Languages and Compilers
(SProg og Oversættere)

Lecture 5
Context Free Grammars

Bent Thomsen

Department of Computer Science

Aalborg University

Programming Language Specification

- A Language specification has (at least) three parts
  - Syntax of the language:
    - **usually formal CFG in BNF or EBNF**
    - Tokens defined using regular expressions (RE)
  - Contextual constraints:
    - scope rules (often written in English, but can be formal)
    - type rules (formal or informal)
  - Semantics:
    - defined by the implementation
    - informal descriptions in English
    - formal using operational or denotational semantics

Syntax Specification

Syntax is specified using “Context Free Grammars”:

- A finite set of **terminal symbols**
- A finite set of **non-terminal symbols**
- A **start symbol**
- A finite set of **production rules**

A CFG defines a set of strings

- This is called the language of the CFG.

How to design a grammar?

- Let's write a CFG for C-style function prototypes!
- Write examples:
  - void myf1(int x, double y);
  - int myf2();
  - int myf3(double z);
  - double myf4(int, int w, int);
  - void myf5(void);

- Terminals: void, int, double, (,), , , ; , ident
  - ident = [a-z]([a-z]|[0-9])*

Designing a grammar for Function Prototypes

- Here is one possible grammar

S → Ret ident (Args);

Ret → Type | void

Type → int | double

Args → ε | void | ArgList

ArgList → OneArg | ArgList, OneArg

OneArg → Type | Type ident

- Examples
  - void ident(int ident, double ident);
  - int ident();
  - int ident(double ident);
  - double ident(int, int ident, int);
  - void ident(void);

Designing a grammar for Function Prototypes

- Here is another possible grammar

S → Ret ident Args ;

Ret → int | double | void

Type → int | double

Args → () | (void)| (ArgList)

ArgList → OneArg |OneArg,ArgListArg

OneArg → Type | Type ident

- Examples
  - void ident(int ident, double ident);
  - int ident();
  - int ident(double ident);
  - double ident(int, int ident, int);
  - void ident(void);

Context-Free Grammars

- Components: $G=(N,\Sigma,P,S)$
  - A finite **terminal alphabet** Σ: the set of tokens produced by the scanner
  - A finite **nonterminal alphabet** N: variables of the grammar
  - A **start symbol** S: $S \in N$ that initiates all derivations
    - *Goal symbol*
  - A finite set of **productions** P: $A \rightarrow X_1 \ldots X_m$, where $A \in N$, $X_i \in N \cup \Sigma$, $1 \le i \le m$ and $m \ge 0$.
    - *Rewriting rules*
- Vocabulary $V=N \cup \Sigma$
  - $N \cap \Sigma = \phi$

- CFG: recipe for creating strings
- *Derivation*: a rewriting step using the production A→α replaces the nonterminal A with the vocabulary symbols in α
  - Left-hand side (LHS): A
  - Right-hand side (RHS): α
- *Context-free language* of grammar G *L(G)*: the set of terminal strings derivable from S

- notation:
  - $A \rightarrow \alpha$
    $\mid \beta$
    $\ldots$
    $\mid \zeta$
- or
  - $A \rightarrow \alpha$
    $A \rightarrow \beta$
    $\ldots$
    $A \rightarrow \zeta$

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$: one step of *derivation* using the production $A \rightarrow \gamma$
  - $\Rightarrow^+$: derives in one or more steps
  - $\Rightarrow^*$: derives in zero or more steps
- $S \Rightarrow^* \beta$: β is a sentential form of the CFG
- SF(G): the set of sentential forms of G
- $L(G)=\{w \in \Sigma^* \mid S \Rightarrow^+ w\}$
  - $L(G)=SF(G) \cap \Sigma^*$

Two conventions that nonterminals are rewritten in some systematic order

Leftmost derivation: from left to right

Rightmost derivation: from right to left

Leftmost Derivation

- A derivation that always chooses the leftmost possible nonterminal at each step
  - $=>_{lm}$, $=>^{+}_{lm}$, $=>^{*}_{lm}$
  - A left sentential form
    - A sentential form produced via a leftmost derivation
    - E.g. production sequence in top-down parsers
    - (Fig. 4.1)

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Figure 4.1: A simple expression grammar.

- E.g: a leftmost derivation of f (v + v)
  - $E \Rightarrow_{lm}$ Prefix (E)
    $\Rightarrow_{lm}$ f (E)
    $\Rightarrow_{lm}$ f (v Tail)
    $\Rightarrow_{lm}$ f (v + E)
    $\Rightarrow_{lm}$ f (v + v Tail)
    $\Rightarrow_{lm}$ f (v + v)

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Rightmost Derivations

- The rightmost possible nonterminal is always expanded
  - $\Rightarrow_{rm}$, $\Rightarrow^{+}_{rm}$, $\Rightarrow^{*}_{rm}$
  - A right sentential form
    - A sentential form produced via a rightmost derivation
    - E.g. produced by bottom-up parsers (Ch. 6)
    - (Fig. 4.1)

- E.g: a rightmost derivation of f (v + v)
  - $E \Rightarrow_{rm} \text{Prefix} (E)$
    $\Rightarrow_{rm} \text{Prefix} (v \text{ Tail})$
    $\Rightarrow_{rm} \text{Prefix} (v + E)$
    $\Rightarrow_{rm} \text{Prefix} (v + v \text{ Tail})$
    $\Rightarrow_{rm} \text{Prefix} (v + v)$
    $\Rightarrow_{rm} f (v + v)$

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

Parse Trees

- Parse tree: graphical representation of a derivation
  - Root: start symbol S
  - Each node: either grammar symbol or λ (or ε)
  - Interior nodes: nonterminals
    - An interior node and its children: production
  - E.g. Fig. 4.2

