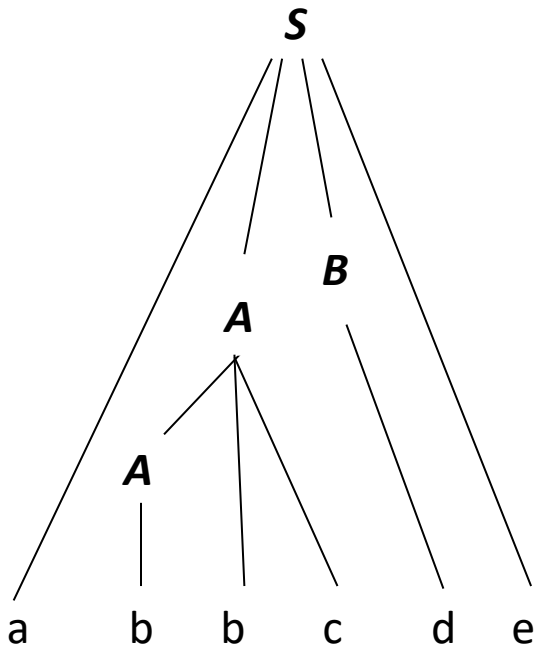
Top-down



Right Most Derivation

When to reduce

Bottom-up



Top-down Parsing

Concept of Top-Down Parsing (1)

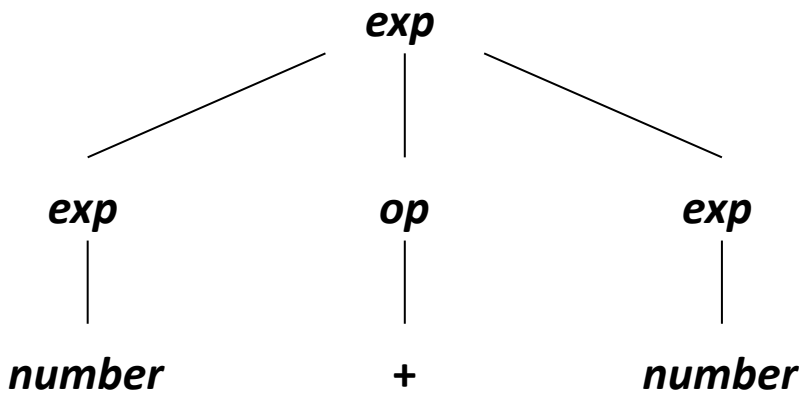
- It generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals.
- It parses an input string of tokens by *tracing out the steps* in a leftmost derivation.
  - It starts from the start symbol and ends on the terminals.
  - And the implied traversal of the parse tree is a preorder traversal and, thus, occurs from the **root** to the **leaves**.

Concept of Top-Down Parsing (2)

- The example: number + number, and corresponds to the following parse tree
- The parse tree corresponds to the leftmost derivations:

(1) *exp* => *exp op exp*
(2) => ***number*** *op exp*
(3) => ***number*** + *exp*
(4) => ***number*** + ***number***

exp → exp op exp | number
op → + | –



Two forms of Top-Down Parsers

- **Predictive parsers**
  - attempts to predict the next construction in the input string using one or more look-ahead tokens
- **Backtracking parsers**
  - try different possibilities for a parse of the input, backing up an arbitrary amount in the input if one possibility fails.
  - It is more powerful but much slower, unsuitable for practical compilers.

Predictive Parsing

Predictive Parsing

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

Tradeoffs in Prediction

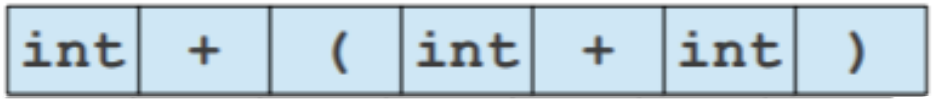
- Predictive parsers are fast.
- Many predictive algorithms can be made to run in linear time.
- Often can be table-driven for extra performance.
- Predictive parsers are weak.
- Not all grammars can be accepted by predictive parsers.
- Trade expressiveness for speed.

