

Lecture 5 – Formal grammars, regular and context-free grammars

NTIN071 Automata and Grammars

Jakub Bulín (KTIML MFF UK)

Spring 2024

** Adapted from the Czech-lecture slides by Marta Vomlelová with gratitude.
The translation, some modifications, and all errors are mine.*

Recap of Lecture 4

- regular expressions
- Kleene's theorem (two variants)
- constructions: RE to ϵ -NFA, DFA to RE
- state elimination algorithm
- string substitution, homomorphism, inverse homomorphism
- decision properties

CHAPTER 2: GRAMMARS

2.1 Formal grammars

Palindromes are not regular

palindrome: $w = w^R$, e.g. racecar, step_on_no_pets

Example

The language $L_{\text{pal}} = \{w \in \{0,1\}^* \mid w = w^R\}$ is not regular.

(A standard Pumping lemma argument using $w = 0^n 1 0^n$.)

How to represent L_{pal} ? We can use a (context-free) grammar, $G = (\{S\}, \{0,1\}, \mathcal{P}, S)$ with the following set of production rules:

$$\begin{aligned}\mathcal{P} = \{ & S \rightarrow \epsilon, \\ & S \rightarrow 0, \\ & S \rightarrow 1, \\ & S \rightarrow 0S0, \\ & S \rightarrow 1S1\}\end{aligned}$$

For brevity, we also write $\mathcal{P} = \{S \rightarrow \epsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1\}$.

Formal grammar: the definition

A **formal (generative) grammar**: $G = (V, T, \mathcal{P}, S)$ where

- V is a finite nonempty set of **nonterminals (variables)**
- T is a finite set of **terminal symbols (terminals)**, $V \cap T = \emptyset$
- $S \in V$ is the **start symbol**
- \mathcal{P} is a finite set of **production rules** of the form $\alpha \rightarrow \beta$ where
 - $\alpha \in (V \cup T)^+ \setminus T^+$, the **head** (must contain some variable!)
 - $\beta \in (V \cup T)^*$, the **body**

A grammar is **context-free** (a **CFG**) if the head is a single variable, i.e., the rules are of the form $A \rightarrow \beta$ for $A \in V$ and $\beta \in (V \cup T)^*$.

The production rules thus represent a **recursive definition of the language**, starting from the start symbol (see the example).

Derivation, the language of a grammar

Let $G = (V, T, \mathcal{P}, S)$ be a grammar.

- γ **one-step derives** δ (write $\gamma \Rightarrow_G \delta$ or just $\gamma \Rightarrow \delta$) if $\gamma = \eta\alpha\nu$ and $\delta = \eta\beta\nu$ for some $\alpha \rightarrow \beta \in \mathcal{P}$ and $\eta, \nu \in (V \cup T)^*$
- γ **derives** δ (write $\gamma \Rightarrow_G^* \delta$ or just $\gamma \Rightarrow^* \delta$) if there are $\beta_1, \dots, \beta_n \in (V \cup T)^*$ s.t. $\gamma = \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_n = \delta$ (Note that always $\gamma \Rightarrow^* \gamma$.)
- the sequence β_1, \dots, β_n is a **derivation** of δ from γ , it is **minimal** if $\beta_i \neq \beta_j$ for $i \neq j$
- a **sentential form** is any $\delta \in (V \cup T)^*$ such that $S \Rightarrow_G^* \delta$

The language **generated by** G , $L(G)$ consists of words over the terminals derivable from the start symbol:

$$L(G) = \{\omega \in T^* \mid S \Rightarrow_G^* \omega\}$$

(Similarly, for any $A \in V$ define $L(A) = \{\omega \in T^* \mid A \Rightarrow_G^* \omega\}$.)

2.2 Chomsky hierarchy

Chomsky hierarchy (of grammars)

Restricting the form of production rules:

- Type 0: **general grammars**
 - $\alpha_1 A \alpha_2 \rightarrow \beta$ where $A \in V$, $\alpha_1, \alpha_2, \beta \in (V \cup T)^*$
 - recursively enumerable languages \mathcal{L}_0
- Type 1: **context-sensitive grammars**
 - $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \gamma \alpha_2$, $A \in V$, $\gamma \in (V \cup T)^+$, $\alpha_1, \alpha_2 \in (V \cup T)^*$
 - note: the variable must be rewritten to at least one symbol
 - sometimes we allow $S \rightarrow \epsilon$, then S cannot appear in any body
 - context-sensitive languages \mathcal{L}_1
- Type 2: **context-free grammars**
 - $A \rightarrow \beta$ where $A \in V$, $\beta \in (V \cup T)^*$
 - context-free languages \mathcal{L}_2
- Type 3: **right-linear grammars** (aka regular grammars)
 - $A \rightarrow \omega B$ or $A \rightarrow \omega$ where $A, B \in V$, $\omega \in T^*$
 - regular languages \mathcal{L}_3

