

# Syntactical analysis

- Context-free grammars
- Derivations
- Parse Trees
- Left-recursive grammars
- Top-down parsing
  - non-recursive predictive parsers
  - construction of parse tables
- Bottom-up parsing
  - shift/reduce parsers
  - LR parsers
  - GLR parsers
  - SGLR parsers

# Syntactical analysis

A *context-free grammar* is a 4-tuple  $G = (N, \Sigma, P, S)$

1.  $N$  is a set of non terminals
2.  $\Sigma$  is a set of terminals (disjoint from  $N$ )
3.  $P$  is a subset of  $(N \cup \Sigma)^* \times N$

An element  $(\alpha, A) \in P$  is called a *production*

$A ::= \alpha$  or  $\alpha \rightarrow A$

4.  $S \in N$  is the start symbol

The sets  $N, \Sigma, P$  are finite

# Syntactical analysis

A context-free grammar can be consider as a simple rewrite system:

$$\alpha A \beta \Rightarrow \alpha \gamma \beta \text{ if } \gamma \rightarrow A \in P \ (\alpha, \beta, \gamma \in (N \cup \Sigma)^*, A \in N)$$

**Example**  $N = \{E\}, \Sigma = \{+, *, (, ), -, a\}, S = E,$

$$P = \{ E + E \rightarrow E$$

$$E * E \rightarrow E$$

$$( E ) \rightarrow E$$

$$- E \rightarrow E$$

$$a \rightarrow E \}$$

**Derivation:**  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(a+E) \Rightarrow -(a+a)$

# Syntactical analysis

The *language*  $L(G)$  generated by the context-free grammar  $G = (N, \Sigma, P, S)$  is:

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

A sentence  $w \in L(G)$  contains only terminals

A *sentential form*  $\alpha$  is a string of terminals and non-terminals which can be derived from  $S$ :

$$S \Rightarrow^* \alpha \text{ with } \alpha \in (N \cup \Sigma)^*$$

A sentence in  $L(G)$  is a sentential form in which *no* non-terminals occur

# Syntactical analysis

## Left/right derivations

- There are choices to be made for each derivation step:
  - which non-terminal must be replaced?
  - which alternative of the selected non-terminal must be applied?
- *Always* selecting the leftmost non-terminal in the sentential form gives a *leftmost derivation*:  $\Rightarrow_{lm}$
- There exists also a *rightmost derivation*:  $\Rightarrow_{rm}$

# Syntactical analysis

## Consider the context-free grammar for expressions:

- Leftmost derivation for  $-(a+a)$ 

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(a+E) \Rightarrow -(a+a)$$
- Rightmost derivation for  $-(a+a)$ 

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+a) \Rightarrow -(a+a)$$

# Syntactical analysis

A parse tree for a context-free grammar is a  $G = (N, \Sigma, P, S)$  tree:

1. The root is labeled with  $S$  (the start non-terminal)
2. Each leaf is labeled with a terminal ( $\in \Sigma$ ) or  $\epsilon$
3. All other nodes are labeled with a non-terminal

If  $A$  is the label of a node and  $X_1, \dots, X_n$  are the labels of the children (from left to right) then

$$X_1, \dots, X_n \rightarrow A$$

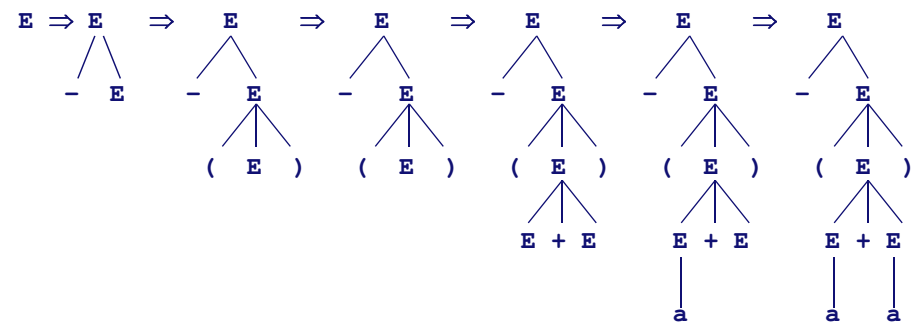
must be a production rule in  $G$  (with  $X_i$  is either a terminal or a non-terminal)

Special case:  $\epsilon \rightarrow A$  with label  $A$  which has exactly one child with label  $\epsilon$

# Syntactical analysis

## Example:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(a+E) \Rightarrow -(a+a)$$



The parse tree abstracts from the derivation order

# Syntactical analysis

## Acceptor and parser

For each grammar  $G$  there exists a decision procedure (acceptor)

$AG$  for  $L(G)$ :

$AG: \text{STRING} \rightarrow \{\text{true}, \text{false}\}$

such that

$AG(w) = \text{true} \Leftrightarrow w \in L(G)$

A *parser* is an acceptor which constructs a parse tree as well.