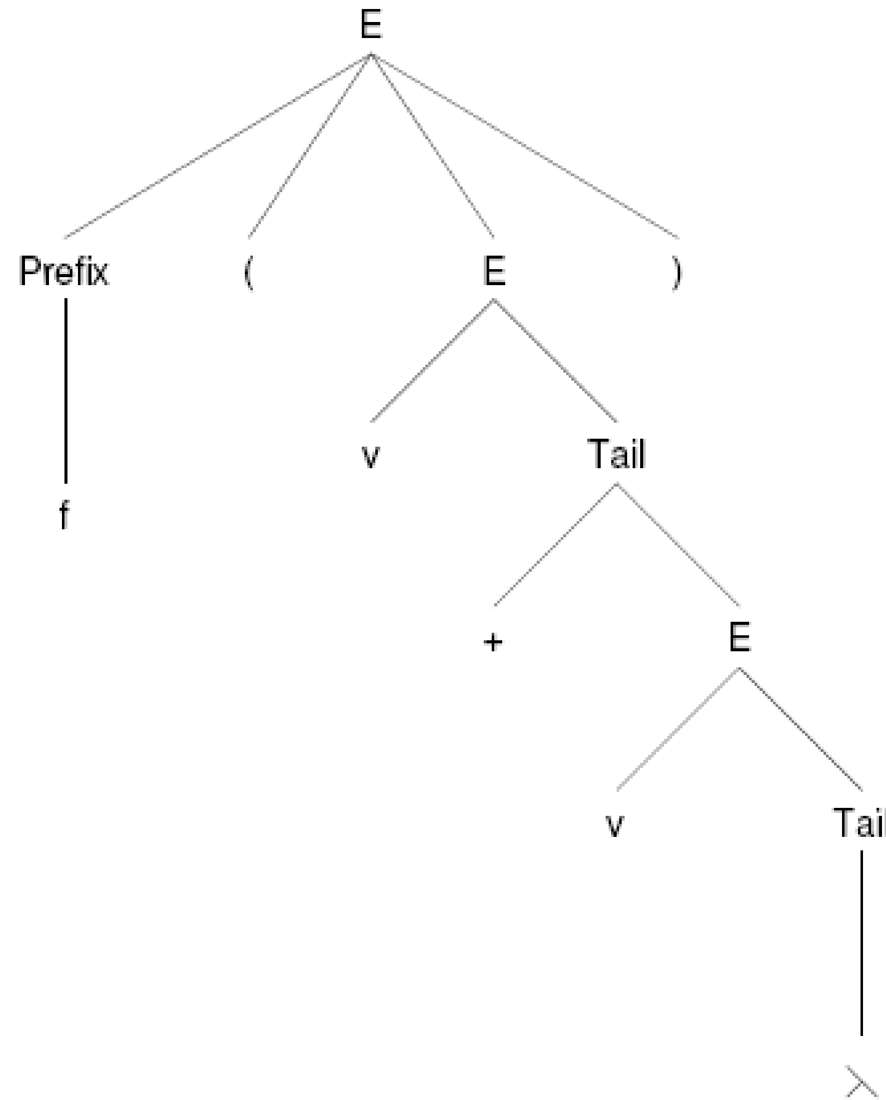


Figure 4.2: The parse tree for f (v + v) .

BNF form of grammars

- Backus-Naur Form (BNF) is a formal grammar for expressing context-free grammars.
- The single grammar rule format:
  - Non-terminal → zero or more grammar symbols

- It is usual to combine all rules with the same left-hand side into one rule, such as:

  $N \rightarrow \alpha$

  $N \rightarrow \beta$

  $N \rightarrow \gamma$

  Greek letters α,β, or γ means a string of symbols.

  are combined into one rule:

  $N \rightarrow \alpha \mid \beta \mid \gamma$

  α, β and γ are called the ***alternatives*** of N.

Extended BNF form of grammars

- BNF is very suitable for expressing nesting and recursion, but less convenient for repetition and optionality.

- Three additional postfix operators +,?, and *, are thus introduced:
  - R+ indicates the occurrence of one or more Rs, to express repetition (sometime R_opt isused).
  - R? indicates the occurrence of zero or one Rs, to express optionality (sometimes [R] is used).
  - R* indicates the occurrence of zero or more Rs, to express repetition (sometimes {R} is used).
- The grammar that allows the above is called Extended BNF (EBNF).

Extended forms of grammars

An example is the grammar rule in EBNF:

parameter_list →

('IN' | 'OUT')? identifier (',' identifier)*

or

parameter_list →

['IN' | 'OUT'] identifier {',' identifier}

which produces program fragments like:

a, b

IN year, month, day

OUT left, right

Extended forms of grammars

- Rewrite EBNF grammar to CFG
  - Given the EBNF grammar:

    expression → term (+ term)*

  Rewrite it to:

    expression → term term_tmp

    term_tmp → + term term_tmp

    | λ

foreach $p \in Prods$ of the form "$\mathsf{A} \rightarrow \alpha \; [\; \mathcal{X}_1 \ldots \mathcal{X}_n \;] \; \beta$" **do**

$N \leftarrow$ NewNonTerm( )

$p \leftarrow$ "$\mathsf{A} \rightarrow \alpha \; N \; \beta$"

$Prods \leftarrow Prods \cup \{$"$N \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n$"$\}$

$Prods \leftarrow Prods \cup \{$"$N \rightarrow \lambda$"$\}$

foreach $p \in Prods$ of the form "$\mathsf{B} \rightarrow \gamma \; \{\; \mathcal{X}_1 \ldots \mathcal{X}_m \;\} \; \delta$" **do**

$M \leftarrow$ NewNonTerm( )

$p \leftarrow$ "$\mathsf{B} \rightarrow \gamma \; M \; \delta$"

$Prods \leftarrow Prods \cup \{$"$M \rightarrow \mathcal{X}_1 \ldots \mathcal{X}_n \; M$"$\}$

$Prods \leftarrow Prods \cup \{$"$M \rightarrow \lambda$"$\}$

Figure 4.4: Algorithm to transform a BNF grammar into standard form.

Properties of grammars

- A non-terminal N is **left-recursive** if, starting with a sentential form N, we can produce another sentential form starting with N.
  – ex: expression → expression ‘+’ factor | factor

- right-recursion also exists, but is less important.
  – ex: expression → term ‘+’ expression

Properties of grammars (Cont.)