The classes are ordered by (strict) inclusion

$$\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3$$

- context-sensitive languages are recursively enumerable: the head of a CSG contains a variable
- context-free languages are context-sensitive: the context α_1, α_2 is empty; we can **eliminate ϵ -productions** $A \rightarrow \epsilon$
- regular languages are context-free: body can have any form
- strict inclusion: we will give examples later

2.3 Regular grammars

Right-linear grammars

A grammar G is **right-linear** (**regular**, **type 3**), if its production rules are of the form $A \rightarrow \omega B$ or $A \rightarrow \omega$ where $A, B \in V$, $\omega \in T^*$.

(At most one variable in the body, it can only be at the end.)

Example

$G = (\{S, A, B\}, \{0, 1\}, \mathcal{P}, S)$ where

$$\mathcal{P} = \{S \rightarrow 0S \mid 1A \mid \epsilon, A \rightarrow 0A \mid 1B, B \rightarrow 0B \mid 1S\}$$

A derivation of $01101 \in L(G)$:

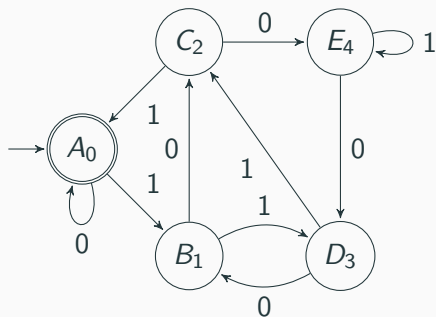
$$S \Rightarrow 0S \Rightarrow 01A \Rightarrow 011B \Rightarrow 0110B \Rightarrow 01101S \Rightarrow 01101$$

Corresponds to FA: nonterminals are states, generate means read.

Theorem

A language is regular, iff it is generated by a right-linear grammar.

Example: binary numbers divisible by 5



$$A \rightarrow 1B \mid 0A \mid \epsilon$$

$$B \rightarrow 0C \mid 1D$$

$$C \rightarrow 0E \mid 1A$$

$$D \rightarrow 0B \mid 1C$$

$$E \rightarrow 0D \mid 1E$$

Derivation examples

$$A \Rightarrow 0A \Rightarrow 0 \quad (n = 0)$$

$$A \Rightarrow 1B \Rightarrow 10C \Rightarrow 101A \Rightarrow 101 \quad (n = 5)$$

$$A \Rightarrow 1B \Rightarrow 10C \Rightarrow 101A \Rightarrow 1010A \Rightarrow 1010 \quad (n = 10)$$

$$A \Rightarrow 1B \Rightarrow 11D \Rightarrow 111C \Rightarrow 1111A \Rightarrow 1111 \quad (n = 15)$$

Finite automaton to right-linear grammar

Given a DFA $A = (Q, \Sigma, \delta, q_0, F)$ define a right-linear grammar $G = (Q, \Sigma, \mathcal{P}, q_0)$, i.e. nonterminals are states, with productions:

- $p \rightarrow aq$ for all transitions $\delta(p, a) = q$
- $p \rightarrow \epsilon$ for every final state $p \in F$

To show that $L(A) = L(G)$:

- For the empty word:

$$\epsilon \in L(A) \text{ iff } q_0 \in F \text{ iff } q_0 \rightarrow \epsilon \in \mathcal{P} \text{ iff } \epsilon \in L(G)$$

- For a word $w = a_1 \dots a_n$: $a_1 \dots a_n \in L(A)$ iff there are $q_0, \dots, q_n \in Q$ s.t. $\delta(q_i, a_{i+1}) = q_{i+1}$ for $i < n$ and $q_n \in F$.
This means that $q_0 \Rightarrow a_1 q_1 \Rightarrow \dots \Rightarrow a_1 \dots a_n q_n \Rightarrow a_1 \dots a_n$ is derivation of $a_1 \dots a_n$, which shows that $a_1 \dots a_n \in L(G)$. \square

(Note: Same construction works for NFA or ϵ -NFA.)

Right linear grammar to finite automaton

Given a right-linear grammar we construct a ϵ -NFA.

