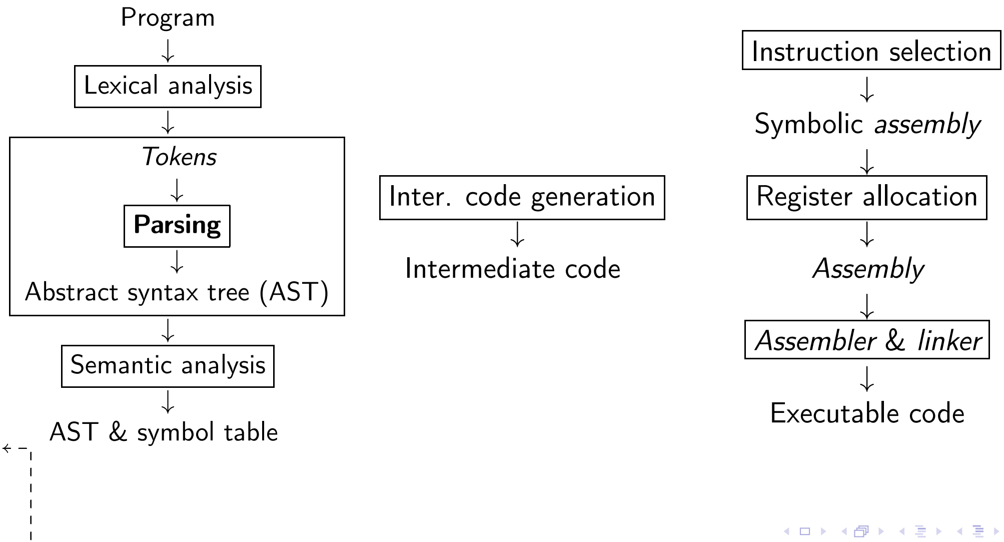
Compilers
Top-down Parsing

LEIC

FEUP-FCUP

2022

Compiler


Program
Lexical analysis
Tokens
Parsing
Abstract syntax tree (AST)
Semantic analysis
AST & symbol table
Inter. code generation
Intermediate code
Instruction selection
Symbolic *assembly*
Register allocation
Assembly
Assembler & linker
Executable code

This lecture

Parsing

Recursive descent parsing

LL Parsing

Parsing

- Build an AST from a list of *tokens*
  (or reject the program with a syntax error output)
- Parsing:

  top-down beging by the root (non-terminal initial symbol S) and find the leftmost derivation.

  bottom-up begin by the tokens and find the reversed rightmost derivation.
- *Top-down* parsing:
  - *recursive descent parsing*
  - predictive *parsing* (LL(1))

This lecture

Parsing

Recursive descent parsing

LL Parsing

Implemented directly in a programming language:

- Each non-terminal symbol corresponds to a function (or method)
- Each production correspond to a different case
  (if the production is recursive, so it is the function)
- Consume tokens from left to right
- Decide which production to use using the next token

Example

Programming Language:

```
begin
 if 1=1 then
    begin
    print 0=11 ;  print 123=4
    end
 else
    print 11=42
end
```

Example (cont.)

Grammar:

$$
\begin{array}{ll}
S \rightarrow \text{if } E \text{ then } S \text{ else } S & L \rightarrow \text{end} \\
S \rightarrow \text{begin } S\ L & L \rightarrow \ ;\ S\ L \\
S \rightarrow \text{print } E & E \rightarrow \text{num} = \text{num}
\end{array}
$$

Implementation in C/Java

- *parsing* consumes *tokens* from the standard input

```
Token getToken(void);  // read next token from the standard input
```

- Keep *look-ahead* token in a global variable:

```
Token next;               // next token
void advance(void) {  // goes to next token
   next = getToken();
}
```

- *parsing* algorithm decides what to do using the *token* and the *look-ahead*
- `consume(...)` consumes a specific *token*