- A *top-down* parser constructs the tree starting from the root
- A *bottom-up* parser constructs the tree starting from the leaves

# Syntactical analysis

During parsing the following problems may occur:

- The grammar is ambiguous
- The grammar is left recursive
- The grammar contains cycles

A grammar  $G$  is *ambiguous* if one word  $w \in L(G)$  has at least two parse trees

- Expression grammar with associativities and priorities
- Dangling else problem

# Syntactical analysis

- A grammar is immediate *left recursive* if the grammar contains a rule of the form  $\alpha \rightarrow A$
- A grammar is *left recursive* if there exists a non-terminal  $A$  and a string  $\alpha \in (N \cup \Sigma)^*$  such that  $A \Rightarrow A\alpha$
- This means that after one or more steps in a derivation an occurrence of  $A$  reduces again to an occurrence of  $A$  without recognizing any terminal in the input sentence.

# Syntactical analysis

Examples of indirect left recursion

$B \alpha \rightarrow A$

$A \beta \rightarrow B$

or worse

$B \alpha \rightarrow A$

$D A \beta \rightarrow B$

$\varepsilon \rightarrow D$

$\gamma G \rightarrow D$

It is easy to remove left recursion from a context-free grammar

# Syntactical analysis

## Elimination of left recursion

$$A \alpha \rightarrow A$$

$$\beta \rightarrow A$$

produce the sentential forms:  $\beta \alpha^n$

A set of equivalent (non left recursive) rules are

$$\beta A' \rightarrow B$$

$$\alpha A' \rightarrow A'$$

$$\varepsilon \rightarrow A'$$

# Syntactical analysis

## Example:

$$(1) E + T \rightarrow E \text{ (immediate left rec.)}$$

$$(2) T \rightarrow E$$

$$(3) T * F \rightarrow T \text{ (immediate left rec.)}$$

$$(4) F \rightarrow T$$

$$(5) ( E ) \rightarrow F$$

$$(6) a \rightarrow F$$

## Applying the left recursion elimination transformation:

$$(1) E \alpha \rightarrow E \text{ (with } \alpha = + T \text{)}$$

$$(2) \beta \rightarrow E \text{ (with } \beta = T \text{)}$$

# Syntactical analysis

## Example:

$$(1') T E' \rightarrow E$$

$$(2') + T E' \rightarrow E'$$

$$(2'') \varepsilon \rightarrow E'$$

the same for:

$$(3') F T' \rightarrow T$$

$$(4') * F T' \rightarrow T$$

$$(4'') \varepsilon \rightarrow F$$

# Syntactical analysis

## Indirect left recursion elimination

- Suppose we have a rule of the form

$$B \alpha \rightarrow A$$

$$\beta_1 \rightarrow B$$

$$\beta_2 \rightarrow B$$

...

$$\beta_n \rightarrow B$$

- The rule  $B \alpha \rightarrow A$  is now transformed into:

$$\beta_1 \alpha \rightarrow A$$

$$\beta_2 \alpha \rightarrow A$$

...

$$\beta_n \alpha \rightarrow A$$

## Syntactical analysis

This process is repeated until either

- $t\gamma \rightarrow A$ ; the process stops, or
- $A\gamma \rightarrow A$ ; the immediately left recursion elimination rule can be applied

## Syntactical analysis

Left factorization

- In general it is efficient to move the difference between the alternatives of a non-terminal as far as possible to the left
- Productions of the form
$$\alpha\beta_1 \rightarrow A$$
$$\alpha\beta_2 \rightarrow A$$
$$\dots$$
$$\alpha\beta_n \rightarrow A$$
- Are equivalent with
$$\alpha A' \rightarrow A$$
$$\beta_1 \rightarrow A'$$
$$\dots$$
$$\beta_n \rightarrow A'$$

## Syntactical analysis

Example

`if b then S else S`  $\rightarrow S$

`if b then S`  $\rightarrow S$

Only at the occurrence of `else` it can be decided which alternative should have been selected

An equivalent grammar is

`if b then S S'`  $\rightarrow S$

`else S`  $\rightarrow S'$

`$\epsilon$`   $\rightarrow S'$

## Syntactical analysis

Top-down parsing

- A top-down parser “guesses” the next alternative to be recognized, and verifies whether this alternative can be recognized in the input. If not, another alternative will be tried
- Constructs the parse tree starting at the root
- Finds the leftmost derivation of the sentence
  
- Alternative types of top-down parsers:
  - recursive descent parser with backtracking
  - recursive descent parser without backtracking (“predictive parser”)
  - non-recursive predictive parser (uses push-down automaton)