- A non-terminal N is **nullable**, if starting with a sentential form N, we can produce an empty sentential form.

  example:

  expression → λ

- A non-terminal N is **useless**, if it can never produce a string of terminal symbols.

  example:

  expression → + expression
  | - expression

Grammar Transformations

Left factorization

$$X\ Y \mid X\ Z \Rightarrow X\,(\,Y \mid Z\,)$$

X Y=λ Z

Example:

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

```
single-Command
   ::= V-name := Expression
     | if Expression then single-Command
                (  λ  | else single-Command)
```

Grammar Transformations (ctd)

Elimination of Left Recursion

$$N \; ::= \; X \mid N \; Y \quad \Rightarrow \quad N \; ::= \; X \; Y^*$$

$$N \; ::= \; X \mid N \; Y \quad \Rightarrow \quad \begin{array}{l} N \; ::= \; X \; M \\ M \; ::= \; Y \; M \mid \lambda \end{array}$$

Example:

```
Identifier ::= Letter
             | Identifier Letter
             | Identifier Digit
```

```
Identifier ::= Letter
             | Identifier (Letter|Digit)
```

```
Identifier ::= Letter (Letter|Digit)*
```

Grammar Transformations (ctd)

Substitution of non-terminal symbols

$$\begin{aligned} N &::= X \\ M &::= \alpha\ N \beta \end{aligned} \quad \Rightarrow \quad \begin{aligned} N &::= X \\ M &::= \alpha\ X \beta \end{aligned}$$

Example:

```
single-Command
   ::= for contrVar := Expression
       to-or-dt Expression do single-Command
to-or-dt ::= to | downto
```

```
single-Command ::=
    for contrVar := Expression
    (to|downto) Expression do single-Command
```

From tokens to parse tree

The process of finding the structure in the flat stream of tokens is called **parsing**, and the module that performs this task is called **parser**.

Parsing methods

There are two well-known ways to parse:

1) top-down

Left-scan, **L**eftmost derivation (**LL**).

2) bottom-up

Left-scan, **R**ightmost derivation in reverse (**LR**).

- LL constructs the parse tree in pre-order;
- LR in post-order.

Different kinds of Parsing Algorithms

- Two big groups of algorithms can be distinguished:
  - bottom up strategies
  - top down strategies
- Example parsing of "Micro-English"

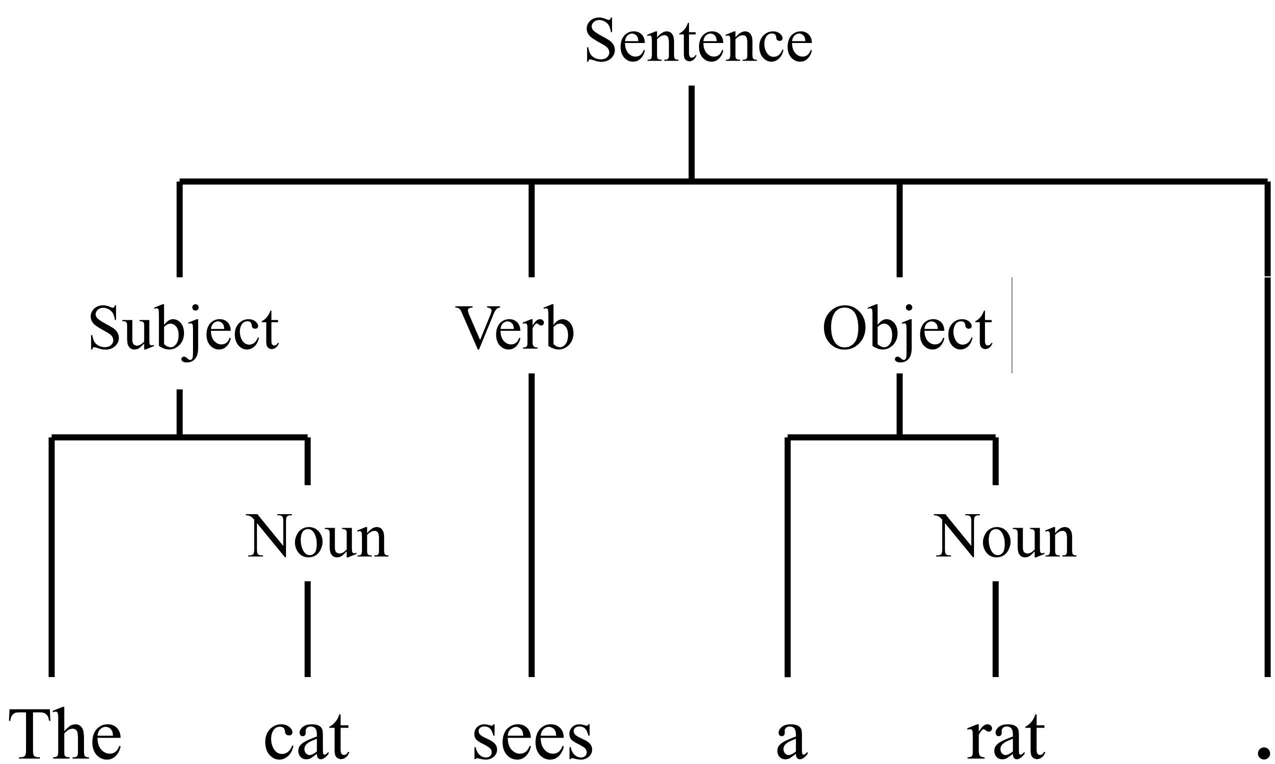
```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

The cat sees the rat.
The rat sees me.
I like a cat

The rat like me.
I see the rat.
I sees a rat.

Top-down parsing

The parse tree is constructed starting at the top (root).



Left derivations

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Sentence

→ Subject Verb Object .

→ The Noun Verb Object.

→ The cat Verb Object.

→ The cat sees Object.

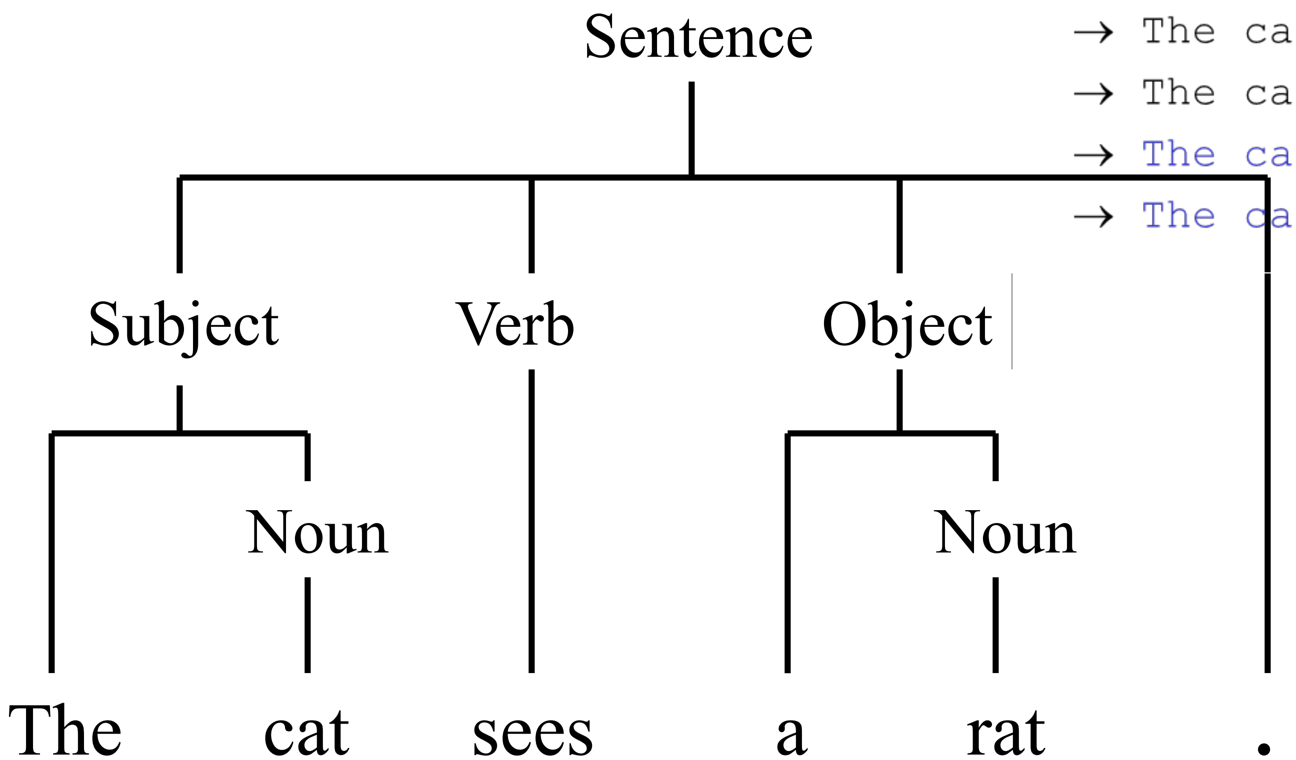
→ The cat sees a Noun .

→ The cat sees a rat .

Top-down parsing

The parse tree is constructed starting at the top (root).