Implementation in C/Java (cont.)

```
void parse_S(void) {
  switch(next) {
  case IF:
    advance(); parse_E(); consume(THEN);  parse_S();
    consume(ELSE);  parse_S();
    break;
  case BEGIN:
    advance();  parse_S();  parse_L();
    break;
  case PRINT:
    advance(); parse_E();
    break;
  default:
    error("syntax error");
  }
}
```

Implementation in C/Java (cont.)

```
void parse_E(void) {
  consume(NUM);  consume(EQUAL); consume(NUM);
}

void parse_L(void) {
  switch(next) {
  case END:
    advance();
    break;
  case SEMI:
    advance(); parse_S(); parse_L();
    break;
  default:
    error("syntax error");
  }
}
```

Stop

- The program terminates without redundant *tokens*
- The parser must know when to finish
- Add a special *token* \$ meaning the end of file
- Add a new production rule $S' \rightarrow S\$$
- S' is now the new initial symbol

```
void accepted(void) {
   parse_S();
   consume(EOF);
}
```

This lecture

Parsing

Recursive descent parsing

LL Parsing

LL Parsing

- Parsing by recursive descent chooses the production rule based on the next terminal symbol
- We may need to rewrite the grammar to know what is the next terminal symbol

Example

$$E \to E + T \qquad T \to T * F \qquad F \to \text{num}$$
$$E \to E - T \qquad T \to T \,/\, F \qquad F \to (E)$$
$$E \to T \qquad T \to F$$

Problems:

- How do we choose which rule to use from E and T?
- How to avoid a cycle caused by left recursion in E and T ?

Eliminar recursão à esquerda

Consider the following grammar:

$$E \rightarrow E \; + \; T$$
$$E \rightarrow T$$

E produces sums of terms, i.e. $E \Rightarrow^* T + T + \cdots + T$.

Let us define an equivalent grammar adding a new non-terminal symbol E':

$$E \rightarrow T \; E'$$

$$E' \rightarrow + \; T \; E'$$
$$E' \rightarrow \varepsilon$$

This grammar is right recursive.

Eliminar recursão à esquerda (cont.)

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow \varepsilon$$

- Rules in E' have recursion on the right and not on the left
- We now may decide which rule to use based on the next symbol:
- $+$ use $E' \rightarrow +\ T\ E'$
  otherwise use $E' \rightarrow \varepsilon$

Eliminar recursão à esquerda (cont.)

Applying the same transformation to the initial grammar we get:

$$
\begin{array}{lll}
E \to T\ E' & T \to F\ T' & \\
E' \to +\ T\ E' & T' \to *\ F\ T' & F \to \mathsf{num} \\
E' \to -\ T\ E' & T' \to /\ F\ T' & F \to (\ E\ ) \\
E' \to \varepsilon & T' \to \varepsilon &
\end{array}
$$

We now may define a recursive descent parser.

Exercise: implement a recursive descent *parser* for this grammar.

Grammars $LL(1)$

- These grammars belong to a class called $LL(1)$: *Left-to-right parse, Leftmost derivation, 1-symbol look-ahead*
- LL(1) contains all the grammars that may be implemented using recursive descent
- $LL(k)$ means: *Left-to-right parse, Leftmost derivation, k-symbols look-ahead*
- Let us implement a parser for LL(1) grammars without recursion but using an auxiliary explicit stack.

FIRST

Let $G = (\Sigma, N, S, P)$ be a grammar and X a non-terminal symbol.

When the next token is x, we may use rule $X \to \gamma$ if

$$x \in \text{FIRST}(\gamma)$$

meaning that x is an *initial symbol* in derivations starting at γ.

To choose between two rule $X \to \gamma$ and $X \to \gamma'$ using only the next symbol we must guarantee that they do not share initial symbols:

$$\text{FIRST}(\gamma) \cap \text{FIRST}(\gamma') = \emptyset$$

FIRST (cont.)

Definition

$$\text{FIRST}(\gamma) = \{x \in \Sigma : \gamma \Rightarrow^* x\beta, \text{ for some } \beta\}$$

This means that FIRST(γ) is the set of tokens at the beginning of words derived by γ.