Exploiting Lookahead

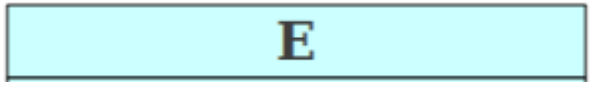
- Given just the start symbol, how do you know which productions to use to get to the input program?
- Idea: Use lookahead tokens.
- When trying to decide which production to use, look at some number of tokens of the input to help make the decision.

Example

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



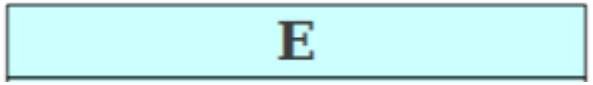
Example



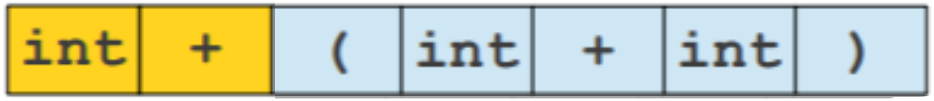
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

int	+	(	int	+	int	)

Example



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



Example

E
T + E

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

int	+	(	int	+	int	)

Example

E
T + E

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

int	+	(	int	+	int	)

Example

E
T + E

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

int	+	(	int	+	int	)

Example

E
T + E
int + E

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

int	+	(	int	+	int	)

Example

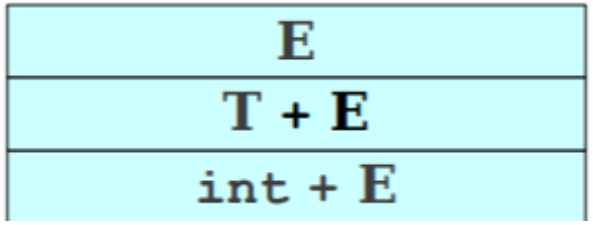
E
T + E
int + E

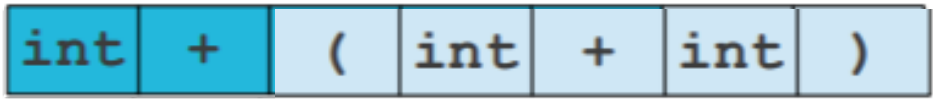
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

int	+	(	int	+	int	)

Example

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





Example

E
T + E
int + E

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

int	+	(	int	+	int	)

Example

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

E
T + E
int + E
int + T

int	+	(	int	+	int	)

Example

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

E
T + E
int + E
int + T

int	+	(	int	+	int	)

Example

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

E
T + E
int + E
int + T

int	+	(	int	+	int	)

Example

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

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

int	+	(	int	+	int	)

Example

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

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

int	+	(	int	+	int	)

Example

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

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

int	+	(	int	+	int	)

Example

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

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

Example

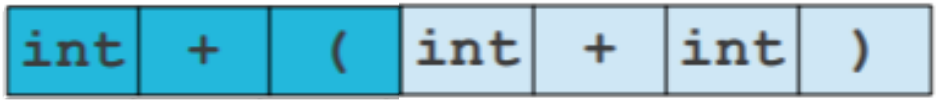
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

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

Example

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

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



Example

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

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

Example

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

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

int	+	(	int	+	int	)

Example

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

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

int	+	(	int	+	int	)

Example

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

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

int	+	(	int	+	int	)

Example

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

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

Example

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

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

Example

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

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

int	+	(	int	+	int	)

Example

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

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

Example

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

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

int	+	(	int	+	int	)

Example

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

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

int	+	(	int	+	int	)


Q
A

LL(1) Predictive Parsing

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.
- It uses parsing table to generate the parse tree.