```
Sentence
→ Subject Verb Object .
→ The Noun Verb Object.
→ The cat Verb Object.
→ The cat sees Object.
→ The cat sees a Noun .
→ The cat sees a rat .
```



Right derivations

```
Sentence   ::= Subject Verb Object .
Subject    ::= I | a Noun | the Noun
Object     ::= me | a Noun | the Noun
Noun       ::= cat | mat | rat
Verb       ::= like | is | see | sees
```

Sentence

→ Subject Verb Object .

→ Subject Verb a Noun .

→ Subject Verb a rat .

→ Subject sees a rat .

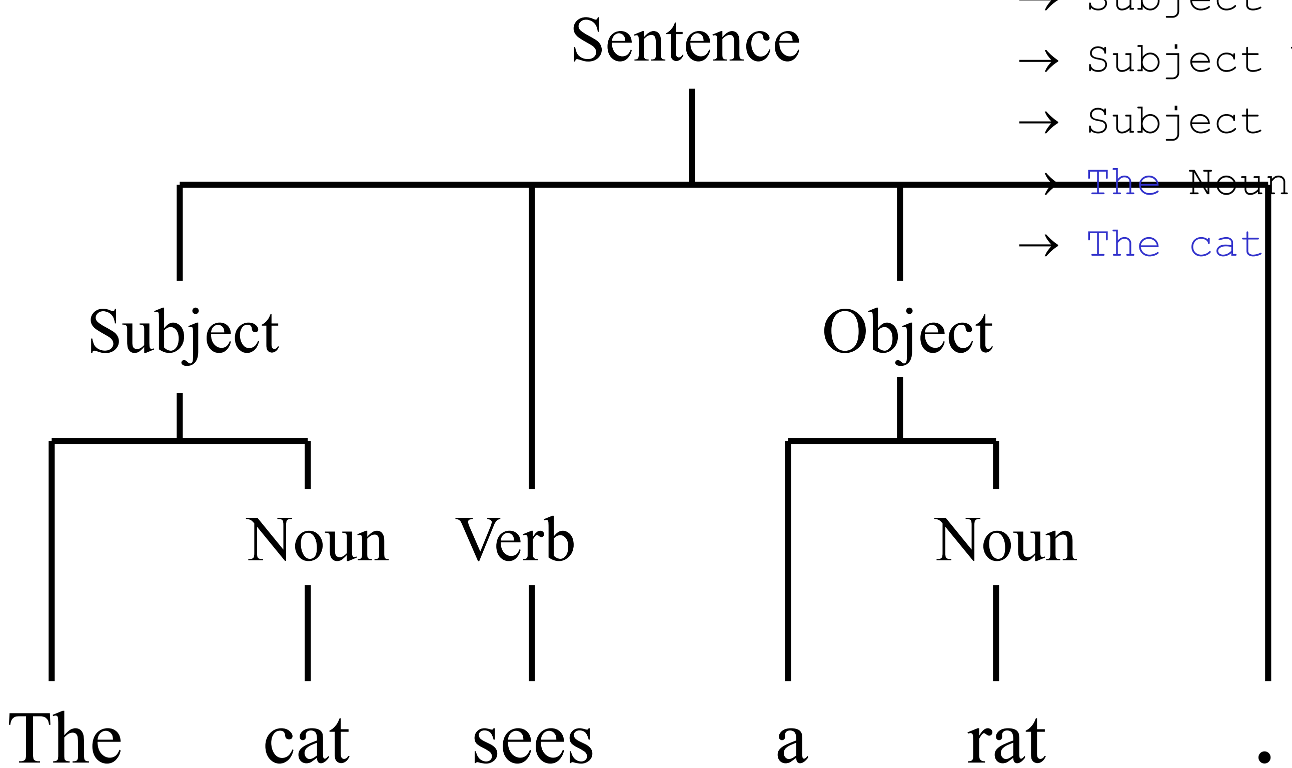
→ The Noun sees a rat .

→ The cat sees a rat .

Bottom up parsing

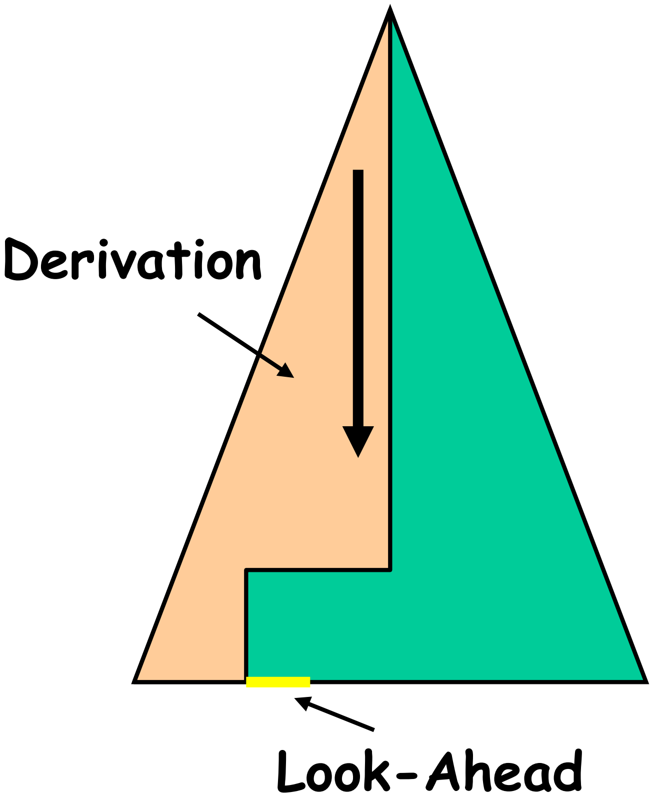
The parse tree "grows" from the bottom (leafs) up to the top (root).
Just read the right derivations backwards

