Compilers
Context Free Grammars

LEIC

FEUP-FCUP

2022

Compilers - Important dates

- Check Point: April 5 and April 6
- Written Test 1 (25%): April 20
- Deadline for Project 1st part (12,5%): May 8
- Project presentations (1st part): May 10 and May 11
- Check Point: May 31 and June 1
- Project Deadline (30%): June 12
- Project presentations (7,5%): June 14 and June 15
- Written Test 2 (25%): June 21

Compiler steps

Input program

↓

Lexical analysis

↓

list of *tokens*

↓

Parsing

↓

AST

↓

Semantic Analysis

↓

AST & Symbol Table

Code generation

↓

intermediate code

Instruction selection

↓

symbolic *assembly*

↓

Register allocation

↓

real *assembly*

↓

Assembler & linker

↓

executable code

This lecture

Syntactic Analysis

Context free grammars

Examples

Parsing

Recursive descent parsing

LL Parsing

Syntactic analysis

- Check that a program syntax is correct with respect to a given grammar e.g.:
  - open and closed brackets { }, ( ) match
  - operators +, $*$, etc. respect their arity;
  - instructions end correctly ;)
- Note that a sentence may have a correct syntax and still does not make sense
  Example (Chomsky, 1957): "*Colorless green ideas sleep furiosly*"
- The (*parser*) builds an abstract syntax tree (AST) from a list of *tokens* (or outputs a syntax error)
- Main framewok: context free grammars

This lecture

Syntactic Analysis

Context free grammars

Examples

Parsing

Recursive descent parsing

LL Parsing

Context free grammars

A context free grammar $G = (\Sigma, N, S, P)$ is defined by:

Σ set of *terminal* symbols;

N set of *non-terminal* symbols;

$S \in N$ initial symbol;

P set of de *production rules* $X \rightarrow \alpha$ where:

- X is non-terminal;
- α is a sequence (maybe empty) of terminal or non-terminal symbols

Example

Terminal symbols:

$$\Sigma = \{a, b\}$$

Non-terminal symbols:

$$N = \{S, B\}$$

Initial symbol:

$$S$$

Production rules:

$$\begin{aligned} S &\rightarrow aSB \\ S &\rightarrow \varepsilon \\ S &\rightarrow B \\ B &\rightarrow Bb \\ B &\rightarrow b \end{aligned}$$

Derivations

A derivation relation $\Rightarrow$ replaces a non-terminal symbol by the right-hand side of its corresponding rule.

Example:

$$
\begin{align}
S &\rightarrow aSB \tag{1}\\
S &\rightarrow \varepsilon \tag{2}\\
S &\rightarrow B \tag{3}\\
B &\rightarrow Bb \tag{4}\\
B &\rightarrow b \tag{5}
\end{align}
$$

$$S \stackrel{1}{\Rightarrow} aSB \stackrel{1}{\Rightarrow} aaSBB \stackrel{2}{\Rightarrow} aaBB \stackrel{4}{\Rightarrow} aaBbB \stackrel{5}{\Rightarrow} aabbB \stackrel{5}{\Rightarrow} aabbb$$

Language defined by a grammar

- Beginning with the initial symbol...
- expand the non-terminals using the corresponding production rules...
- when there only terminal symbols: we reach a *word* described by the grammar.

For the previous grammar G:

$$S \Rightarrow aSB \Rightarrow aaSBB \Rightarrow aaBB \Rightarrow aaBbB \Rightarrow aabbB \Rightarrow aabbb$$

Thus: $aabb \in L(G)$.

Language defined by a grammar (cont.)

Thus, if $G = (\Sigma, N, S, P)$ then

$$L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$$

($\Rightarrow^*$ is the *transitive closure* of the derivation.)

Exercise

$$
\begin{aligned}
G: \quad S &\to aSB \\
S &\to \varepsilon \\
S &\to B \\
B &\to Bb \\
B &\to b
\end{aligned}
$$

Describe the language produced by G.

