CS 152  
Compiler Design

Rajiv Gupta

Compilers (more generally, Translators):

- **input**: an object in some language (the “source language”)

- **output**: an (“equivalent”) object in another (lower-level) language (the “target language”).

**Examples:**

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latex text</td>
<td>HTML text</td>
<td>latex2html</td>
</tr>
<tr>
<td>Postscript</td>
<td>HTML text</td>
<td>txt2html</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ps2text</td>
</tr>
<tr>
<td>Fortran</td>
<td>C</td>
<td>f2c</td>
</tr>
<tr>
<td>C++</td>
<td></td>
<td>cfront</td>
</tr>
<tr>
<td>C</td>
<td>asm/machine code</td>
<td>cc</td>
</tr>
<tr>
<td>Fortran</td>
<td>asm/machine code</td>
<td>f77</td>
</tr>
<tr>
<td>Alpha binary</td>
<td>asm</td>
<td>dis</td>
</tr>
</tbody>
</table>
Languages:

- **syntax**:
  - "structural" aspects of program units
  - specified by a grammar

- **semantics**
  - "meaning" (i.e., behavior) of program units
  - given by "semantic rules" ("actions")
Phases of a Compiler:

1. **lexical analyzer** (*scanner*) : input characters $\Rightarrow$ tokens
2. **syntax analyzer** (*parser*) : tokens $\Rightarrow$ syntax tree
3. **semantic analyzer** : type checking etc.
4. **intermediate code generator** : syntax tree $\Rightarrow$ intermediate code
5. **code optimizer** : improvements to intermediate code
6. **final code generator** : intermediate code $\Rightarrow$ executable/assembly code

(Also: symbol table manager, error handler)

---

**Lexical Analysis** ("scanning")

- analogous to finding words and punctuation in English text.

- Functions of a lexical analyzer:
  - read source code (a stream of characters)
  - return a stream of tokens, e.g.:

  ```
  punctuation  (  $\Rightarrow$ lparen
  operators     $\geq$  $\Rightarrow$ op_GE
  constants    65   $\Rightarrow$ intcon(65)
                 6.5  $\Rightarrow$ floatcon(6.5)
  identifiers  count $\Rightarrow$ id("count")
  keywords      while $\Rightarrow$ WHILE
  ```
**Syntax Analysis** ("parsing")

- reads tokens from lexical analyzer
- parses token stream according to the grammar of the language
  (analogous to constructing sentences from words and punctuation)
- reports syntax errors to the user
- produces a “parse tree” or “syntax tree” that represents the syntactic structure of the program.

**Semantic Analysis**

- checks semantic rules (types, declarations, etc.)
- gathers semantic information, e.g. type information, for subsequent code generation.

**Intermediate Code Generation**

- takes the internal representation of the program (from syntax analysis) + type information etc., and generates “low level” code that is still machine independent.
Code Optimization

- Improves speed/size of a (intermediate code) program by analyzing the code, e.g.:
  - elimination of redundant instructions
  - replacing expensive operations by cheaper ones
  - reducing the frequency of execution of some instructions

Final Code Generation

- generates assembly/object code for the machine at hand.

- machine-specific decisions:
  - machine-dependent instruction selection
  - register allocation
Phases of a Compiler

Grouping of Phases:

- **Front end**: machine independent phases:
  - lexical analysis
  - syntax analysis
  - semantic analysis
  - intermediate code generation
  - some code optimization

- **Back end**: machine dependent phases:
  - some code optimization
  - final code generation