

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 → S
if b then S           → 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' → S
else S         → S'
ε              → 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:

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

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
}
```

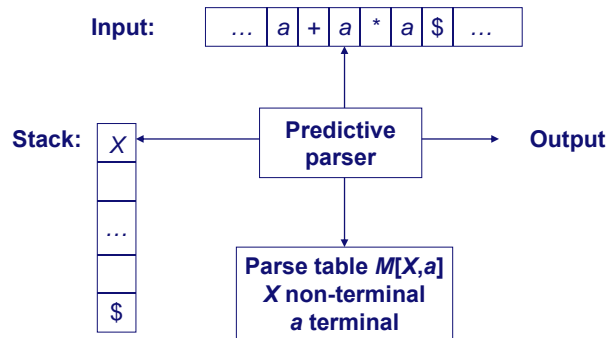
Syntactical analysis

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:

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

Parse table:

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

Syntactical analysis

Stack

$\$E$
 $\$E'T$
 $\$E'T'F$
 $\$E'T'a$
 $\$E'T'$
 $\$E'$
 $\$E'T+$
 $\$E'T$
 $\$E'T'F$
 $\$E'T'a$
 $\$E'T'$
 $\$E'T'F*$
 $\$E'T'F$
 $\$E'T'a$
 $\$E'T'$
 $\$E'$
 $\$$

Input

$a + a * a \$$
 $a + a * a \$$
 $a + a * a \$$
 $a + a * a \$$
 $+ a * a \$$
 $+ a * a \$$
 $+ a * a \$$
 $a * a \$$
 $a * a \$$
 $a * a \$$
 $* a \$$
 $* a \$$
 $a \$$
 $a \$$
 $a \$$
 $\$$
 $\$$

Output

$T E' \rightarrow E$
 $F T' \rightarrow T$
 $a \rightarrow F$
 $\epsilon \rightarrow T'$
 $+ T E' \rightarrow E'$
 $F T' \rightarrow T$
 $a \rightarrow F$
 $* F T' \rightarrow T'$
 $a \rightarrow F$
 $a \rightarrow F$
 $\epsilon \rightarrow T'$
 $\epsilon \rightarrow E'$
 accept

Syntactical analysis

Construction of parse table

First(α): set of all terminals where strings which are derived from α 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

Syntactical analysis

First(X):

1. Initially $\text{First}(X) = \{\}$ for all $X \in N$
2. If X is a terminal: $\text{First}(X) := \{X\}$
3. If $\varepsilon \rightarrow X$ then: $\text{First}(X) := \text{First}(X) \cup \{\varepsilon\}$
4. If $Y_1 \dots Y_{i-1} Y_i \dots Y_k \rightarrow X$, and $\varepsilon \in \text{First}(Y_j)$, $j = 1, \dots, i-1$ then $\text{First}(X) := \text{First}(X) \cup \text{First}(Y_i) \setminus \{\varepsilon\} \cup \dots \cup \text{First}(Y_{i-1}) \setminus \{\varepsilon\} \cup \text{First}(Y_i)$
If $Y_1 \dots Y_{i-1} Y_i \dots Y_k \rightarrow X$, and $\varepsilon \in \text{First}(Y_j)$, $j = 1, \dots, k$ then $\text{First}(X) := \text{First}(X) \cup \{\varepsilon\}$
5. Repeat step 4 as long as new elements are added to any First set

Syntactical analysis

Follow(A):

1. For all $A \neq S$ (where S is the start symbol):
 $\text{Follow}(A) = \{\}$
 $\text{Follow}(S) = \{\$ \}$
2. If there is a production $\alpha B \beta \rightarrow A$ then $\text{Follow}(B) := \text{First}(\beta) \setminus \{\varepsilon\}$
If there is either a production $\alpha B \rightarrow A$, or $\alpha B \beta \rightarrow A$, with $\varepsilon \in \text{First}(\beta)$, then $\text{Follow}(B) := \text{Follow}(B) \cup \text{Follow}(A)$
3. Repeat step 2 as long as new elements are added to any Follow set

Syntactical analysis

Grammar:

$\mathbb{T} \mathbb{E}'$	$\rightarrow \mathbb{E}$	$+$	$\mathbb{T} \mathbb{E}'$	$\rightarrow \mathbb{E}'$
		ε		$\rightarrow \mathbb{E}'$
$\mathbb{F} \mathbb{T}'$	$\rightarrow \mathbb{T}$	$*$	$\mathbb{F} \mathbb{T}'$	$\rightarrow \mathbb{T}$
		ε		$\rightarrow \mathbb{F}$
(\mathbb{E})	$\rightarrow \mathbb{F}$	a		$\rightarrow \mathbb{F}$

$\text{First}(\mathbb{E}) = \text{First}(\mathbb{T}) = \text{First}(\mathbb{F}) = \{ (, a \}$

$\text{First}(\mathbb{E}') = \{ +, \varepsilon \}$

$\text{First}(\mathbb{T}') = \{ *, \varepsilon \}$

$\text{Follow}(\mathbb{E}) = \text{Follow}(\mathbb{E}') = \{), \$ \}$

$\text{Follow}(\mathbb{T}) = \text{Follow}(\mathbb{T}') = \{ +,), \$ \}$

$\text{Follow}(\mathbb{F}) = \{ *, +,), \$ \}$

Syntactical analysis

Construction of parse table for top-down parser

Input: context-free grammar G

Output: parse table T for G

1. For all non-terminals A and terminals a : $T[A,a] := \{\}$
2. For every rule $\alpha \rightarrow A$ in G :
 - a. For every terminal $a \in \text{First}(\alpha)$: $T[A,a] := T[A,a] \cup \{\alpha \rightarrow A\}$
 - b. If $\epsilon \in \text{First}(\alpha)$, then:
 - i. For all $b \in \text{Follow}(A)$: $T[A,b] := T[A,b] \cup \{\alpha \rightarrow A\}$
 - ii. If $\$ \in \text{Follow}(A)$, then: $T[A,\$] := T[A,\$] \cup \{\alpha \rightarrow A\}$
3. Give all empty entries of T the value `error`.

Syntactical analysis

LL(1) condition:

A grammar with a parse table which does not contain multiple entries can be parsed predicatively. For each input symbol there is a unique choice.

The property is the LL(1)-property:

LL(1) = Left-to-right-parsing,
Left-most derivation,
1 symbol look ahead

Syntactical analysis

Alternative definition of LL(1):

Grammar G is LL(1) if and only if for each pair of productions $\alpha \rightarrow A$ and $\beta \rightarrow A$ holds:

1. $\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$ (meaning α and β can not produce strings starting with the same terminal)
2. Either α or β may produce empty
3. If $\epsilon \in \text{First}(\beta)$, then holds: $\text{First}(\alpha) \cap \text{Follow}(A) = \emptyset$ (meaning if β produces the empty string, then α may not produce strings beginning with a terminal that follows A)