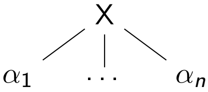
- Where may we have occurrences of a and b?
- What is the relation between the *number* of as and bs?

Syntax trees

Each production rule

$$X \rightarrow \alpha_1 \ldots \alpha_n$$

corresponds to a node with *n* sub-trees:

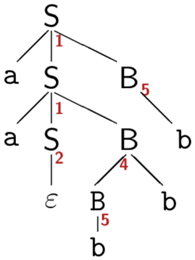


Syntax trees (cont.)

Example:

$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{2}{\Rightarrow} aaBB \overset{4}{\Rightarrow} aaBbB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb$$

corresponds to the tree:



$$
\begin{array}{llll}
G: & S & \rightarrow aSB & (1) \\
 & S & \rightarrow \varepsilon & (2) \\
 & S & \rightarrow B & (3) \\
 & B & \rightarrow Bb & (4) \\
 & B & \rightarrow b & (5)
\end{array}
$$

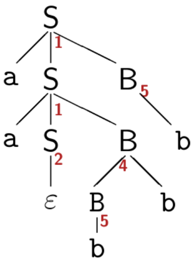
Syntax trees (cont.)

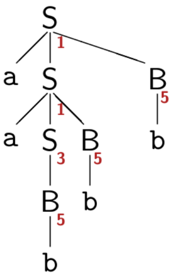
$S \Rightarrow^* aabbb$ may have two different derivations:

$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{2}{\Rightarrow} aaBB \overset{4}{\Rightarrow} aaBbB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb \qquad (6)$$

$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{3}{\Rightarrow} aaBBB \overset{5}{\Rightarrow} aabBB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb \qquad (7)$$

(6) corresponds to the tree on the left-hand side; (7) corresponds to the tree on the right-hand side.





$$G: \begin{array}{lll} S & \rightarrow aSB & (1) \\ S & \rightarrow \varepsilon & (2) \\ S & \rightarrow B & (3) \\ B & \rightarrow Bb & (4) \\ B & \rightarrow b & (5) \end{array}$$

Ambiguous grammars

A grammar is ambiguous if it produces words with different syntax trees.

G is ambiguous

$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{2}{\Rightarrow} aaBB \overset{4}{\Rightarrow} aaBbB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb$$

$$S \overset{1}{\Rightarrow} aSB \overset{1}{\Rightarrow} aaSBB \overset{3}{\Rightarrow} aaBBB \overset{5}{\Rightarrow} aabBB \overset{5}{\Rightarrow} aabbB \overset{5}{\Rightarrow} aabbb$$

because the two previous derivations correspond to two different syntax trees.

Ambiguous grammars

A grammar is ambiguous if it produces words with different syntax trees.

G is ambiguous

$$S \stackrel{1}{\Rightarrow} aSB \stackrel{1}{\Rightarrow} aaSBB \stackrel{2}{\Rightarrow} aaBB \stackrel{4}{\Rightarrow} aaBbB \stackrel{5}{\Rightarrow} aabbB \stackrel{5}{\Rightarrow} aabbb$$

$$S \stackrel{1}{\Rightarrow} aSB \stackrel{1}{\Rightarrow} aaSBB \stackrel{3}{\Rightarrow} aaBBB \stackrel{5}{\Rightarrow} aabBB \stackrel{5}{\Rightarrow} aabbB \stackrel{5}{\Rightarrow} aabbb$$

because the two previous derivations correspond to two different syntax trees.

Note:

- different derivations may correspond to the same syntax tree
- an ambiguous grammar must produce *different syntax tree* and not only *different derivations*

This lecture

Arithmetic expressions

Non-terminals: E

Terminals (tokens): num $\quad + \quad * \quad (\quad)$

Production rules:

$$
\begin{aligned}
E &\to E{+}E \\
E &\to E{*}E \\
E &\to \text{num} \\
E &\to (E)
\end{aligned}
$$

Or...:

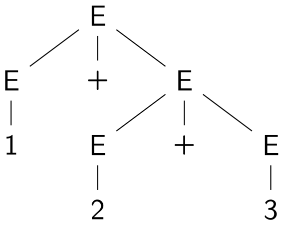
$$E \to E{+}E \;\mid\; E{*}E \;\mid\; \text{num} \;\mid\; (E)$$

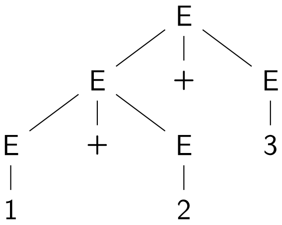
Arithmetic expressions (cont.)