```
Sentence
→ Subject Verb Object .
→ Subject Verb a Noun .
→ Subject Verb a rat .
→ Subject sees a rat .
→ The Noun sees a rat .
→ The cat sees a rat .
```

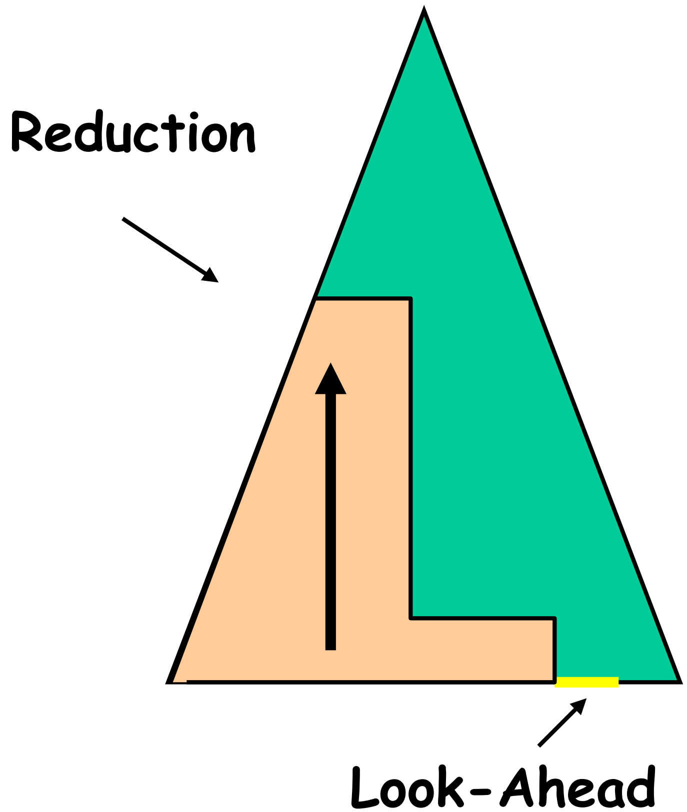


Top-Down vs. Bottom-Up parsing

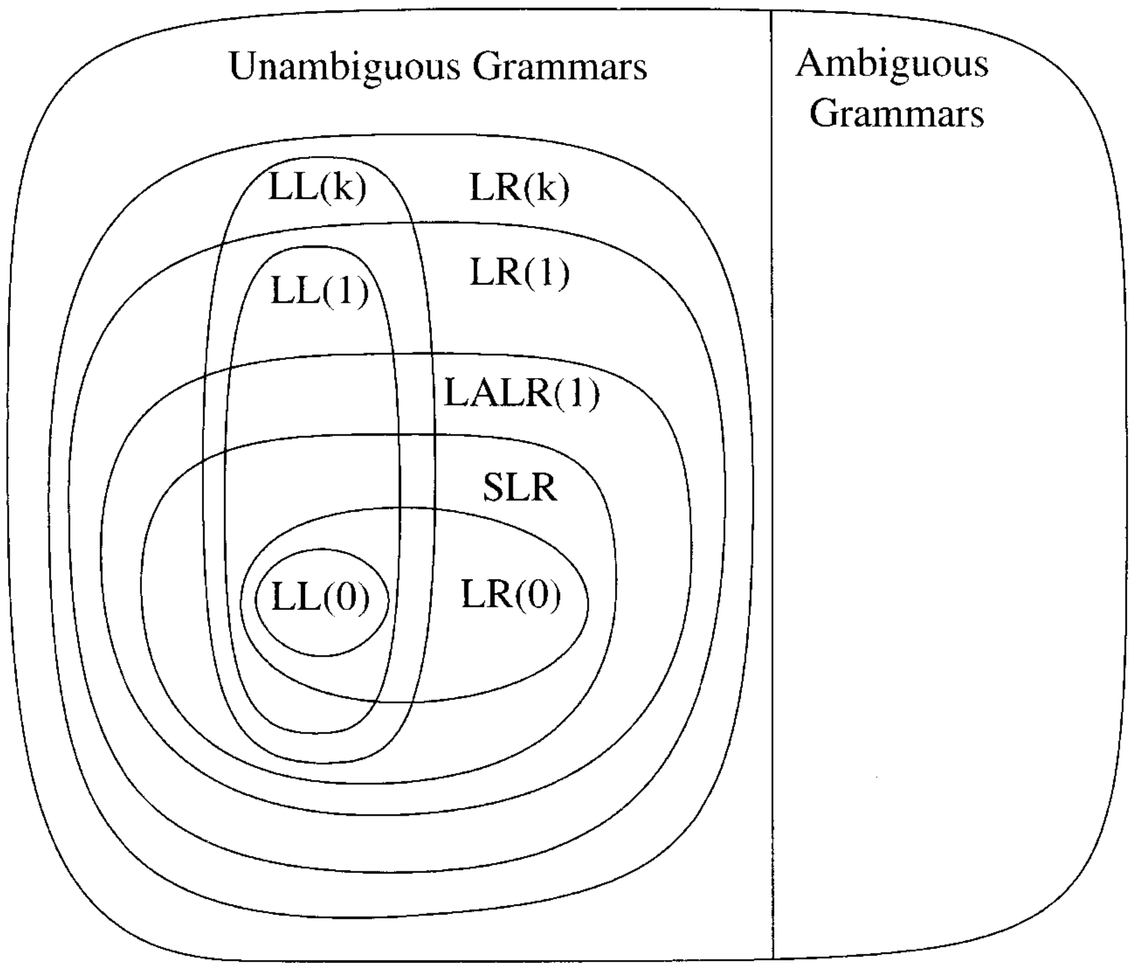
LL-Analyse (Top-Down)
Left-to-Right Left Derivative



LR-Analyse (Bottom-Up)
Left-to-Right Right Derivative



Hierarchy



Unambiguous Grammars
Ambiguous Grammars
LL(k)
LR(k)
LR(1)
LL(1)
LALR(1)
SLR
LL(0)
LR(0)

Pause

Formal definition of LL(1)

A grammar G is LL(1) iff
for each set of productions $X ::= X_1 \mid X_2 \mid \ldots \mid X_n$:

1. $first[X_1], first[X_2], \ldots, first[X_n]$ are all pairwise disjoint
2. If $X_i \Rightarrow^* \lambda$ then $first[X_j] \cap follow[X] = \emptyset$, for $1 \leq j \leq n. i \neq j$

If G is λ-free then 1 is sufficient

Define FIRST(α),where α is any string of grammar symbols, to be:
the set of terminals
that begin strings derived from α

First Sets

- The set of all terminal symbols that can begin a sentential form derivable from the string α
  - $\text{First}(\alpha)=\{\, a\in\Sigma \mid \alpha \Rightarrow^{*} a\beta \,\}$
  - We never include λ in First(α) even if $\alpha \Rightarrow \lambda$
  - E.g. (in Fig.4.1)
    - First(Tail) = {+}
    - First(Prefix) = {f}
    - First(E) = {v, f, (}

1	E	$\rightarrow$	Prefix (E)
2		$\mid$	v Tail
3	Prefix	$\rightarrow$	f
4		$\mid$	λ
5	Tail	$\rightarrow$	+ E
6		$\mid$	λ