- This definition is not useful to compute FIRST(γ) for each rule $X \to \gamma$
- Let us see how to compute the set FIRST

FIRST (cont.)

$$\begin{array}{lll} E \to E + T & T \to T * F & F \to \text{num} \\ E \to T & T \to F & F \to (E) \end{array}$$

- Start computing FIRST directly for non-recursive rules e.g.

$$\text{FIRST}(F) = \{\text{num}, (\}$$

- Recursive rules must respect some equations; e.g. for T:

$$\begin{aligned} & \text{FIRST}(T) = \text{FIRST}(T * F) \cup \text{FIRST}(F) \\ \iff & \text{FIRST}(T) = \text{FIRST}(T) \cup \text{FIRST}(F) \end{aligned}$$

- The least solution for the previous equation is:

$$\text{FIRST}(T) = \text{FIRST}(F) = \{\text{num}, (\}$$

Let us see how to get the solution using an iterative method

FIRST (cont.)

- We will also need a predicate NULLABLE(γ) to decide if a sequence may generate the empty word

Equations for FIRST and NULLABLE

- In the previous example we may define the simplification

$$\mathrm{FIRST}(T * F) = \mathrm{FIRST}(T)$$

  because it is not possible to derive the empty word ε from T
- In general: to compute FIRST we need to know which non-terminals may derive ε

$$\mathrm{NULLABLE}(X) = \begin{cases} \mathrm{True} & \text{, if } X \Rightarrow^* \varepsilon \\ \mathrm{False} & \text{, otherwise} \end{cases}$$

- Let us define this predicate by a set of equations

Equations for FIRST and NULLABLE (cont.)

$$
\begin{aligned}
\text{FIRST}(\varepsilon) &= \emptyset \\
\text{FIRST}(a) &= \{a\} \qquad (a \in \Sigma) \\
\text{FIRST}(\alpha\beta) &= \begin{cases} \text{FIRST}(\alpha) \cup \text{FIRST}(\beta)\,, & \text{if NULLABLE}(\alpha) \\ \text{FIRST}(\alpha)\,, & \text{otherwise} \end{cases} \\
\text{FIRST}(X) &= \text{FIRST}(\gamma_1) \cup \ldots \cup \text{FIRST}(\gamma_n)\,, \\
&\qquad \text{where } X \to \gamma_i \text{ are all the rules for } X
\end{aligned}
$$

$$
\begin{aligned}
\text{NULLABLE}(\varepsilon) &= \text{True} \\
\text{NULLABLE}(a) &= \text{False} \qquad (a \in \Sigma) \\
\text{NULLABLE}(\alpha\beta) &= \text{NULLABLE}(\alpha) \wedge \text{NULLABLE}(\beta) \\
\text{NULLABLE}(X) &= \text{NULLABLE}(\gamma_1) \vee \ldots \vee \text{NULLABLE}(\gamma_n) \\
&\qquad \text{where } X \to \gamma_i \text{ are all the rules for } X
\end{aligned}
$$

Algorithm for computing FIRST and NULLABLE

Iterative algorithm:

1. Inicially NULLABLE(X) := False and FIRST$(X) := \emptyset$ for every non-terminal symbol
2. Compute new elements for the right-hand side of productions using the previous equations
3. Repeat this util the sets do not change (reach a *fixpoint*)

Algorithm for computing FIRST and NULLABLE

Iterative algorithm:

1. Inicially NULLABLE(X) := False and FIRST(X) := $\emptyset$ for every non-terminal symbol
2. Compute new elements for the right-hand side of productions using the previous equations
3. Repeat this util the sets do not change (reach a *fixpoint*)

- We may compute NULLABLE and then FIRST
- The algorithm terminates because the previous equations define a *monotonic* function in a finite complete partial order (remember partial orders from Discrete Mathematics...)