LL(1) Parse Tables

$$E \rightarrow \texttt{int}$$
$$E \rightarrow (E\ Op\ E)$$
$$Op \rightarrow +$$
$$Op \rightarrow *$$

LL(1) Parse Tables

$$E \rightarrow \texttt{int}$$
$$E \rightarrow (E\ Op\ E)$$
$$Op \rightarrow +$$
$$Op \rightarrow *$$

Non-Terminals

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

Terminals

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))$

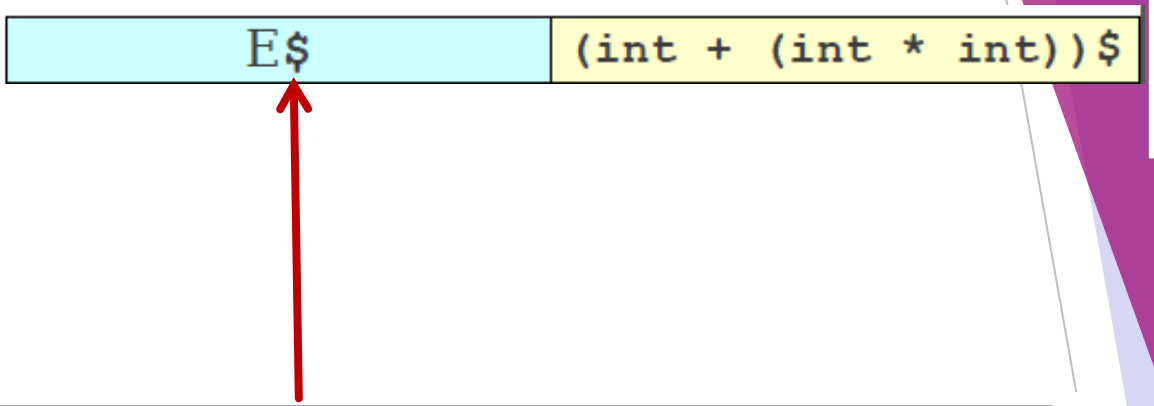
(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