```
function First(α) returns Set
    foreach A ∈ NonTerminals( ) do VisitedFirst(A) ← false        (9)
    ans ← InternalFirst(α)
    return (ans)
end
function InternalFirst(𝒳β) returns Set
    if 𝒳β = ⊥                                                     (10)
    then return (∅)
    if 𝒳 ∈ Σ                                                      (11)
    then return ({𝒳})
    /*  𝒳 is a nonterminal.                                    */ (12)
    ans ← ∅
    if not VisitedFirst(𝒳)
    then
        VisitedFirst(𝒳) ← true                                    (13)
        foreach rhs ∈ ProductionsFor(𝒳) do
            ans ← ans ∪ InternalFirst(rhs)                        (14)
    if SymbolDerivesEmpty(𝒳)                                      (15)
    then ans ← ans ∪ InternalFirst(β)
    return (ans)                                                  (16)
end
```

Figure 4.8: Algorithm for computing First(α).

Follow Sets

- The set of terminals that can follow a nonterminal A in some sentential form
  - For $A \in N$,
    - $\text{Follow}(A) = \{b \in \Sigma \mid S \Rightarrow^{+} \alpha A b \beta\}$
  - The right context associated with A
  - Fig. 4.11

Follow Sets

- Follow(A) is the set of prefixes of strings of terminals that can follow any derivation of A in G
  - $\$ \in$ follow(S) (sometimes <eof> $\in$ follow(S))
  - if $(B \rightarrow \alpha A \beta) \in P$, then
  - first(β)$\oplus$follow(B)$\subseteq$ follow(A)

- The definition of follow usually results in recursive set definitions. In order to solve them, you need to do several iterations on the equations.
  - E.g. (in Fig.4.1)
    - Follow(Tail) = {)}
    - Follow(Prefix) = {(}
    - Follow(E) = {$,)}

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

