CSE 307: Principles of Programming Languages

Syntax
1. Intro
2. Lexical Structure
   - Regular expressions
   - Finite-State Automata
3. Syntactic Structure
   - Grammars
   - Derivations
   - Ambiguity
   - Parse Trees
   - Using Grammars to Describe Syntax
Section 1

Intro
Syntax describes the structure of a program
  - Determines which programs are legal
  - Consists of two parts
    - Lexical structure: Structure of words
      Distinguish between words in the language from random strings
    - Grammar: How words are combined into programs
      Similar to how English grammar governs the structure of sentences in English

Programs following syntactic rules may or may not be semantically correct.
  - Compare with grammatically correct but nonsensical English sentences

Formal mechanisms used to describe syntax and semantics to ensure that a language specification is unambiguous and precise
Meta Languages

- Formal mechanisms are used to describe all allowable programs in a language
  - Backus-Naur Form
  - Grammars

- We need *languages to define languages* (called meta-languages)
  BNFs, Grammars etc. will be described in meta languages
Section 2

Lexical Structure
Lexical Structure

Constants and Literals: \((6.023e + 23, "Enter:\", etc.)\)

White space: Typically, blank, tab, or new line characters. Used to separate words, but otherwise ignored.

Special Symbols: “<”, “;”, etc. Can be used as separator, but not ignored.

Identifiers: \((x, \text{getChar, id\_f2})\)

Words with prespecified meaning: if, boolean, class.

- In some languages, these words could also be used as identifiers — in this case, they are called keywords as their use is not reserved.
Describing the Lexical Structure

**Regular Expressions are used as the meta language.**

- $(0 \mid 1 \mid \ldots \mid 9)^+$
  
  (describes non-negative integer constants)

- Short-hand notations are often used: e.g.,
  - $[0 - 9]^+$ (one more more occurrences of characters in range $[0 - 9]$)
  - `//.*` (two slashes followed by sequence of zero or more non-newline characters)
    
    (C++-style single-line comments)
Language of Regular Expressions

Notation to represent (potentially) infinite sets of strings over alphabet $\Sigma$. Let $R$ be the set of all regular expressions over $\Sigma$. Then,

**Empty String** : $\epsilon \in R$

**Unit Strings** : $\alpha \in \Sigma \Rightarrow \alpha \in R$

**Concatenation** : $r_1, r_2 \in R \Rightarrow r_1r_2 \in R$

**Alternative** : $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$

**Kleene Closure** : $r \in R \Rightarrow r^* \in R$
Regular Expression

\( a \) : stands for the set of strings \( \{a\} \)

\( a | b \) : stands for the set \( \{a, b\} \)
  - *Union* of sets corresponding to REs \( a \) and \( b \)

\( ab \) : stands for the set \( \{ab\} \)
  - Analogous to set *product* on REs for \( a \) and \( b \)
    - \((a|b)(a|b)\): stands for the set \( \{aa, ab, ba, bb\} \).

\( a^* \) : stands for the set \( \{\epsilon, a, aa, aaa, \ldots\} \) that contains all strings of zero or more \( a \)’s.
  - Analogous to *closure* of the product operation.
Regular Expression Examples

\((a|b)^*\) : Set of strings with zero or more a’s and zero or more b’s:

\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}

\((a^*b^*)\) : Set of strings with zero or more a’s and zero or more b’s such that all a’s occur before any b:

\{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, \ldots\}

\((a^*b^*)^*\) : Set of strings with zero or more a’s and zero or more b’s:

\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}
Semantics of Regular Expressions

**Semantic Function \( \mathcal{L} \):** Maps regular expressions to sets of strings.

\[
\begin{align*}
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(\alpha) &= \{\alpha\} \quad (\alpha \in \Sigma) \\
\mathcal{L}(r_1 | r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 \cdot r_2) &= \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{align*}
\]
Finite State Automata

Regular expressions are used for *specification*, while FSA are used for computation. FSAs are represented by a labeled directed graph.

- A finite set of *states* (vertices).
- *Transitions* between states (edges).
- *Labels* on transitions are drawn from $\Sigma \cup \{\epsilon\}$.
- One distinguished *start* state.
- One or more distinguished *final* states.
Consider the Regular Expression $(a \mid b)^*a(a \mid b)$.

$L((a \mid b)^*a(a \mid b)) = \{aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, \ldots\}$.

The following (non-deterministic) automaton determines whether an input string belongs to $L((a \mid b)^*a(a \mid b))$:
Determinism

\((a \mid b)^* a(a \mid b):\)

Nondeterministic:

\(\text{(NFA)}\)

Deterministic:

\(\text{(DFA)}\)
Lexical Analysis

- Regular expressions describing the lexical structure are converted into a finite-state machine
- This FSM can recognize words very quickly
  - algorithm linear in the size of input program
- Efficient FSMs generated automatically from RE-based definitions
- Lex was the first lexical-analyzer generator
  - Now superceded by Flex (and other similar tools)
Ambiguity Resolution

Consider a language with lexical definitions

\[ \text{Integer} ::= [0 - 9] + (i.e., [0 - 9][0 - 9]^*) \]

\[ \text{Identifier} ::= [a - z]^* ([a - z]|[0 - 9])^* \]

Consider the string “xx21”

- Is this to be treated as a single identifier,
- or as an identifier “xx” followed by an integer 21?