Example: arithmetic expressions

$$E \to E + T \qquad T \to T * F \qquad F \to \text{num}$$
$$E \to T \qquad T \to F \qquad F \to (E)$$

$$\text{NULLABLE}(E) = (\text{NULLABLE}(E) \wedge \text{NULLABLE}(+) \wedge \text{NULLABLE}(T)) \vee \text{NULLABLE}(T)$$
$$\text{NULLABLE}(T) = (\text{NULLABLE}(T) \wedge \text{NULLABLE}(*) \wedge \text{NULLABLE}(F)) \vee \text{NULLABLE}(F)$$
$$\text{NULLABLE}(F) = \text{NULLABLE}(\text{num}) \vee \text{NULLABLE}((E)) = \text{False}$$

	iterations	
non-terminals	0	1
NULLABLE(E)	False	False
NULLABLE(T)	False	False
NULLABLE(F)	False	False

Reaches a fixpoint in iteration 1.

Example: arithmetic expressions (cont.)

$$E \to E + T \qquad T \to T * F \qquad F \to \text{num}$$
$$E \to T \qquad T \to F \qquad F \to (E)$$

$$\text{FIRST}(E) = \text{FIRST}(E + T) \cup \text{FIRST}(T) = \text{FIRST}(E) \cup \text{FIRST}(T)$$
$$\text{FIRST}(T) = \text{FIRST}(T * F) \cup \text{FIRST}(F) = \text{FIRST}(T) \cup \text{FIRST}(F)$$
$$\text{FIRST}(F) = \text{FIRST}(\text{num}) \cup \text{FIRST}((E)) = \{\text{num}, (\}$$

	iterations				
non-terminals	0	1	2	3	4
FIRST(E)	$\emptyset$	$\emptyset$	$\emptyset$	$\{\text{num}, (\}$	$\{\text{num}, (\}$
FIRST(T)	$\emptyset$	$\emptyset$	$\{\text{num}, (\}$	$\{\text{num}, (\}$	$\{\text{num}, (\}$
FIRST(F)	$\emptyset$	$\{\text{num}, (\}$	$\{\text{num}, (\}$	$\{\text{num}, (\}$	$\{\text{num}, (\}$

Reaches a fixpoint in iteration 4.

Example: arithmetic expressions (cont.)

Solutions:

$$\begin{aligned} \mathrm{FIRST}(E) &= \{\mathrm{num}, (\} \\ \mathrm{FIRST}(T) &= \{\mathrm{num}, (\} \\ \mathrm{FIRST}(F) &= \{\mathrm{num}, (\} \end{aligned}$$

Thus this grammar is not $LL(1)$ because the FIRST sets of the right-hand side of rules

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \end{aligned}$$

are not disjoint — in fact they are both equal to $\{\mathrm{num}, (\}$.

Exercise

Write the equations and compute NULLABLE and FIRST for the following grammar:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aAb \mid \varepsilon \\
B &\rightarrow bB \mid \varepsilon
\end{aligned}
$$

Exercise

Write the equations and compute NULLABLE and FIRST for the following grammar:

$$\begin{aligned} S &\to AB \\ A &\to aAb \mid \varepsilon \\ B &\to bB \mid \varepsilon \end{aligned}$$

Solutions:

$$\text{NULLABLE}(S) = \text{NULLABLE}(A) = \text{NULLABLE}(B) = \mathit{True}$$

$$\begin{aligned} \text{FIRST}(S) &= \{a, b\} \\ \text{FIRST}(A) &= \{a\} \\ \text{FIRST}(B) &= \{b\} \end{aligned}$$

FOLLOW

- The set FIRST is not enough to characterize $LL(1)$ grammars
- For rules $X \to \gamma$ where NULLABLE(γ) we need to know the tokes which may occur *after* X (FIRST(γ) is not enough to know this)
- Set FOLLOW(X): tokens that occur after X in a derivation beginning in S

$$\text{FOLLOW}(X) = \{c \in \Sigma \;:\; \text{existem } \alpha, \beta \text{ tais que } S \Rightarrow^* \alpha X c \beta\}$$

(FOLLOW) Equations:

Add a new non-terminal symbol \$ and a new rule to capture the end of the input list of token:

$$S' \to S\$$$

For each non terminal symbol X, for each rule: $Y \to \alpha X \beta$:

- FOLLOW(X) $\supseteq$ FIRST(β)
- Se NULLABLE(β) then FOLLOW(X) $\supseteq$ FOLLOW(Y)

How to compute the set FOLLOW

$$\begin{aligned} S' &\rightarrow S\$ \\ S &\rightarrow AB \\ A &\rightarrow aAb \mid \varepsilon \\ B &\rightarrow bB \mid \varepsilon \end{aligned}$$

$S' \rightarrow S\$$	$\text{FOLLOW}(S) \supseteq \{\$\}$	$\text{FIRST}(\$) = \{\$\}$
$S \rightarrow AB$	$\text{FOLLOW}(A) \supseteq \{b\}$	$\text{FIRST}(B) = \{b\}$
	$\text{FOLLOW}(A) \supseteq \text{FOLLOW}(S)$	because $\text{NULLABLE}(B)$
	$\text{FOLLOW}(B) \supseteq \text{FOLLOW}(S)$	
$A \rightarrow aAb$	$\text{FOLLOW}(A) \supseteq \{b\}$	$\text{FIRST}(b) = \{b\}$
$B \rightarrow bB$	$\text{FOLLOW}(B) \supseteq \text{FOLLOW}(B)$	
	($A \rightarrow \varepsilon$ e $B \rightarrow \varepsilon$ are useless)	

How to compute the set FOLLOW (cont.)

We get the following equations:

$$
\begin{aligned}
\mathrm{FOLLOW}(S) &\supseteq \{\$\} \\
\mathrm{FOLLOW}(A) &\supseteq \{b\} \\
\mathrm{FOLLOW}(A) &\supseteq \mathrm{FOLLOW}(S) \\
\mathrm{FOLLOW}(B) &\supseteq \mathrm{FOLLOW}(S) \\
\mathrm{FOLLOW}(B) &\supseteq \mathrm{FOLLOW}(B)
\end{aligned}
$$

Solve the equations iteratively, beginning with $\emptyset$ for every token.

	iterations			
non-terminals	0	1	2	3
FOLLOW(S)	$\emptyset$	$\{\$\}$	$\{\$\}$	$\{\$\}$
FOLLOW(A)	$\emptyset$	$\{b\}$	$\{b, \$\}$	$\{b, \$\}$
FOLLOW(B)	$\emptyset$	$\emptyset$	$\{\$\}$	$\{\$\}$

Parsing table

Use NULLABLE, FIRST and FOLLOW to build the *parsing table*:

- columns: *tokens*
- lines: *non-terminal symbols*
- write $X \to \gamma$:
  - on line X column t for each $t \in \text{FIRST}(\gamma)$;
  - if NULLABLE(γ), on line X column t for each $t \in \text{FOLLOW}(X)$.

The grammar is $LL(1)$ if and only if each entry in the table has at most one rule. (this guarantees that we can choose which rule to use during parsing looking only to the next token).

Parsing table (cont.)

Example:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Parsing table (check NULLABLE, FIRST, FOLLOW in the previous slides):

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

Each table entry haz zero or one rules $\Longrightarrow$ the grammar is $LL(1)$.

Table for Predictive *Parsing*

- Recursive descent *Parsing* uses the programming language *exectution stack* (e.g. Java or C)
- We may implement predictive *parsing* without recursion using the parsing table and an explicit stack.

Table for Predictive *Parsing* (cont.)