Encoding productions based on their form:

- $A \rightarrow aB$ are encoded directly as transitions
- $A \rightarrow \epsilon$ (**ϵ -productions**) define accepting states
- $A \rightarrow B$ (**unit productions**) correspond to ϵ -transitions

Productions with more terminals, $A \rightarrow a_1 \dots a_n B$ or $A \rightarrow a_1 \dots a_n$:

- introduce new variables Y_1, Y_2, \dots, Y_{n+1}
- replace with $A \rightarrow a_1 Y_2, Y_2 \rightarrow a_2 Y_3, \dots, Y_{n-1} \rightarrow a_{n-1} Y_n$,
and finally either $Y_n \rightarrow a_n B$ or $Y_n \rightarrow a_n Y_{n+1}, Y_{n+1} \rightarrow \epsilon$

Similarly, $A \rightarrow a$ can be rewritten to $A \rightarrow aY, Y \rightarrow \epsilon$.

(Think of a state diagram but edges labelled with words, subdivide them. For edges pointing nowhere, add a new final state.)

Standardization of a right-linear grammar

Sometimes we want to get rid of unit productions too, this can be done by taking transitive closure (same as removing ϵ -transitions).

We call grammars G and G' **equivalent** if $L(G) = L(G')$.

Lemma

For any right-linear grammar G there exist an equivalent G' which only has productions of the form $A \rightarrow aB$ or $A \rightarrow \epsilon$.

Formalizing the previous slide, define $G' = (V', T, \mathcal{P}', S)$ where V' contains the original variables V plus all new variables used for encoding. Productions \mathcal{P}' are as described.

To remove unit productions ($A \rightarrow B$), take the transitive closure $U(A) = \{B \in V \mid A \Rightarrow^* B\}$. For every production $B \rightarrow \gamma \in \mathcal{P}$ with $B \in U(A)$ add the production $A \rightarrow \gamma$ to \mathcal{P}' . □

Formalizing the construction of an automaton

Given a right-linear grammar, first standardize it: $G = (V, T, \mathcal{P}, S)$ with productions only of the form $A \rightarrow aB$ or $A \rightarrow \epsilon$.

Define an NFA $A = (Q, \Sigma, \delta, S_0, F)$, where $Q = V$, $\Sigma = T$, $S_0 = \{S\}$, $F = \{A \mid A \rightarrow \epsilon \in \mathcal{P}\}$, and the transitions are:

$$\delta(A, a) = \{B \mid A \rightarrow aB \in \mathcal{P}\} \text{ for } A \in V, a \in T$$

To show that $L(G) = L(A)$: For the empty word, $\epsilon \in L(G)$ iff $S \rightarrow \epsilon \in \mathcal{P}$ iff $S \in F$ iff $\epsilon \in L(A)$. Otherwise, $w = a_1 \dots a_n \in L(G)$ iff there is a derivation $S \Rightarrow a_1 X_1 \Rightarrow \dots \Rightarrow a_1 \dots a_n X_n \Rightarrow a_1 \dots a_n$.

Equivalently, in A there are states $X_0, X_1, \dots, X_n \in Q$ such that $X_0 = S \in S_0$, $X_n \in F$ and $X_{i+1} \in \delta(X_i, a_i)$. But this means that $a_1 \dots a_n \in L(A)$. □

(Note: Easier to leave unit productions in, construct an ϵ -NFA:
 $\delta(A, \epsilon) = \{B \mid A \rightarrow B \in \mathcal{P}\}.$)

Linear grammars and linear languages

A context-free grammar is

- **left-linear** if productions are of the form $A \rightarrow B\omega$ or $A \rightarrow \omega$,
- **linear** if productions are of the form $A \rightarrow \omega B\omega'$ or $A \rightarrow \omega$

where $A, B \in V$ and $\omega, \omega' \in T^*$. A language is **linear** if it is generated by some linear grammar.

- Left-linear grammars generate regular languages. (L is regular iff L^R is, reversing bodies gives a right-linear grammar.)
- But not every linear language is regular! Example:
 $L = \{0^n 1^n \mid n \geq 1\}$, linear rules $S \rightarrow 0S1 \mid 01$
- Observe: productions can be split to left-linear and right-linear
- Not every context-free language is linear, for example
 $L = \{w \in \{0, 1\}^* \mid |w|_0 = |w|_1\}$. Context-free grammar later, to prove non-linearity use a version of PL for linear languages.

