

Languages and Grammars

- Chapter 3, excluding 3.4 and 3.5

Formal Languages

- Basis for the design and implementation of programming languages
- **Alphabet**: finite set Σ of symbols
- **String**: finite sequence of symbols
 - Empty string ε
 - Σ^* - set of all strings over Σ (incl. ε)
 - Σ^+ - set of all non-empty strings over Σ
- **Language**: set of strings $L \subseteq \Sigma^*$

Grammars

- $G = (N, T, S, P)$
 - Finite set of **non-terminal symbols** N
 - Finite set of **terminal symbols** T
 - Starting non-terminal symbol $S \in N$
 - Finite set of **productions** P
- Production: $x \rightarrow y$
 - $x \in (N \cup T)^+$, $y \in (N \cup T)^*$
- Applying a production: $uxv \Rightarrow uyw$

Example: Non-negative Integers

- $N = \{ I, D \}$
- $T = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
- $S = I$
- $P = \{ I \rightarrow D,$
 $I \rightarrow DI,$
 $D \rightarrow 0,$
 $D \rightarrow 1, \dots \}$

More Common Notation

$I \rightarrow D \mid DI$

- two production alternatives

$D \rightarrow 0 \mid 1 \mid \dots \mid 9$

- ten production alternatives

Terminals: 0 ... 9

Starting non-terminal: I

- Examples of production applications:

- $I \Rightarrow DI$

- $D6I \Rightarrow D6D$

- $DI \Rightarrow DDI$

- $D6D \Rightarrow 36D$

- $DI \Rightarrow DDI$

- $36D \Rightarrow 361$

- $DDI \Rightarrow D6I$

Languages and Grammars

- String derivation
 - $w_1 \Rightarrow w_2 \Rightarrow \dots \Rightarrow w_n$; denoted $w_1 \xRightarrow{*} w_n$
- Language generated by a grammar
 - $L(G) = \{ w \in T^* \mid S \xRightarrow{*} w \}$
- Traditional classification of languages and grammars
 - Regular \subset Context-free \subset Context-sensitive \subset Unrestricted

Regular Languages

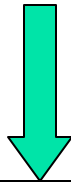
- Generated by **regular grammars**
 - All productions are $A \rightarrow wB$ and $A \rightarrow w$
 - $A, B \in N$ and $w \in T^*$
 - Or all productions are $A \rightarrow Bw$ and $A \rightarrow w$
- e.g. $L = \{ a^n b \mid n > 0 \}$ is a regular language
 - $S \rightarrow Ab$ and $A \rightarrow a \mid Aa$
- $I \rightarrow D \mid DI$ and $D \rightarrow 0 \mid 1 \mid \dots \mid 9$: is this a regular grammar?

Regular Grammars

- Alternative equivalent formalisms
 - Regular grammars
 - Regular expressions: e.g. a^*b for $\{ a^n b \mid n \geq 0 \}$
 - Deterministic finite automata (DFA)
 - Nondeterministic finite automata (NFA)
- Example: regular expression: $(a|b)^*abb$
 - Show a regular grammar
 - Show a non-deterministic finite automaton
 - Show a deterministic finite automaton

Lexical Analysis in Compilers

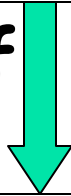
stream of
characters



w,h,i,l,e,(,a,1,5,>,b,b,),d,o,...

Lexical Analyzer (uses a regular grammar)

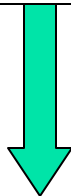
stream of
tokens



keyword[while], leftparen, id[a15], op[>],
id[bb], rightparen, keyword[do], ...

