Language Translation

if $E$ then $\implies \ldots$ code for $E$
else $S_1$ jmpz ,11
endif

$S_2$

W1: \ldots$ code for $S$
jmp ,12
\ldots$ code for $S$
jmp ,13
\ldots$ code for $S$
jmp ,13

Compilation

Translating a program in a high-level language (*source language*) to a program (in a *target language*) that can be readily executed,

- The source and the target programs must be *semantically equivalent*.
- That is, the compilation process must be *meaning preserving*.

Correctness of Translation

The compiler writer needs to:

- Prove that the algorithms used in translation process are meaning preserving, and
- Ensure that the implementation faithfully follows the algorithms.

*Syntax-directed translation* is a technique that simplifies the above tasks.

*Recursively translate each structure in the source language to a corresponding structure in the target language.*
Structure of a Language

**Grammars**: Notation to succinctly represent the structure of a language.

**Example:**

- **Stmt** → if Expr then Stmt else Stmt
- **Stmt** → while Expr do Stmt
- **Stmt** → do Stmt until Expr
- **Expr** → Expr + Expr

Grammars

- **Stmt** → if Expr then Stmt else Stmt

  - **Terminal symbols**: if, then, else
    - Terminal symbols represent group of characters in input language: Tokens
  - **Nonterminal symbols**: Stmt, Expr
    - Nonterminal symbols represent a sequence of terminal symbols.
    - Analogous to sentences.

Language Translation

- **do** → .D1:
  - **S** → .... code for S
  - **until** E → .... code for E
    - jmp .D1
    - jmpz .D2:

- **if** E1 then → .... code for E1
  - **do** → .... code for E1
  - **S1** → jmp .D1
  - **until** E2 → .... code for E2
  - **else** → .D1:
    - **S2** → .... code for E2
    - **endif** → .... code for E2
      - jmp .D2:
      - jmp .D1
      - jmp .D3:
      - jmp .D2:
      - jmp .D3:
      - jmp .D2:
      - jmp .D1
      - jmp .D3:
Language Processing

Flexible control: programmable combination of primitive operations,

- Express input to the system in a well defined *language*.
- Translate the input into the sequence of primitive operations,
  - Direct execution
  - Byte code emulation
  - Object code compilation

Language processing techniques have evolved over the last 30 years. In almost every domain, at least three steps can be identified: *lexical analysis, parsing, and syntax-directed translation*.

Lexical Analysis

Convert a stream of characters into a stream of tokens,

- Simplicity: Conventions about “words” are often different from conventions about “sentences”.
- Efficiency: Word identification problem has a much more efficient solution than sentence identification problem.
- Portability: Character set, special characters, device features,

Phases of Syntax Analysis

1. Identify the words: *Lexical Analysis*.
   - Converts a stream of characters [input program] into a stream of tokens.
   - Also called *Scanning* or *Tokenizing*.

2. Identify the sentences: *Parsing*.
   - Derive the structure of sentences: construct parse trees from a stream of tokens.

Applications of Languages

- Command Interpreters: *bash, perl,* ...
- Programming: *FORTRAN, SmallTalk,* ...
- Document Structuring: *troff, LaTeX, HTML,* ...
- Page Definition: *PostScript, PCL,* ...
- Databases: *SQL,* ...
- Hardware Design: *VHDL, VeriLog,* ...
- ... and many many more
Patterns

How do we *compactly* represent the set of all lexemes corresponding to a token?

For instance:

The token `integer_constant` represents the set of all integers:
that is, all sequences of digits (0-9), preceded by an optional
sign (+ or -).

Obviously, we cannot simply enumerate all lexemes.

Use Regular Expressions.

Regular Expressions

Notation to represent (potentially) infinite sets of strings over alphabet Σ,

- `a`: stands for the set {a} that contains a single string a.
- `ab`: stands for the set {ab} that contains a single string ab.
- `a | b`: stands for the set {a, b} that contains two strings a and b.
- `a*`: stands for the set {ε, a, aa, aaa, ...} that contains all
strings of zero or more a’s.

ε stands for the empty string.

Terminology

- **Token**: Name given to a family of words,
e.g., `integer_constant`
- **Lexeme**: Actual sequence of characters representing a word,
e.g., `32894`
- **Pattern**: Notation used to identify the set of lexemes
represented by a token,
e.g., `[0-9]+`

<table>
<thead>
<tr>
<th>Token</th>
<th>Sample Lexemes</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while</code></td>
<td><code>while</code></td>
<td><code>while</code></td>
</tr>
<tr>
<td><code>integer_constant</code></td>
<td><code>32894</code>, <code>-1003</code>, <code>0</code></td>
<td><code>[0-9]+</code></td>
</tr>
<tr>
<td><code>identifier</code></td>
<td><code>buffer_size</code></td>
<td><code>[a-zA-Z]+</code></td>
</tr>
</tbody>
</table>
Regular Definitions

Assign "names" to regular expressions.
For example,

\[
\begin{align*}
\text{digit} & \rightarrow 0 | 1 | \ldots | 9 \\
\text{natural} & \rightarrow \text{digit digit}^*
\end{align*}
\]

SHORTHANDS:

- \( a^+ \): Set of strings with one or more occurrences of \( a \).
- \( a^2 \): Set of strings with zero or one occurrences of \( a \).

Example:

\[
\text{integer} \rightarrow (\lt a\gt)^{\lt d\gt}\text{digit}^+
\]

Regular Expressions

Examples of Regular Expressions over \( \{a,b\} \):

- \( \{a,b\}^* \): 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:
  \( \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\} \)

Language of Regular Expressions

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: \( r_1, r_2 \in R \Rightarrow r_1 r_2 \in R \)
- Alternative: \( r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R \)
- Kleene Closure: \( r \in R \Rightarrow r^* \in R \)
Exercises

Read:
Chapter 1, 2 (scan and familiarize)
Sections 3.1, 3.3 (in depth)

Problems:
3.6, 3.7