2.4 Context-free grammars

Example: simple expressions

Recall that in a CFG, the head always consists of a single variable.

Consider $G = (\{E, I\}, \{+, *, (,), a, b, 0, 1\}, \mathcal{P}, E)$ where

$$\begin{aligned}\mathcal{P} = \{ & E \rightarrow I, \\ & E \rightarrow E + E, \\ & E \rightarrow E * E, \\ & E \rightarrow (E),\end{aligned}$$

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1\}$$

- Rules 1-4 describe expressions E .
- Rules 5-10 describe identifiers I , correspond to the regular expression $(a + b)(a + b + 0 + 1)^*$.

Parse trees

The tree is the data structure of choice to represent the source program in a compiler, facilitating translation into executable code.

Derivations from CFG naturally correspond to trees. Apply a production \rightsquigarrow append symbols from body as children of head.

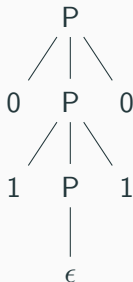
(a) Expressions:

$$E \Rightarrow^* a + E$$



(b) Palindromes:

$$P \Rightarrow^* 0110$$



The definition

A **parse tree** for a CFG is a labeled ordered tree such that:

- inner nodes are labeled by variables from V
- the root is labeled by the start symbol S
- leaves have labels from $V \cup T \cup \{\epsilon\}$
- if a leaf is labeled ϵ , it must be the only child of its parent
- if an inner node is labeled A and its children X_1, \dots, X_k (ordered left-to-right), then $A \rightarrow X_1, \dots, X_k \in P$

The **yield** of a parse tree is the string $\gamma \in (V \cup T)^*$ obtained by reading the labels on all leaves left-to-right.

Note: yields containing only terminals \iff words from the language

Leftmost and rightmost derivations

Leftmost derivation $\Rightarrow_{lm}, \Rightarrow_{lm}^*$: in each step rewrite the leftmost (first) variable; **rightmost** $\Rightarrow_{rm}, \Rightarrow_{rm}^*$: rewrite the rightmost (last)

Example: Same productions for each variable but different order

- **leftmost:** $E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E \Rightarrow_{lm} a * (E) \Rightarrow_{lm} a * (E + E) \Rightarrow_{lm} a * (I + E) \Rightarrow_{lm} a * (a + E) \Rightarrow_{lm} a * (a + I) \Rightarrow_{lm} a * (a + I0) \Rightarrow_{lm} a * (a + I00) \Rightarrow_{lm} a * (a + b00)$
- **rightmost:** $E \Rightarrow_{rm} E * E \Rightarrow_{rm} E * (E) \Rightarrow_{rm} E * (E + E) \Rightarrow_{rm} E * (E + I) \Rightarrow_{rm} E * (E + I0) \Rightarrow_{rm} E * (E + I00) \Rightarrow_{rm} E * (E + b00) \Rightarrow_{rm} E * (I + b00) \Rightarrow_{rm} E * (a + b00) \Rightarrow_{rm} I * (a + b00) \Rightarrow_{rm} a * (a + b00)$

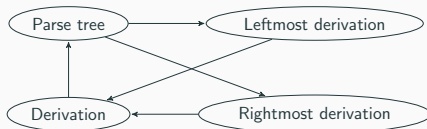
Derivations and parse trees

Theorem

Given a context-free grammar $G = (V, T, P, S)$ and $\gamma \in (V \cup T)^*$, the following are equivalent:

- (i) $A \Rightarrow^* \gamma$
- (ii) $A \Rightarrow_{lm}^* \gamma$
- (iii) $A \Rightarrow_{rm}^* \gamma$
- (iv) There is a parse tree with root A and yield γ .

Proof (ii) \Rightarrow (i) and (iii) \Rightarrow (i) are trivial. We will show (i) \Rightarrow (iv) and (iv) \Rightarrow (ii) [analogously (iv) \Rightarrow (iii)].