```
function Follow( A ) returns Set
    foreach A ∈ NonTerminals( ) do
        VisitedFollow(A) ← false                                   17
    ans ← InternalFollow( A )
    return (ans)
end
function InternalFollow( A ) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                         18
    then
        VisitedFollow(A) ← true                                    19
        foreach a ∈ Occurrences( A ) do                            20
            ans ← ans ∪ First( Tail( a ) )                         21
            if AllDeriveEmpty( Tail( a ) )                         22
            then
                targ ← LHS( Production( a ) )
                ans ← ans ∪ InternalFollow( targ )                 23
    return (ans)                                                   24
end
function AllDeriveEmpty( γ ) returns Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ
        then return (false)
    return (true)
end
```

Figure 4.11: Algorithm for computing Follow(A).

A few provable facts about LL(1) grammars

- No left-recursive grammar is LL(1)
- No ambiguous grammar is LL(1)
- Some languages have no LL(1) grammar
- A λ-free grammar, where each alternative X_j for N ::= X_j begins with a distinct terminal, is a simple LL(1) grammar

LR Grammars

- A Grammar is an LR Grammar if it can be parsed by an LR parsing algorithm
- Harder to implement LR parsers than LL parsers
  - but tools exist (e.g. JavaCUP, Yacc, C#CUP and SableCC)
- Can recognize LR(0), LR(1), SLR, LALR grammars (bigger class of grammars than LL)
  - Can handle left recursion!
  - Usually more convenient because less need to rewrite the grammar.
- LR parsing methods are the most commonly used for automatic tools today (LALR in particular)

Other Types of Grammars

- Regular grammars: less powerful
- Context-sensitive and unrestricted grammars: more powerful
- Parsing Expression Grammars

Designing CFGs is a craft.

- When thinking about CFGs:
  - Think recursively: Build up bigger structures from smaller ones.
- Have a construction plan:
  - Know in what order you will build up the string.
- Store information in nonterminals:
  - Have each nonterminal correspond to some useful piece of information.

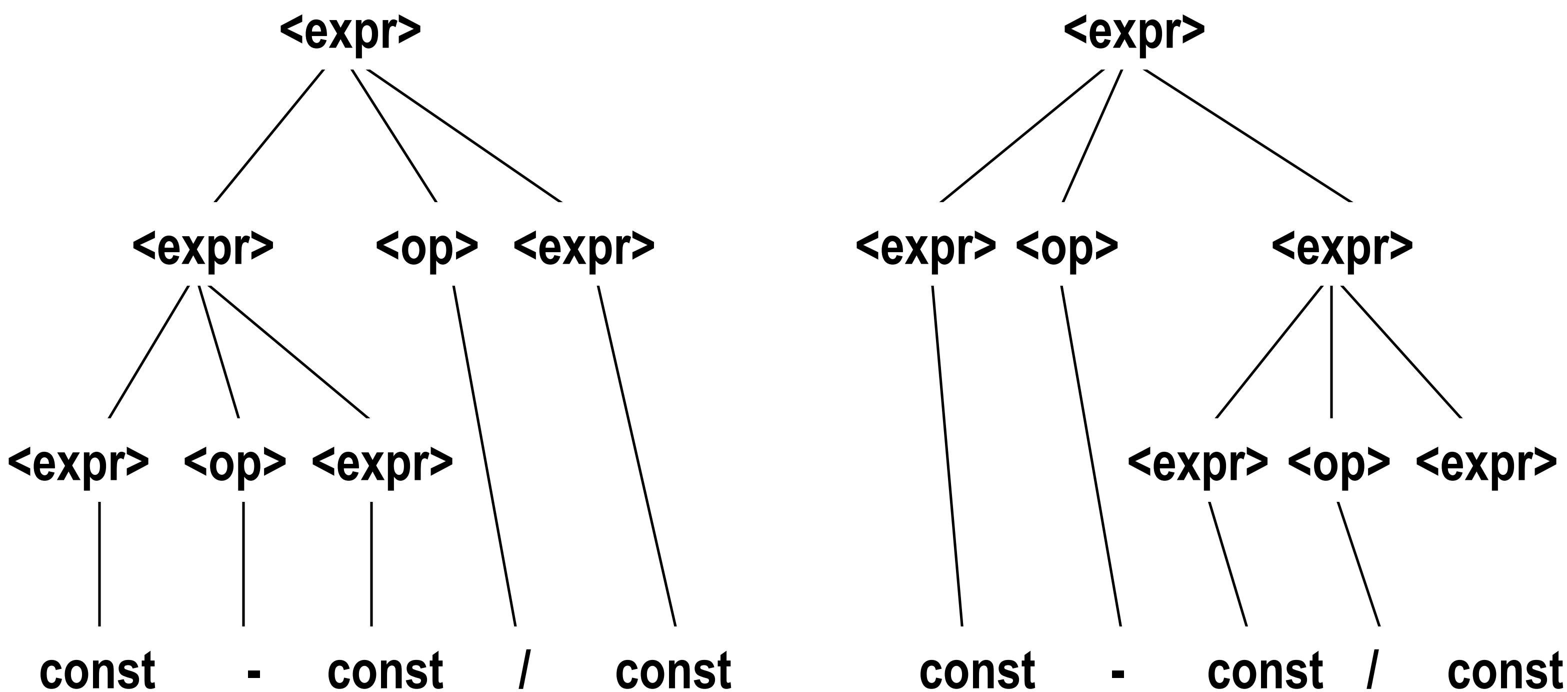
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

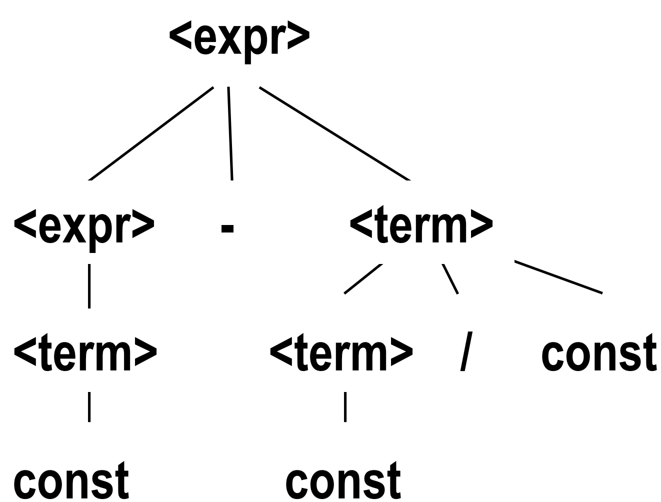
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```

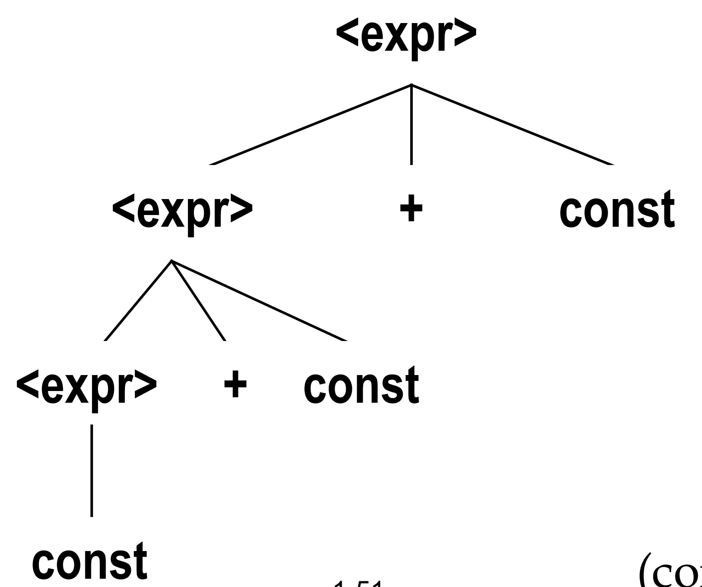


const – (const / const)

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



(const + const) + const

Associativity and Left Resursion

```
<expr> -> <expr> + const  |  const
(unambiguous, but left recursive)

<expr> -> const + <expr>  |  const
(unambiguous, right recursive, but => right assoc.)

i.e. const + (const + const)
Not a problem for +, but what about - ?