- This grammar is *ambiguous*

Arithmetic expressions (cont.)

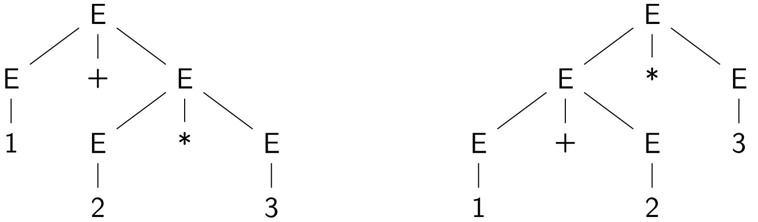
1+2+3:





Arithmetic expressions (cont.)

1+2∗3:



How to eliminate ambiguity

For the previous example we must define:

- associativity properties

  left: 1+2+3 means $(1+2)+3$

  right: 1+2+3 means $1+(2+3)$

- a priority between + and $*$

  e.g. 1+2*3 means $1+(2\times 3)$ or $(1+2)\times 3$

How to eliminate ambiguity (cont.)

$$E \rightarrow E + T \qquad T \rightarrow T * F \qquad F \rightarrow \mathrm{num}$$
$$E \rightarrow T \qquad T \rightarrow F \qquad F \rightarrow (E)$$

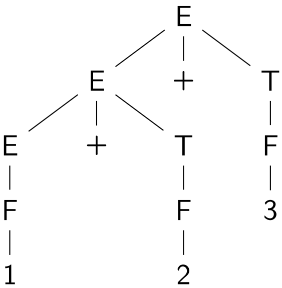
- In this grammar:

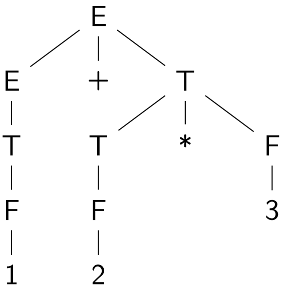
  expressions E are sums of *terms*;
  terms T are products of *factors*;
  factors F are constants or expressions between brackets.

- Productions of E and T with left recursion mean left associativity of + and $*$

How to eliminate ambiguity (cont.)

Now 1+2+3 and 1+2*3 have unique syntaxt trees:





Example: sequences of statements

Non-terminals: S (*statements*) E (*expressions*)

Terminals (tokens): ident num = ( ) + , ; ++

Production rules:

$$\begin{array}{lll} S \to S \; ; \; S & \qquad & E \to \text{ident} \\ S \to \text{ident} = E & & E \to \text{num} \\ S \to \text{ident} \; {++} & & E \to E + E \end{array}$$

Example:

```
a = 17; b = 2
a = 0; (a++; b=a+5)
```

Example: sequences of statements (cont.)

Exercises:

1. Show that the previous grammar is ambiguous.
2. Rewrite the grammar to eliminate ambiguity.
   (Note: the problem is not only with expressions!)

“Dangling else”

if/then with optional *else*:

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{if } E \text{ then } S$$
$$S \rightarrow \textit{etc.}$$

Then

$$\text{if } e_1 \text{ then if } e_2 \text{ then } s_1 \text{ else } s_2$$

may have the two meanings:

$$\text{if } e_1 \text{ then } \{\text{if } e_2 \text{ then } s_1 \text{ else } s_2\} \tag{8}$$
$$\text{if } e_1 \text{ then } \{\text{if } e_2 \text{ then } s_1\} \text{ else } s_2 \tag{9}$$

Usually programming languages use (8): associate the *else* to the nearest *if*.

