Formal Languages and Grammars

Chapter 2: Sections 2.1 and 2.2
Formal Languages

• Basis for the design and implementation of programming languages

• **Alphabet**: finite set $\Sigma$ of symbols

• **String**: finite sequence of symbols
  – Empty string $\varepsilon$: sequence of length zero
  – $\Sigma^*$ - set of all strings over $\Sigma$ (incl. $\varepsilon$)
  – $\Sigma^+$ - set of all non-empty strings over $\Sigma$

• **Language**: set of strings $L \subseteq \Sigma^*$
  – E.g., for Java, $\Sigma$ is Unicode, a string is a program, and $L$ is defined by a grammar in the language spec
Formal 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 $P$
  – Describes a language $L \subseteq T^*$

• Production: $x \rightarrow y$
  – $x$ is a non-empty sequence of terminals and non-terminals; $y$ is a seq. of terminals and non-terminals

• Applying a production: $uxv \Rightarrow uyw$
Example: Non-negative Integers

• $N = \{ I, D \}$
• $T = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
• $S = I$
• $P = \{ \quad I \rightarrow D, \quad I \rightarrow DI, \quad D \rightarrow 0, \quad D \rightarrow 1, \quad ..., \quad D \rightarrow 9 \}$
More Common Notation

I → D | DI - two production alternatives

D → 0 | 1 | ... | 9 - ten production alternatives

• Terminals: 0 ... 9
• Starting non-terminal: I
  – Shown first in the list of productions
• Examples of production applications:
  I ⇒ DI
  DI ⇒ DDI
  DDI ⇒ D6I
  D6I ⇒ D6D
  D6D ⇒ 36D
  36D ⇒ 361
Languages and Grammars

• String derivation
  
  – $w_1 \Rightarrow w_2 \Rightarrow ... \Rightarrow w_n$; denoted $w_1 \Rightarrow^* w_n$
  
  – If $n>1$, non-empty derivation sequence: $w_1 \Rightarrow w_n$

• Language generated by a grammar
  
  – $L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}$

• Fundamental theoretical characterization: Chomsky hierarchy (Noam Chomsky, MIT)
  
  – Regular languages $\subset$ Context-free languages $\subset$
    Context-sensitive languages $\subset$ Unrestricted languages
  
  – Regular languages in PL: for **lexical analysis**
  
  – Context-free languages in PL: for **syntax analysis**
Regular Languages (1/5)

- **Operations on languages**
  - **Union**: $L \cup M = \text{all strings in } L \text{ or in } M$
  - **Concatenation**: $LM = \text{all } ab \text{ where } a \text{ in } L \text{ and } b \text{ in } M$
  - $L^0 = \{ \varepsilon \}$ and $L^i = L^{i-1}L$
  - **Closure**: $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$
  - **Positive closure**: $L^+ = L^1 \cup L^2 \cup \ldots$

- **Regular expressions**: notation to express languages constructed with the help of such operations
  - Example: $(0|1|2|3|4|5|6|7|8|9)^+$
Regular Languages (2/5)

• Given some alphabet, a regular expression is
  – The empty string $\varepsilon$
  – Any symbol from the alphabet
  – If $r$ and $s$ are regular expressions, so are $r | s$, $rs$, $r^*$, $r^+$, $r?$, and $(r)$
  – */+/? have higher precedence than concatenation, which has higher precedence than $|$
  – All are left-associative
Regular Languages (3/5)

• Each regular expression r defines a language L(r)
  – \( L(\varepsilon) = \{ \varepsilon \} \)
  – \( L(a) = \{ a \} \) for alphabet symbol a
  – \( L(r|s) = L(r) \cup L(s) \)
  – \( L(rs) = L(r)L(s) \)
  – \( L(r^*) = (L(r))^* \)
  – \( L(r^+) = (L(r))^+ \)
  – \( L(r?) = \{ \varepsilon \} \cup L(r) \)
  – \( L((r)) = L(r) \)

• Example: what is the language defined by
  \[ 0(x|X)(0|1|\ldots|9|a|b|\ldots|f|A|B|\ldots|F)^+ \]
Regular Languages (4/5)

• **Regular grammars**
  
  – All productions are $A \rightarrow wB$ and $A \rightarrow w$
    
    • $A$ and $B$ are non-terminals; $w$ is a sequence of terminals
    
    • This is a right-regular grammar
  
  – Or all productions are $A \rightarrow Bw$ and $A \rightarrow w$
    
    • Left-regular grammar

• Example: $L = \{ a^n b \mid n > 0 \}$ is a regular language
  
  – $S \rightarrow Ab$ and $A \rightarrow a \mid Aa$

• $I \rightarrow D \mid DI$ and $D \rightarrow 0 \mid 1 \mid \ldots \mid 9$: is this a regular grammar?
Regular Languages (5/5)