Parse tree to leftmost derivation: an example



- Root: $E \Rightarrow_{lm} E * E$
- Leftmost child of the root: $E \Rightarrow_{lm} I \Rightarrow_{lm} a$
- Rightmost child of the root:
 $E \Rightarrow_{lm} (E) \Rightarrow_{lm} (E + E) \Rightarrow_{lm} (I + E) \Rightarrow_{lm} (a + E)$
 $\Rightarrow_{lm} (a + I) \Rightarrow_{lm} (a + I0) \Rightarrow_{lm} (a + I00) \Rightarrow_{lm}$
 $(a + b00)$
- Leftmost integrated to root:
 $E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E$
- Full derivation:
 $E \Rightarrow_{lm} E * E \Rightarrow_{lm} I * E \Rightarrow_{lm} a * E \Rightarrow_{lm}$
 $\Rightarrow_{lm} a * (E) \Rightarrow_{lm} a * (E + E) \Rightarrow_{lm} a * (I + E) \Rightarrow_{lm}$
 $\Rightarrow_{lm} a * (a + E) \Rightarrow_{lm} a * (a + I) \Rightarrow_{lm} a * (a + I0) \Rightarrow_{lm}$
 $\Rightarrow_{lm} a * (a + I00) \Rightarrow_{lm} a * (a + b00)$

Parse tree to leftmost derivation: the proof

Observe: If $\beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_n$ is a derivation, then for any $\alpha, \alpha' \in (V \cup T)^*$, $\alpha\beta_1\alpha' \Rightarrow \alpha\beta_2\alpha' \Rightarrow \dots \Rightarrow \alpha\beta_n\alpha'$ is a derivation.

Suppose we have a parse tree with root A and yield γ . Induction on the depth of the tree.

Base: depth 1, root A with children that read γ

$A \rightarrow \gamma$ is a production, thus $A \Rightarrow_{lm} \gamma$ is a one-step derivation

Induction step: depth $n > 1$, root A with children X_1, X_2, \dots, X_k

- If X_i is a terminal, define $\gamma_i = X_i$
- If X_i is a variable, then by induction $X_i \Rightarrow_{lm}^* \gamma_i$.

To construct the leftmost derivation, show by induction on

$i = 1, \dots, k$ that $A \Rightarrow_{lm}^* \gamma_1 \gamma_2 \dots \gamma_i X_{i+1} X_{i+2} \dots X_k$

Finally, when $i = k$, the result is a leftmost derivation of γ from A .

The induction within the induction

Assuming that $A \Rightarrow_{lm}^* \gamma_1 \gamma_2 \dots \gamma_{i-1} X_i X_{i+1} X_{i+2} \dots X_k$, show

$$A \Rightarrow_{lm}^* \gamma_1 \gamma_2 \dots \gamma_i X_{i+1} X_{i+2} \dots X_k$$

- If X_i is a terminal, do nothing, just increment i .
- If X_i is a variable, rewrite the derivation

$$X_i \Rightarrow_{lm} \alpha_1 \Rightarrow_{lm} \alpha_2 \dots \Rightarrow_{lm} \gamma_i$$

to the following:

$$\gamma_1 \gamma_2 \dots \gamma_{i-1} X_i X_{i+1} X_{i+2} \dots X_k \Rightarrow_{lm}$$

$$\gamma_1 \gamma_2 \dots \gamma_{i-1} \alpha_1 X_{i+1} X_{i+2} \dots X_k \Rightarrow_{lm}$$

$$\vdots$$

$$\Rightarrow_{lm} \gamma_1 \gamma_2 \dots \gamma_{i-1} \gamma_i X_{i+1} X_{i+2} \dots X_k$$

(To construct a rightmost derivation, go from $i = k$ down to 1.)

Derivation to parse tree

Suppose we have a derivation $A = \beta_1 \Rightarrow \beta_2 \Rightarrow \cdots \Rightarrow \beta_n = \gamma$.

We construct a parse tree with root A and yield γ by induction on the number of steps n in the derivation.

Base ($n = 1$): A is a single-vertex parse tree

Induction step ($n > 1$) : We have $A \Rightarrow^* \beta_{n-1} \Rightarrow \beta_n$.

Suppose $\beta_{n-1} = \alpha C \alpha'$ and $\beta_n = \alpha \delta \alpha'$ for a production $C \rightarrow \delta$.

By induction, we have a parse tree with root A and yield $\alpha C \alpha'$.

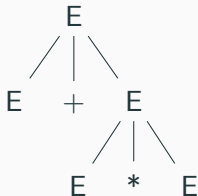
Find the leaf corresponding to C and append to it new leaves labelled by the symbols from δ .