“Dangling else” (cont.)

Two new non-terminals: M (*matched statements*) and U (*unmatched statements*).

$$
\begin{aligned}
&S \rightarrow M \\
&S \rightarrow U \\[4pt]
&M \rightarrow \text{if } E \text{ then } M \text{ else } M \\
&M \rightarrow \textit{etc.} \\[4pt]
&U \rightarrow \text{if } E \text{ then } S \\
&U \rightarrow \text{if } E \text{ then } M \text{ else } U
\end{aligned}
$$

In practice: it may be better to solve these ambiguities in the parser implementation...

Left Factoring

$$A \to \alpha\beta \mid \alpha\gamma$$

Example: dangling else

An LL(1) parser cannot distinguish between the production choices.

Solution: rewrite the rule as:

$$A \to \alpha A'$$

$$A' \to \beta \mid \gamma$$

This lecture

Syntactic Analysis

Context free grammars

Examples

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 begin 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

- Syntactic Analysis
- Context free grammars
- Examples
- 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:

$$S \to \text{if } E \text{ then } S \text{ else } S \qquad L \to \text{end}$$
$$S \to \text{begin } S\ L \qquad L \to \ ;\ S\ L$$
$$S \to \text{print } E \qquad E \to \text{num} = \text{num}$$

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

- Syntactic Analysis
- Context free grammars
- Examples
- 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

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

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 ?

Left Recursion Removal

Consider the following grammar:

$$E \to E \;+\; T$$
$$E \to T$$

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':

$$E \to T \; E'$$

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

This grammar is right recursive.

Left Recursion Removal (cont.)

$$E \to T\ E'$$

$$E' \to +\ T\ E'$$
$$E' \to \varepsilon$$

- 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 $E' \to +\ T\ E'$
  otherwise use $E' \to \varepsilon$

Left Recursion Removal (cont.)

Applying the same transformation to the initial grammar we get:

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

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

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

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:

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

FIRST (cont.)

Definition

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

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.)

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

- 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

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

- Let us define this predicate by a set of equations

Equations for FIRST and NULLABLE (cont.)

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

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

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

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

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

	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.)

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

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

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

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

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:

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

Exercise

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

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

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

$$\text{FOLLOW}(X) = \{c \in \Sigma \ : \ \text{there are } \alpha, \beta \text{ such that } S \Rightarrow^* \alpha X c \beta\}$$

(FOLLOW) Equations:

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

$$S' \rightarrow S\$$$

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

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

How to compute the set FOLLOW

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

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

LL(1) Parsing

The rule to choose productions is:

We choose a production rule $N \to \alpha$ on input symbol c if:

1. $c \in FIRST(\alpha)$, or
2. $Nullable(\alpha)$ and $c \in FOLLOW(N)$.

If we can always choose a production uniquely by using these rules, this is called *LL(1) parsing*. A grammar that can be parsed using LL(1) parsing is called an *LL(1) grammar*.

Theorem: A grammar is LL(1) if the FIRST sets corresponding to different rules defining the same non-terminal symbol have an empty intersection and for every nonterminal A such that FIRST(A) contains ϵ, $FIRST(A) \cap FOLLOW(A) = \emptyset$

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 $t \in \text{FOLLOW}(X)$.

A grammar is $LL(1)$ if and only if each entry in the table has at most one rule.

Parsing table (cont.)

Example:

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

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

	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		$B \rightarrow bB$	$B \rightarrow \varepsilon$

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:

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

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**
$\underline{S'}$	$\underline{a}abbb\$$	

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**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	

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**
$\underline{S'}$	$\underline{a}abbb\$$	$S' \to S\$$
$\underline{S}\$$	$\underline{a}abbb\$$	$S \to AB$
$\underline{A}B\$$	$\underline{a}abbb\$$	

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:

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

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
$\underline{A}bB\$$	$\underline{a}bbb\$$	$A \to aAb$
$\underline{a}AbbB\$$	$\underline{a}bbb\$$	

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`