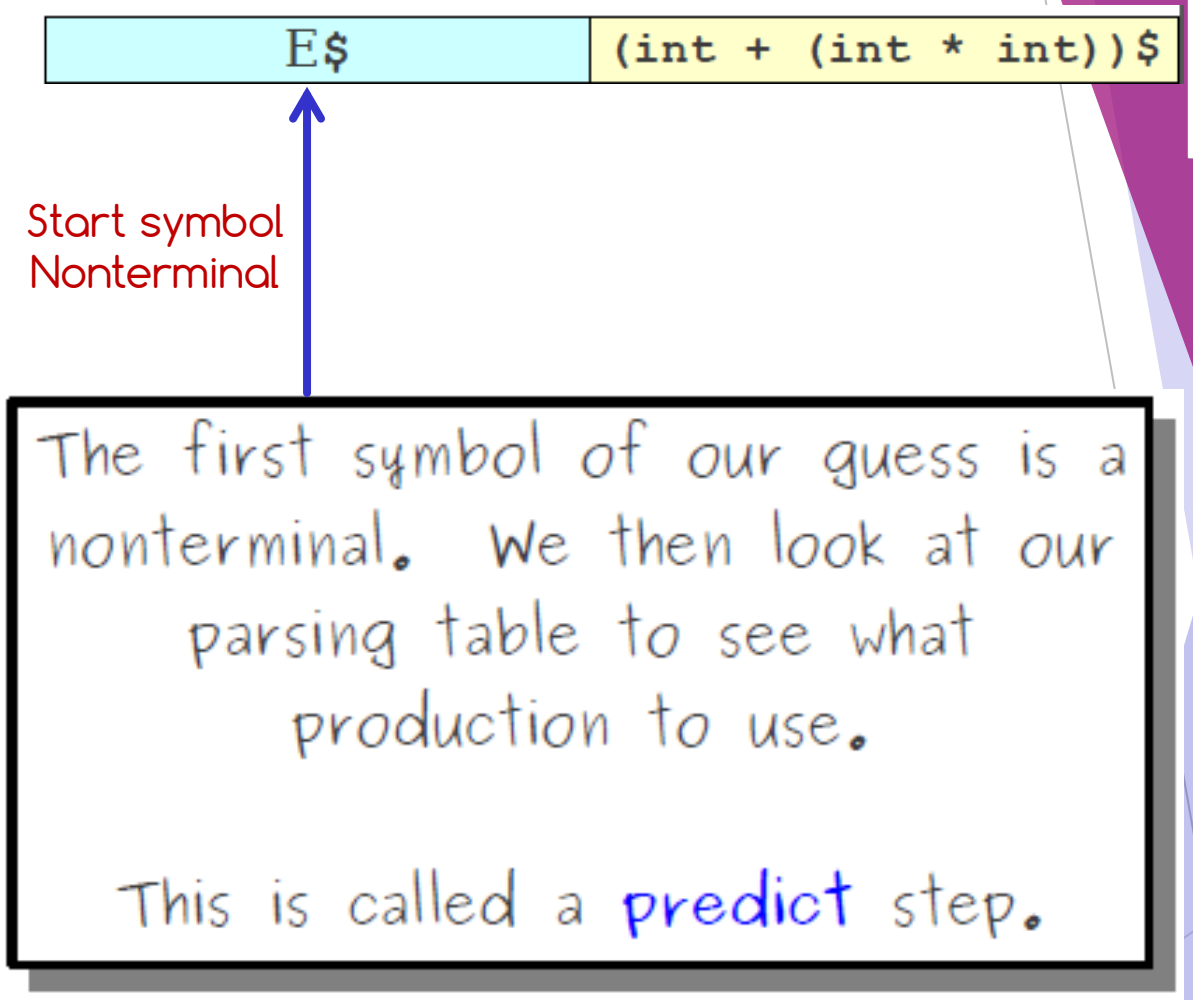


The $ symbol is the end-of-input marker and is used by the parser to detect when we have reached the end of the input. It is not a part of the grammar.

LL(1) Parsing

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

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

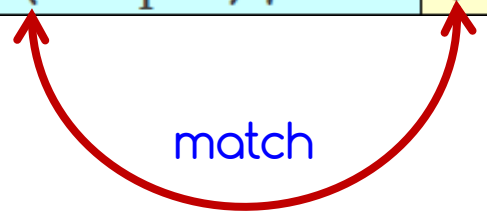
	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

(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))$



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

(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))$

predict

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

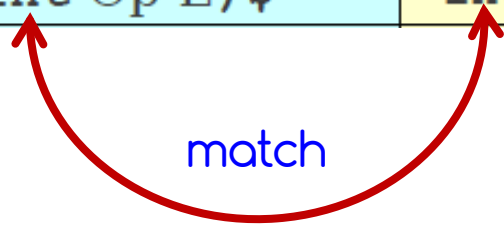
E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$

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))$



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))$

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))$

	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))$

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))$

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))$

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 – Example 1

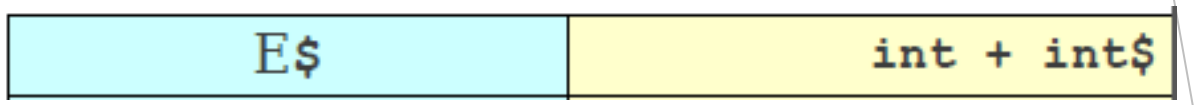
(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

```
int + int$
```

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection – Example 1

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



	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection – Example 1

(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 – Example 1

(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 – Example 1

(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 – Example 1

(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 – Example 2

(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 – Example 2

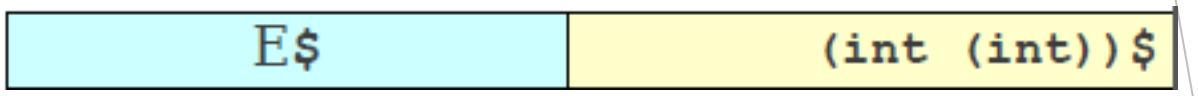
(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

E$	(int (int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection – Example 2

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



	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection – Example 2

(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 – Example 2

(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 – Example 2

(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 – Example 2

(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 – Example 2

(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 – Example 2

(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].

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										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		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	


Q
A

Next Lecture

1. First Sets
2. Follow Sets
3. Left factoring

Useful links

- Introduction to parsers and LL(1) parsing

References of this lecture

- Presentation slides of the book: COMPILER CONSTRUCTION, Principles and Practice, by Kenneth C. Louden
- Credits for Dr. Sally Saad, Prof. Mostafa Aref, Dr. Islam Hegazy, and Dr. Abd ElAziz for help in content preparation and aggregation (FCIS-ASU)

See you next lecture