Parser (uses a context-free grammar)

parse
tree



each token is a leaf in the parse tree

... more compiler components

Uses of Regular Languages

- Lexical analysis in compilers
 - e.g., an identifier token is an element of the language **letter (letter|digit)***
 - each token is a **terminal symbol** for the context-free grammar of the parser
- Pattern matching
 - `stdsun> grep "a\+b" foo.txt`
 - Every line from `foo.txt` that contains a string from the language $L = \{ a^n b \mid n > 0 \}$
 - i.e. the language for reg. expr. a^+b

Context-Free Languages

- Subsume regular languages
 - Every regular language is a c.f. language
 - $L = \{ a^n b^n \mid n > 0 \}$ is c.f. but not regular
- Generated by a **context-free grammar**
 - Each production: $A \rightarrow w$
 - $A \in N, w \in (N \cup T)^*$
- BNF: alternative notation for context-free grammars
 - Backus-Naur form: John Backus and Peter Naur, for ALGOL60

Example: Non-negative Integers

- $I \rightarrow D \mid DI$ and $D \rightarrow 0 \mid 1 \mid \dots \mid 9$
- BNF
 - $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$
 - $\langle \text{digit} \rangle ::= 0 \mid 1 \mid \dots \mid 9$
- What if we wanted to disallow zeroes at the beginning?
 - e.g. 506 is OK, but 056 is not
 - Propose at least one BNF that achieves this