This shows that (i) \Rightarrow (iv), and thus the theorem is proved. \square

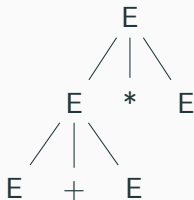
Ambiguity in grammars

Ambiguity: an example

$$E \Rightarrow E + E \Rightarrow E + E * E$$



$$E \Rightarrow E * E \Rightarrow E + E * E$$

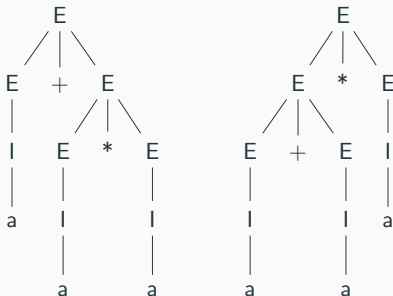


- Two different parse trees for the same expression.
- Important difference: $1 + (2 * 3) = 7$ but $(1 + 2) * 3 = 9$
- Imagine source file interpretable as two different programs.
- This grammar can be modified to remove ambiguity.
- Different derivations with the same parse tree are not an issue (e.g. left-most and right-most).

Ambiguous context-free grammars

A context-free grammar G is **ambiguous** if for some $\omega \in L(G)$ there exist two different parse trees with root S and yield ω . Otherwise the grammar is **unambiguous**.

Example: our grammar for simple expressions is ambiguous, $\omega = a + a * a \in L(G)$ is yielded by both of the following parse trees:



Inherent ambiguity of context-free languages

A context-free language L is **unambiguous** if there exists an unambiguous grammar generating it, and L is **inherently ambiguous** otherwise (i.e., if every CFG for L is ambiguous).

Example: $L = \{a^n b^n c^k d^k \mid n, k \geq 1\} \cup \{a^n b^k c^k d^n \mid n, k \geq 1\}$ is inherently ambiguous. Here is an ambiguous grammar with two parse trees for $\omega = aabbccdd$.

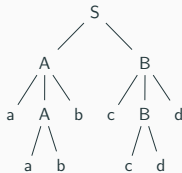
$$S \rightarrow AB \mid C$$

$$A \rightarrow aAb \mid ab$$

$$B \rightarrow cBd \mid cd$$

$$C \rightarrow aCd \mid aDd$$

$$D \rightarrow bDc \mid bc$$



Why **inherently**? Idea: any grammar will generate at least some $a^n b^n c^n d^n$ in the two different ways.

Removing ambiguity

- There is no algorithm deciding if a given CFG is ambiguous.
- There exist inherently ambiguous context-free languages (see the example above).
- There are certain techniques for removing ambiguity.

Different causes for ambiguity:

- The precedence of operators is not respected.
- A sequence of identical operators associates from left or right.
- E.g. for rules $S \rightarrow \text{if then } S \text{ else } S \mid \text{if then } S \mid \epsilon$, the word `if then if then else` has two meanings: `if then (if then else)` or `if then (if then) else`

Possible solutions:

- syntax error (Algol 60)
- else belongs to the closer if (rules ordered by preference)
- parentheses, begin—end, or indentation (Python)

Enforcing precedence

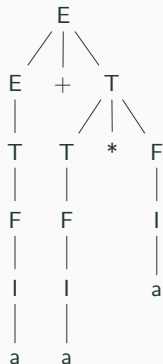
Introduce a new variable for each level of 'binding strength':

- a **factor** is an expression that cannot be broken by any operator: identifiers, parenthesized expressions
- a **term** is an expression not broken by $+$
- an **expression** can be broken by either $*$ or $+$.

An unambiguous grammar:

$$I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$
$$F \rightarrow I \mid (E)$$
$$T \rightarrow F \mid T * F$$
$$E \rightarrow T \mid E + T$$

right: the only parse tree for $a + a * a$



Summary of Lecture 5

- Grammars: general, context-sensitive, context-free, right-linear (regular) – Chomsky hierarchy
- The language of a grammar, derivation
- Right-linear grammars correspond to FA (and so do left/linear)
- Linear grammars are stronger
- Context-free grammars: parse tree and its yield
- (un)ambiguous grammars, inherently ambiguous languages

Appendix: Unambiguity and compilers

Unambiguity and compilers

Compiling an expression (a stack for intermediate results + two registers):

- (1) $E \rightarrow E + T$... pop r1; pop r2; add r1,r2; push r2
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$... pop r1; pop r2; mul r1,r2; push r2
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow a$... push a

- 'a+a*a' is obtained by applying rules 1,2,4,6,3,4,6,6
- reverse the sequence and choose only code-generating rules:
6,6,3,6,1
- now replace the rules with the corresponding code:
push a; push a; pop r1; pop r2; mul r1,r2; push
r2; push a; pop r1; pop r2; add r1,r2; push r2