• Equivalent formalisms for regular languages
  – Regular grammars
  – Regular expressions
  – Nondeterministic finite automata (NFA)
  – Deterministic finite automata (DFA)
  – Additional details: Sections 2.2 and 2.4

• What does this have to do with PLs?
  – Foundation for **lexical analysis** done by a **scanner**
  – You will have to implement a scanner for your interpreter project; Section 2.2 provides useful guidelines
Regular Languages in Compilers & Interpreters

Stream of characters

Scanner (uses a regular grammar to perform lexical analysis)

Stream of tokens

Parser (uses a context-free grammar to perform syntax analysis)

Parse tree

Each token is a leaf in the parse tree

... more compiler/interpreter components
Uses of Regular Languages

• Lexical analysis in compilers
  – E.g., an identifier token is a string from the regular language $\text{letter (letter|digit)}^*$
  – Each token is a terminal symbol for the context-free grammar of the parser

• Pattern matching
  – stdlinux> `grep "a\+b" foo.txt`
  – Find 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

• They 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: \( A \rightarrow w \)
  – \( A \) is a non-terminal, \( w \) is a sequences of terminals and non-terminals

• BNF (Backus-Naur Form): traditional alternative notation for context-free grammars
  – John Backus and Peter Naur, for Algol-58 and Algol-60
    • Backus was also one of the creators of Fortran
  – Both are recipients of the ACM Turing Award
Example: Non-negative Integers

• $I \rightarrow D \mid DI$ and $D \rightarrow 0 \mid 1 \mid \ldots \mid 9$

• BNF
  – $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle\langle \text{integer} \rangle$
  – $\langle \text{digit} \rangle ::= 0 \mid 1 \mid \ldots \mid 9$

• What if we wanted to disallow zeroes at the beginning?
  – e.g. 509 is OK, but 059 is not
    • Possible motivation: in C, leading 0 means an octal constant
  – Propose a context-free grammar that achieves this
    • Is this grammar regular? If not, can you change it to make it regular?
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 this string: depth-first traversal of the tree, always visiting the leftmost unexplored branch
Example of a Derivation Tree

\[
<expr> ::= <term> | <expr> + <term>
\]

\[
<term> ::= id | (<expr>)
\]

Parse tree for \((x+y)+z\)
Equivalent Derivation Sequences

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

One derivation:

<expr> ⇒ <expr> + <term> ⇒ <expr> + z ⇒
<term> + z ⇒ (<expr>) + z ⇒
(<expr> + <term>) + z ⇒ (<expr> + y) + z ⇒
(<term> + y) + z ⇒ (x + y) + z

Another derivation:

<expr> ⇒ <expr> + <term> ⇒ <term> + <term> ⇒
(expr) + <term> ⇒ (<expr> + <term>) + <term> ⇒
(<term> + <term>) + <term> ⇒ (x + <term>) + <term> ⇒
(x + y) + <term> ⇒ (x + y) + z

Many more …
Ambiguous Grammars

• For some string, there are several different parse trees

• An ambiguous grammar gives more freedom to the compiler writer
  – e.g. for code optimizations, to choose the shape of the parse tree that leads to better performance

• For real-world programming languages, we typically have non-ambiguous grammars
  – We need a deterministic specification of the parser
  – To remove the ambiguity: add non-terminals
Elimination of Ambiguity (1/2)

- `<expr> ::= <expr> + <expr> | <expr> * <expr> | id | ( <expr> )`

- Two possible parse trees for `a + b * c`
  - Conceptually equivalent to `(a + b) * c` and `a + (b * c)`

- Operator **precedence**: when several operators are without parentheses, what is an operand of what?
  - Is `a+b` an operand of `*`, or is `b*c` an operand of `+`?

- Operator **associativity**: for several operators with the same precedence, left-to-right or right-to-left?
  - Is `a – b – c` equivalent to `(a – b) – c` or `a – (b – c)`?
Elimination of Ambiguity (2/2)

• In most languages, * has higher precedence than +, and both are left-associative

• Problem: change <expr> ::= <expr> + <expr> | <expr> * <expr> | id | ( <expr> )

  – Eliminate the ambiguity
  – Get the correct precedence and associativity

• Solution: add new non-terminals

  – <expr> ::= <expr> + <term> | <term>
  – <term> ::= <term> * <factor> | <factor>
  – <factor> ::= id | ( <expr> )
The “dangling-else” Problem

• Ambiguity for “else”

\[
\text{<stmt>} ::= \text{if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \\
\text{ | if} \ <\text{expr}> \ \text{then} \ <\text{stmt}> \ \text{else} \ <\text{stmt}>
\]

• \text{if} a \ \text{then if} b \ \text{then} c:=1 \ \text{else} c:=2
  – Two possible parse trees

• Traditional solution: match the \text{else} with the last unmatched \text{then}
Non-Ambiguous Grammar