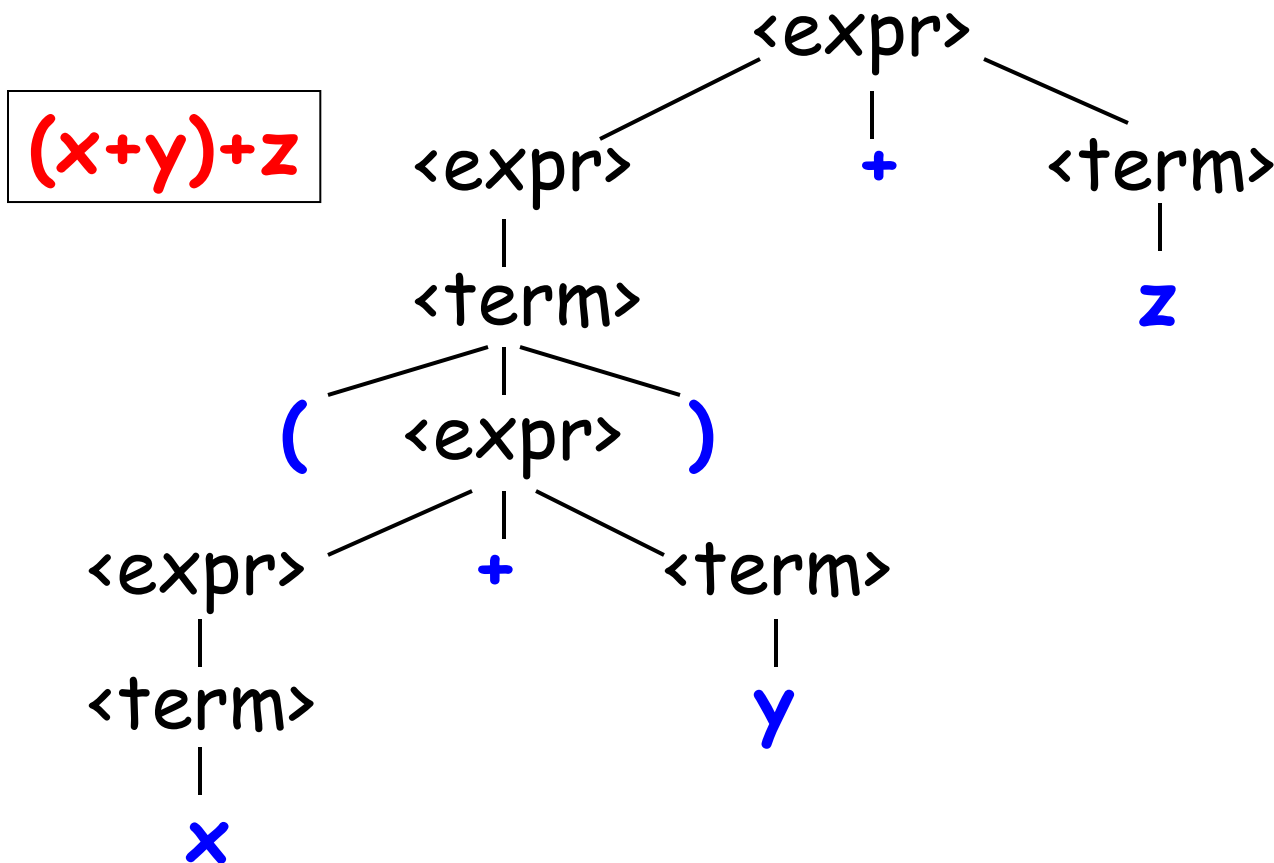
Derivation Tree for a String

- Also called **parse tree** or **concrete syntax tree**
 - Leaf nodes: terminals
 - Inner nodes: non-terminals
 - Root: starting non-terminal of the grammar
- Describes a particular way to derive a string based on a context-free grammar
 - Leaf nodes from left to right are the string
 - to get the string: depth-first traversal, following the leftmost unexplored branch

Example of a Derivation Tree

$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{expr} \rangle + \langle \text{term} \rangle$

$\langle \text{term} \rangle ::= x \mid y \mid z \mid (\langle \text{expr} \rangle)$



Equivalent Derivation Sequences

The set of string derivations that are represented by the same parse tree

One derivation:

$$\begin{aligned} \langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \Rightarrow \langle \text{expr} \rangle + z \Rightarrow \\ \langle \text{term} \rangle + z &\Rightarrow (\langle \text{expr} \rangle) + z \Rightarrow \\ (\langle \text{expr} \rangle + \langle \text{term} \rangle) + z &\Rightarrow (\langle \text{expr} \rangle + y) + z \Rightarrow \\ (\langle \text{term} \rangle + y) + z &\Rightarrow (x + y) + z \end{aligned}$$

Another derivation:

$$\begin{aligned} \langle \text{expr} \rangle &\Rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \Rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \Rightarrow \\ (\langle \text{expr} \rangle) + \langle \text{term} \rangle &\Rightarrow (\langle \text{expr} \rangle + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow \\ (\langle \text{term} \rangle + \langle \text{term} \rangle) + \langle \text{term} \rangle &\Rightarrow (x + \langle \text{term} \rangle) + \langle \text{term} \rangle \Rightarrow \\ (x + y) + \langle \text{term} \rangle &\Rightarrow (x + y) + z \end{aligned}$$

Many more ...

Ambiguous Grammars

- For some string, there are multiple different parse trees
- An ambiguous grammar gives more freedom to the compiler writer
 - e.g. for code optimizations
- For real-world programming languages, we typically have non-ambiguous grammars
 - To remove ambiguity: add non-terminals

Elimination of Ambiguity

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \mid$
 $(\langle \text{expr} \rangle) \mid \text{id}$

1. Prove that this grammar is ambiguous
2. Create an equivalent non-ambiguous grammar with the appropriate precedence and associativity
 - $*$ has higher precedence than $+$
 - both are left-associative

Example: what is the parse tree for
 $a + b * (c+d) * e$?

The “dangling-else” problem

- Ambiguity for “else”

$\langle \text{stmt} \rangle ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle$

| $\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- $\text{if } a \text{ then if } b \text{ then } c := 1 \text{ else } c := 2$
 - Two possible parse trees
- Traditional solution: match the **else** with the last unmatched **then**

Non-Ambiguous Grammar

$\langle \text{stmt} \rangle ::= \langle \text{matched} \rangle \mid \langle \text{unmatched} \rangle$

$\langle \text{matched} \rangle ::= \langle \text{non-if-stmt} \rangle \mid$

$\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{matched} \rangle$

$\langle \text{unmatched} \rangle ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \mid$

$\text{if } \langle \text{expr} \rangle \text{ then } \langle \text{matched} \rangle \text{ else } \langle \text{unmatched} \rangle$

What if we wanted to match the "else" with the first unmatched "then"? Change the grammar ...