# Syntactical analysis

## Recursive descent parser with backtracking

- Grammar
  - $c A d \rightarrow S$
  - $a \rightarrow A$
  - $a b \rightarrow A$
- Parser

```
bool proc S() {
  if input = 'c'
  then inptr += 1;
   if A()
   then if input = 'd'
        then inptr += 1; return(true)
        fi
   fi
fi;
return(false)
}
```

# Syntactical analysis

```
bool proc A() {
  isave := inptr;
  if input = 'a'
  then inptr += 1;
   if input = 'd'
   then inptr += 1; return(true)
   fi
fi;
inptr := isave;
if input = 'a'
then inptr += 1; return(true)
else return(false)
fi
}
```

# Syntactical analysis

## Recursive descent without backtracking

- For "some" context-free grammars a recursive descent grammar without backtracking can be derived

### Grammar:

```
T E'   → E + T E' → E'
        ε       → E'
F T'   → T * F T' → T
        ε       → F
( E )  → F a     → F
```

# Syntactical analysis

## Recursive descent without backtracking

- Parser:

```
proc E() {
  T();
  E'();
}

proc E'() {
  if input = '+'
  then inptr += 1; T(); E'();
fi
}

proc T() {
  F();
  T'();
}

proc T'() {
  if input = '*'
  then inptr += 1; F(); T'();
fi
}
```

# Syntactical analysis

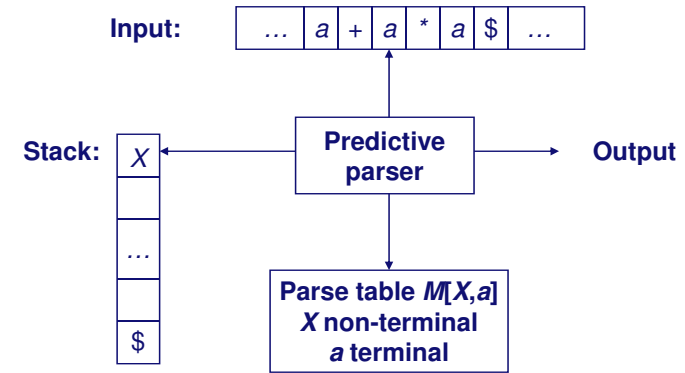
## Recursive descent without backtracking

```

proc F() {
  if input = 'a'
  then inptr += 1
  else if input = '('
    then inptr += 1;
      E();
      if input = ')'
      then inptr += 1
      else ERROR()
    fi
  else ERROR()
  fi
fi
}
    
```

# Syntactical analysis

## Non-recursive predictive parser



Inspects top of stack ( $X$ ) and current input symbol ( $a$ )

# Syntactical analysis

There are three cases:

1.  $X = a = \$$ : successful parse
2.  $X = a \neq \$$ : pop  $X$  and increment input pointer
3.  $X$  is a non-terminal
  - a.  $M[X, a] = \text{error}$
  - b.  $M[X, a] = \{U V W \rightarrow X\}$   
put  $W$ ,  $V$  and  $U$  on the stack ( $U$  on top)

# Syntactical analysis

Grammar:

$$\begin{aligned}
 T E' &\rightarrow E & + T E' &\rightarrow E' \\
 & & \epsilon &\rightarrow E' \\
 F T' &\rightarrow T & * F T' &\rightarrow T \\
 & & \epsilon &\rightarrow F \\
 ( E ) &\rightarrow F & a &\rightarrow F
 \end{aligned}$$

Parse table:

	$a$	$+$	$*$	$($	$)$	$\$$
$E$	$T E'$			$T E'$		
$E'$		$+ T E'$			$\epsilon$	$\epsilon$
$T$	$F T'$			$F T'$		
$T'$		$\epsilon$	$* F T'$		$\epsilon$	$\epsilon$
$F$	$a$			$( E )$		

# Syntactical analysis

Stack	Input	Output
\$E	a + a * a \$	
\$E' T	a + a * a \$	T E' → E
\$E' T' F	a + a * a \$	F T' → T
\$E' T' a	a + a * a \$	a → F
\$E' T'	+ a * a \$	
\$E'	+ a * a \$	ε → T'
\$E' T+	+ a * a \$	+ T E' → E'
\$E' T	a * a \$	
\$E' T' F	a * a \$	F T' → T
\$E' T' a	a * a \$	a → F
\$E' T'	* a \$	
\$E' T' F*	* a \$	* F T' → T'
\$E' T' F	a \$	
\$E' T' a	a \$	a → F
\$E' T'	\$	
\$E'	\$	ε → T'
\$	\$	ε → E'
		accept

# Syntactical analysis

## Construction of parse table

**First( $\alpha$ ):** set of all terminals where strings which are derived from  $\alpha$  can start with;  
**and:** if  $\alpha \Rightarrow^* \varepsilon$  then  $\varepsilon \in \text{First}(\alpha)$

**Follow( $A$ ):** set of all terminals which follow  $A$  immediately in a sentential form