(5 - 3) - 2 = 0
5 - (3 - 2) = 4
```

Eliminating Left recursion

```
<expr> -> <expr> (+ <expr>)*

or

<expr> -> const <exprlist>
<exprlist> -> + const <exprlist> | λ
```

Still gives the wrong parse tree, but this can be sorted when generating AST

Hidden left-factors and hidden left recursion

- Sometimes, left-factors or left recursion are hidden
- Examples:
  - The following grammar:
    - A -> da | ac B
    - B -> ab B | da A | A f
  - has two overlapping productions: B -> da A and B =>*daf .
  - The following grammar:
    - S -> T u | wx
    - T -> S q | vv S
  - has left recursion on T (T =>* Tuq)

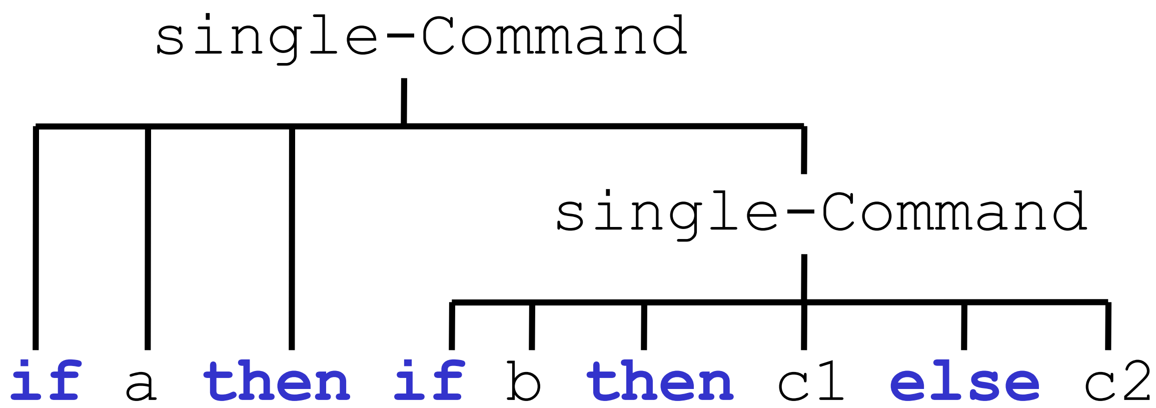
- Solution: expand the production rules by substitution to make
- left-recursion or left factors visible and then eliminate them

Dangling Else Problem

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                      else single-Command
```

This parse tree?

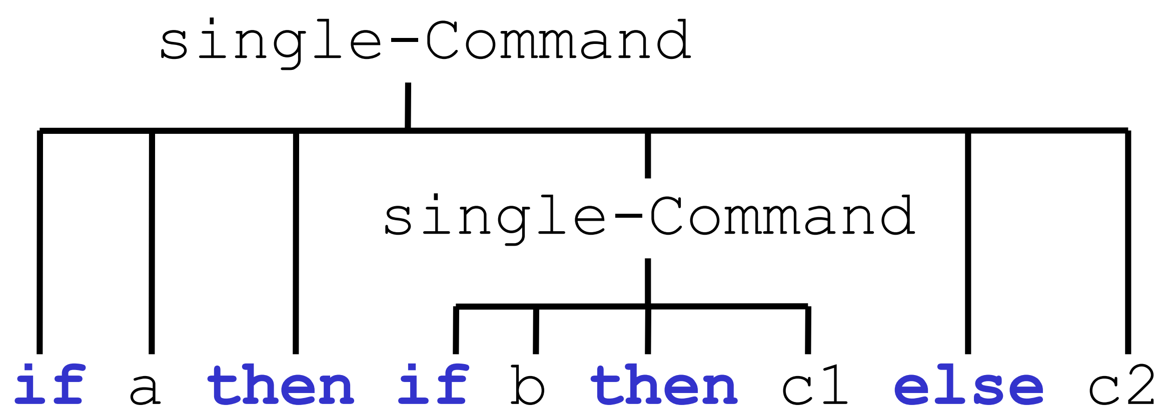


Dangling Else Problem

Example: (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                         else single-Command
```

or this one ?



Dangling Else Problem

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

Rewrite Grammar:

```
sC ::= if E then sC endif
     |  if E then sC else sC endif
```

Dangling Else Problem

Example: "dangling-else" problem (from Mini Triangle grammar)

```
single-Command
   ::= if Expression then single-Command
     | if Expression then single-Command
                          else single-Command
```

Rewrite Grammar:

```
sC   ::= CsC
      |   OsC
CsC  ::= if E then CsC else CsC
CsC  ::= …
OsC  ::= if E then sC
      |   if E then CsC else OsC
```

Ambiguity

- Sometimes obvious
  - Exp ::= Exp + Exp
- Sometimes difficult to spot
- Undecidable Property (known since 1962)

- Engineering approach
  - Try a parser generator
  - Use a Grammar engineering toolbox
    - KfG in AtoCC
    - Context Free Grammer tools
      - http://smlweb.cpsc.ucalgary.ca/start.html
      - http://mdaines.github.io/grammophone/

- Try ACLA
  - (Ambiguity Checking with Language Approximations)
  - http://services2.brics.dk/java/grammar/demo.html

What can you do in your project?

- Start writing a CFG
  - Define keywords, identifiers, numbers, ..
  - Define productions
- Test it with
  - kfG Edit
  - Context Free Grammer tool
  - ACLA

You may need more than one Grammar

- Abstract Syntax
  - To communicate the essentials of the language
  - To serve as design pattern for AST
  - To serve in the formal specification of the semantics
  - May be ambiguous
- Concrete Syntax
  - The grammar we use as specification for building a parser
  - Must be unambiguous

- Lexical elements (Syntax given as Regular Expressions)
  - Identifiers e.g. Id := [a-z]([a-z]|[0-9])*
  - Keywords (or reserved words)
    - if, then, while,
    - begin .. end v.s. { .. }

Grammar tools

- Demo
  - Prefix
  - Exp with ambiguity and without
  - Dangling else
  - LL(1) – first and follow