Extended BNF (EBNF)

- [...] optional element
 - **if** <expr> **then** <stmt> [**else** <stmt>]
- { ... } repetition (0 or more times)
 - <id_list> ::= <id> { , <id> }
 - sometimes shown as { ... } *
- { ... }⁺ repetition (1 or more times)
 - <block> ::= **begin** <stmt> { <stmt> } **end**
 - <block> ::= **begin** { <stmt> }⁺ **end+**
- Do not use it in this class

Core: a toy imperative language

$\langle \text{prog} \rangle ::= \text{program } \langle \text{decl-seq} \rangle \text{ begin } \langle \text{stmt-seq} \rangle \text{ end}$

$\langle \text{decl-seq} \rangle ::= \langle \text{decl} \rangle \mid \langle \text{decl} \rangle \langle \text{decl-seq} \rangle$

$\langle \text{stmt-seq} \rangle ::= \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \langle \text{stmt-seq} \rangle$

$\langle \text{decl} \rangle ::= \text{int } \langle \text{id-list} \rangle ; \quad \langle \text{id-list} \rangle ::= \text{id} \mid \text{id} , \langle \text{id-list} \rangle$

$\langle \text{stmt} \rangle ::= \langle \text{assign} \rangle \mid \langle \text{if} \rangle \mid \langle \text{loop} \rangle \mid \langle \text{in} \rangle \mid \langle \text{out} \rangle$

$\langle \text{assign} \rangle ::= \text{id} := \langle \text{exp} \rangle ;$

$\langle \text{in} \rangle ::= \text{input } \langle \text{id-list} \rangle ; \quad \langle \text{out} \rangle ::= \text{output } \langle \text{id-list} \rangle ;$

$\langle \text{if} \rangle ::= \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt-seq} \rangle \text{ endif} ;$

$\mid \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt-seq} \rangle \text{ else } \langle \text{stmt-seq} \rangle \text{ endif} ;$

Core: a toy imperative language

$\langle \text{loop} \rangle ::= \text{while } \langle \text{cond} \rangle \text{ begin } \langle \text{stmt-seq} \rangle \text{ endwhile } ;$

$\langle \text{cond} \rangle ::= \langle \text{comp} \rangle \mid ! \langle \text{cond} \rangle \mid (\langle \text{cond} \rangle \text{ AND } \langle \text{cond} \rangle) \mid$
 $(\langle \text{cond} \rangle \text{ OR } \langle \text{cond} \rangle)$

$\langle \text{comp} \rangle ::= [\langle \text{operand} \rangle \langle \text{comp-op} \rangle \langle \text{operand} \rangle]$

$\langle \text{comp-op} \rangle ::= < \mid = \mid != \mid > \mid >= \mid <=$

$\langle \text{exp} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{exp} \rangle \mid \langle \text{term} \rangle - \langle \text{exp} \rangle$

$\langle \text{term} \rangle ::= \langle \text{operand} \rangle \mid \langle \text{operand} \rangle * \langle \text{term} \rangle$

$\langle \text{operand} \rangle ::= \text{const} \mid \text{id} \mid (\langle \text{exp} \rangle)$

Another Grammar

- **id** and **const** are terminal symbols for the grammar of the language
 - **tokens** that are provided from the lexical analysis to the parser
- But they are non-terminals for the regular grammar in the lexical analysis
 - The terminals now are characters
 - $\langle \text{id} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{id} \rangle \langle \text{letter} \rangle \mid \langle \text{id} \rangle \langle \text{digit} \rangle$
 - $\langle \text{letter} \rangle ::= \mathbf{A} \mid \mathbf{B} \mid \dots \mid \mathbf{Z} \mid \mathbf{a} \mid \mathbf{b} \mid \dots \mid \mathbf{z}$
 - $\langle \text{const} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{const} \rangle \langle \text{digit} \rangle$
 - $\langle \text{digit} \rangle ::= \mathbf{0} \mid \mathbf{1} \mid \dots \mid \mathbf{9}$

Problems with the Grammar

- Consider $9-5+4$
 - What is the parse tree?
 - What is the problem?
 - How do we fix it?
- It is not possible to write $x := -5;$
 - If the programmer wants this, she will have to write $0-5$
 - We could solve this problem, but we will not