Parsing algorithm:

```
stack := empty; push (S', stack);
while (stack not empty) do
  if top(stack) is a terminal then
    /* consume input */
    consume(top(stack)); pop(stack);
  else if(table[top(stack),next] is empty) then
     report_error();
  else
    /* use a grammar rule */
    symbols := right_hand_side(table[top(stack),next]);
    pop(stack);
    pushList(symbols, stack);
```

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\rightarrow S\$ \\
S &\rightarrow AB \\
A &\rightarrow aAb \mid \varepsilon \\
B &\rightarrow bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow aAb$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow bB$	$B \rightarrow \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \rightarrow S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$\begin{aligned} S' &\rightarrow S\$ \\ S &\rightarrow AB \\ A &\rightarrow aAb \mid \varepsilon \\ B &\rightarrow bB \mid \varepsilon \end{aligned}$$

Table:

	a	b	$\$$
S'	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow aAb$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow bB$	$B \rightarrow \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \rightarrow S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \rightarrow AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$\begin{aligned} S' &\to S\$ \\ S &\to AB \\ A &\to aAb \mid \varepsilon \\ B &\to bB \mid \varepsilon \end{aligned}$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$\begin{aligned} S' &\to S\$ \\ S &\to AB \\ A &\to aAb \mid \varepsilon \\ B &\to bB \mid \varepsilon \end{aligned}$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\rightarrow S\$ \\
S &\rightarrow AB \\
A &\rightarrow aAb \mid \varepsilon \\
B &\rightarrow bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \rightarrow S\$$	$S' \rightarrow S\$$	$S' \rightarrow S\$$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow aAb$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow bB$	$B \rightarrow \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \rightarrow S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \rightarrow AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \rightarrow aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \rightarrow aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	consume b
$\underline{B}\$$	$\underline{b}\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	consume b
$\underline{B}\$$	$\underline{b}\$$	$B \to bB$
$\underline{b}B\$$	$\underline{b}\$$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	consume b
$\underline{B}\$$	$\underline{b}\$$	$B \to bB$
$\underline{b}B\$$	$\underline{b}\$$	consume b
$\underline{B}\$$	$\underline{\$}$	

Example: *parsing* of *aabbb*

Grammar:

$$\begin{aligned} S' &\to S\$ \\ S &\to AB \\ A &\to aAb \mid \varepsilon \\ B &\to bB \mid \varepsilon \end{aligned}$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	consume b
$\underline{B}\$$	$\underline{b}\$$	$B \to bB$
$\underline{b}B\$$	$\underline{b}\$$	consume b
$\underline{B}\$$	$\underline{\$}$	$B \to \varepsilon$
$\underline{\$}$	$\underline{\$}$	

Example: *parsing* of *aabbb*

Grammar:

$$
\begin{aligned}
S' &\to S\$ \\
S &\to AB \\
A &\to aAb \mid \varepsilon \\
B &\to bB \mid \varepsilon
\end{aligned}
$$

Table:

	a	b	$\$$
S'	$S' \to S\$$	$S' \to S\$$	$S' \to S\$$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to aAb$	$A \to \varepsilon$	$A \to \varepsilon$
B		$B \to bB$	$B \to \varepsilon$

stack	**input**	**action**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	$A \to aAb$
$\underline{a}AbB\$$	$\underline{a}abbb\$$	consume a
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	consume a
$\underline{A}bbB\$$	$\underline{b}bb\$$	$A \to \varepsilon$
$\underline{b}bB\$$	$\underline{b}bb\$$	consume b
$\underline{b}B\$$	$\underline{b}b\$$	consume b
$\underline{B}\$$	$\underline{b}\$$	$B \to bB$
$\underline{b}B\$$	$\underline{b}\$$	consume b
$\underline{B}\$$	$\underline{\$}$	$B \to \varepsilon$
$\underline{\$}$	$\underline{\$}$	consume $\$$
ε	ε	**accept**

Conclusions

Parsing *top-down*:

- Recursive descent *parsing*
- Recursive functions in Java or C
- $LL(1)$ is a widely used class of grammars
- Define a *parsing table*
- *Parsing* using the parsing table and an auxiliary stack

Parser generators

- We have studied *parsers* which *recognize* a language (yes or no output)
- The next step is to use parsing to also build a syntactic tree

- It is possible to use tools which automatically build *top-down* parsers:

  JavaCC for Java: `https://javacc.github.io/javacc/`

  Parsec for Haskell: `http://hackage.haskell.org/package/parsec`