\(<\text{stmt}\> ::= <\text{matched}\> \mid <\text{unmatched}\>

\(<\text{matched}\> ::= <\text{non-if-stmt}\> \mid

\text{if}\ <\text{expr}\> \text{ then } <\text{matched}\> \text{ else } <\text{matched}\>

\(<\text{unmatched}\> ::= \text{if}\ <\text{expr}\> \text{ then } <\text{stmt}\> \mid

\text{if}\ <\text{expr}\> \text{ then } <\text{matched}\> \text{ else } <\text{unmatched}\>
Extended BNF (EBNF)

• [ ... ] optional element
  – if <expr> then <stmt> [ else <stmt> ]

• { ... } repetition (0 or more times)
  – <IdList> ::= <id> { , <id> }
  – Sometimes shown as { ... }*

• { ... }+ repetition (1 or more times)
  – <block> ::= begin <stmt> { <stmt> } end
  – <block> ::= begin { <stmt> }+ end

• Does not change the expressive power of the notation (we can always rewrite in plain BNF)
Core: A Toy Imperative Language (1/2)

<prog> ::= program <decl-seq> begin <stmt-seq> end

<decl-seq> ::= <decl> | <decl><decl-seq>

<stmt-seq> ::= <stmt> | <stmt><stmt-seq>

<decl> ::= int <id-list> ;  <id-list> ::= id | id , <id-list>

<stmt> ::= <assign> | <if> | <loop> | <in> | <out>

<assign> ::= id := <expr> ;

<in> ::= input <id-list> ;  <out> ::= output <id-list> ;

<if> ::= if <cond> then <stmt-seq> endif ;

  | if <cond> then <stmt-seq> else <stmt-seq> endif ;
Core: A Toy Imperative Language (2/2)

```pascal
<loop> ::= while <cond> begin <stmt-seq> endwhile ;
<cond> ::= <cmpr> | ! <cond> | ( <cond> AND <cond> )
         | ( <cond> OR <cond> )
<cmpr> ::= [ <expr> <cmpr-op> <expr> ]
<cmpr-op> ::= < | = | != | > | >= | <=
<expr> ::= <term> | <term> + <expr> | <term> − <expr>
<term> ::= <factor> | <factor> * <term>
<factor> ::= const | id | − <factor> | ( <expr> )
```
Parser vs. Scanner

• **id** and **const** are terminal symbols for the grammar of the language
  – **tokens** that are provided from the scanner to the parser

• But they are non-terminals in the regular grammar for the lexical analysis
  – The terminals here are ASCII characters

  \[
  \begin{align*}
  \text{<id>} & ::= \text{<letter>} \mid \text{<id><letter>} \mid \text{<id><digit>} \\
  \text{<letter>} & ::= \text{A} \mid \text{B} \mid \ldots \mid \text{Z} \mid \text{a} \mid \text{b} \mid \ldots \mid \text{z} \\
  \text{<const>} & ::= \text{<digit>} \mid \text{<const><digit>} \\
  \text{<digit>} & ::= \text{0} \mid \text{1} \mid \ldots \mid \text{9}
  \end{align*}
  \]

Note: as written, this grammar is not regular, but can be easily changed to an equivalent regular grammar
Notes for the **Core** Interpreter Project

- Consider $9 - 5 + 4$
  - What is the parse tree? What is the problem?
  - For ease of implementation, we will not fix this
    - But if we wanted to fix it, how can we?

- Manually writing a scanner for this language
  - Ad hoc approach (next slide)
  - Systematic approach: write regular expressions for all tokens, convert to an NFA, convert that to a DFA, minimize it, write code that mimics the transitions of the DFA (Section 2.2)
    - There exist various tools to do this automatically, but you should **not** use them for the project (will use in CSE 5343)
Outline of a Scanner for Core (1/2)

• The parser asks the scanner for the next token
• Skip white spaces and read next character $x$
  • If $x$ is $; , ( ) [ ] = + − *$ return the corresponding token
  • If $x$ is $: ,$ read the next character $y$
    – If $y$ is not $=$, report error, else return the token for $:=$
  • If $x$ is $!$, peek at the next character $y$
    – If $y$ is not $=$, return the token for $!$
    – If $y$ is $=$, read it and return the token for $!=$
    – Peeking can be done easily in C, C++, and Java file I/O
Outline of a Scanner for **Core** (2/2)

- If \( x \) is <, peek at the next character \( y \)
  - If \( y \) is not =, return the token for <
  - If \( y \) is =, read it and return the token for <=

- Similarly when \( x \) is >

- If \( x \) is a letter, keep reading characters; stop before the first not-letter-or-digit character
  - If the string is a keyword, return the keyword token
  - Else return token **id** with the string name attached

- If \( x \) is a digit, keep reading characters; stop before the first not-digit character
  - Return token **const** with the integer value attached