Need disambiguation rules

**Bad**: give priority to RE that occurs first in the language specification

**Better**: prefer longer matches to shorter ones
Section 3

Syntactic Structure
“How to combine words to form programs”

- Context-free grammars (CFG) and Backus-Naur form (BNF)
  - terminals
  - nonterminals
  - productions of the form nonterminal $\rightarrow$ sequence of terminals and nonterminals
- EBNF and syntax diagrams
Syntactic (phrase) structure

Context-Free Grammars:

\[ E \to E + E \]
\[ E \to E \ast E \]
\[ E \to \text{num} \]

- \( E \): Non-terminal symbol
- \( \text{num, +} \): Terminal symbol
- \( E \to \text{num} \): Grammar “rule” or production
- \( \mathcal{L}(E) \): set of strings that can be derived from \( E \) (Language of \( E \))
Grammars and Derivations

\[
\begin{align*}
\langle \text{sent} \rangle & \rightarrow \langle \text{np} \rangle \langle \text{vp} \rangle \\
\langle \text{np} \rangle & \rightarrow \langle \text{art} \rangle \langle \text{noun} \rangle \\
\langle \text{art} \rangle & \rightarrow \text{a} \mid \text{the} \\
\langle \text{noun} \rangle & \rightarrow \text{student} \mid \text{test} \\
\langle \text{vp} \rangle & \rightarrow \langle \text{verb} \rangle \langle \text{np} \rangle \\
\langle \text{verb} \rangle & \rightarrow \text{takes} \mid \text{ruins}
\end{align*}
\]

...
Ambiguity

\[ E \rightarrow E - E \]
\[ E \rightarrow \text{num} \]

\[
\begin{align*}
\text{num} & \quad \text{num} \quad \text{num} \\
E & - \quad E \quad \quad \text{num} \\
E & - \quad E \\
E & \quad \quad E \\
\hline
5 - 3 - 1 \equiv (5-3)-1
\end{align*}
\]

\[
\begin{align*}
\text{num} & \quad \text{num} \quad \quad \text{num} \\
\text{num} & \quad \quad E \quad \quad \quad E \\
\text{num} & \quad \quad E \\
E & \quad \quad E \\
\hline
5 - 3 - 1 \equiv 5-(3-1)
\end{align*}
\]
A Parse Tree succinctly captures the *structure* of a sentence.
Ambiguity (revisited)

A Grammar is **ambiguous** if there are *multiple parse trees* for the same sentence.

Example: $id + id + id$
Binary operators may be left-, right-, or non-associative.

Precedence specifies how tightly arguments are bound to an operator.

Associativity and precedence are specified to remove ambiguity.

A sampling of operators in C:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>-, +</td>
<td>left</td>
</tr>
<tr>
<td>*, /, %</td>
<td>left</td>
</tr>
</tbody>
</table>
## Parsing

Techniques to determine whether a sentence belongs to a language

- Parsing algorithms are more expensive than recognizers for regular languages.
- Grammar may need to be modified to accommodate parsing algorithms (Recursive descent, LALR, ...).
- Parsers typically build an *abstract syntax tree* which omits syntactic details and preserves the overall structure of a sentence.

  e.g.:

  Concrete Syntax: \( \langle s \rangle \rightarrow \text{while} \langle e \rangle \text{ do } \langle s \rangle \)

  Abstract Syntax: \( s \rightarrow \text{while}(e, s) \)

- Abstract syntax are “data types” in an interpreter/compiler.
Grammars in Practice

\[
\langle md \rangle \rightarrow \langle mod \rangle \langle type \rangle \langle id \rangle \ ( \langle params \rangle \ ) \ \langle block \rangle \\
\vdots
\]

\[
\langle params \rangle \rightarrow \langle param \rangle, \langle params \rangle
\]

\[
\langle params \rangle \rightarrow \langle param \rangle \\
\vdots
\]

\[
\langle block \rangle \rightarrow \{ \langle stmts \rangle \}
\]

\[
\langle stmts \rangle \rightarrow \langle stmt \rangle \langle stmts \rangle \\
\langle stmts \rangle \rightarrow \epsilon \\
\vdots
\]
Extended BNF: with “regular expression”-like operators to make grammars more concise.

- \{ A \}: zero or more occurrences of A.
- \[ A \]: zero or one occurrence of A.
- Additionally, we can write rules of the form

\[
\langle s \rangle \rightarrow \langle t_1 \rangle \ (a \ | \ \langle p \rangle \ ) \ \langle t_2 \rangle
\]

to represent two rules in BNF:

\[
\langle s \rangle \rightarrow \langle t_1 \rangle \ a \ \langle t_2 \rangle \\
\langle s \rangle \rightarrow \langle t_1 \rangle \ \langle p \rangle \ \langle t_2 \rangle
\]
EBNF (example)

\[ \langle md \rangle \rightarrow [ \langle mod \rangle ] \langle type \rangle \langle id \rangle \ ( \langle params \rangle \ ) \langle block \rangle \]

\[ \vdots \]

\[ \langle params \rangle \rightarrow \langle param \rangle \{, \langle param \rangle \} \]

\[ \langle params \rangle \rightarrow \langle param \rangle \]

\[ \vdots \]

\[ \langle block \rangle \rightarrow \{ \{ \langle stmt \rangle \} \} \]

\[ \vdots \]