Introduction to Languages

Abstraction

- Programming language (PL) gives an *unambiguous* notation for expressing algorithms
  - Do NOT want ambiguity
- PL source is processed by PL Processors
- PLs differ by level of abstraction
- Abstraction requires translation, often between levels of abstraction
- Editors, frameworks, etc. are still higher than HLL

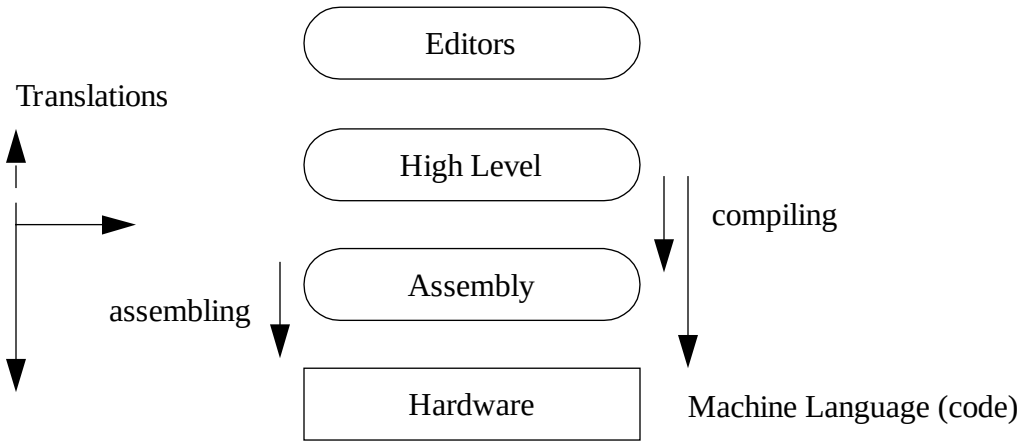

Figure 1. Abstraction and resulting translation

- Historical developments in abstractions/translations
  - machine language (ML)
    - machine dependent
    - binary, hard to use
    - fixed addresses
  - symbolic assembly
    - one-to-one correspondence to ML except for extensions
    - symbolic addresses with storage allocation directives
    - machine dependent
    - extensions: macro processing, functions, modular assembling
  - high level language (HLL)
    - complex expressions
    - data typing (built-in and/or user-defined)
    - complex control structures
    - procedural and data abstraction
    - machine independent (portability)

- some system level dependencies prevent 100% portability

- recursion

Programming Language Processors

- Generating translators
  - source -> target
  - file, persists, can be distributed independently but only on the same target machine
  - compiler: generating translator HLL-> lower level (assembly or ML)
- Interpreters
  - source -> execution
  - every execution needs the translator
  - execution must be time/sliced with translator

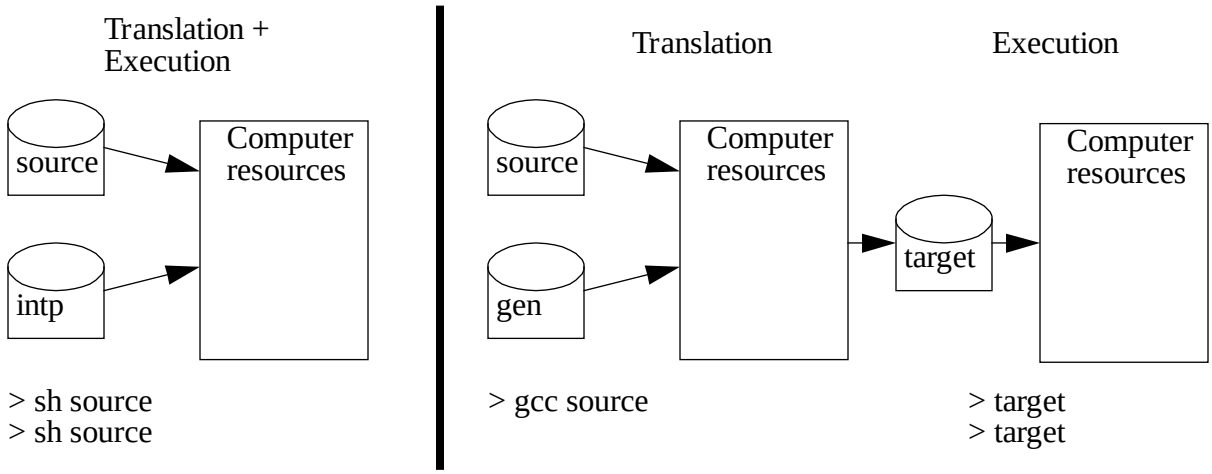

Figure 2. Interpretation (left) vs. generation (right)

Attribute	**Interpreter**	**Generator**
Target	no explicit target, just execution	file
Translation speed		
Translation complexity		
Translation memory req.		
Execution speed, memory		

- Hardware is interpreter its own machine code
    - source (machine code) -> target (actions)
    - hardwired vs. micro-coded
- Translation passes
    - Pass is one read over the input
    - Translators need a number of *passes*
        - Some passes can be on the original source, some on some intermediate representation
        - More passes lead to more portable/reconfigurable compilers (cheaper)
        - Fewer passes lead to faster compilers (more expensive) often losing re-configurability

Programming Language Specification

- PL is defined by
    - *Syntax*
        - What are *lexical* elements and how to put them together
    - *Semantics* (static and execution semantics)
        - What is the meaning of instructions
            - *Static* semantics are constraints/rules (*contextual constraints*) that can be processed by the translator which could not be expressed by syntax
            - *Execution* (dynamic) semantics

*S*yntax

- alphabet ∑ - set of valid characters
- grammar – set of rules for generating/recognizing valid strings of the language
        - grammar is defined on a different alphabet (tokens T)

Assume binary alphabet ∑={0, 1}.

- The set of all strings over the alphabet is ∑*.
    - For example, 0, 011, 101111, as well as the empty string ε are all strings generated from this alphabet.
- However, a given language L will have grammar rules/restrictions allowing only a certain subset of ∑* to be valid.
    - For example, L_1 = 01*will have valid strings such as 0, 0111, but 10 would not be valid

The C programming language

- has alphabet containing letters, digits, symbols such as %, *, &, but not symbols such as $ or @
- has about a hundred classes of tokens such as
  - if, while, void keywords
  - operators such as +, ++
  - delimiters such as , (
  - identifiers such as x
  - numbers such as 5, 2.3
- grammar says how the tokens can be arranged
  - e.g., statement must end with ;

Semantics

- The meaning of expressions and statements
  - denotational vs. operational semantics
  - informal vs. formal (or hybrid)

The semantics of an assignment in C can be for example explained informally using operational semantics

- evaluate the right hand side
- assign the resulting value into lefty hand side, performing any necessary type coercion

Classification of Languages

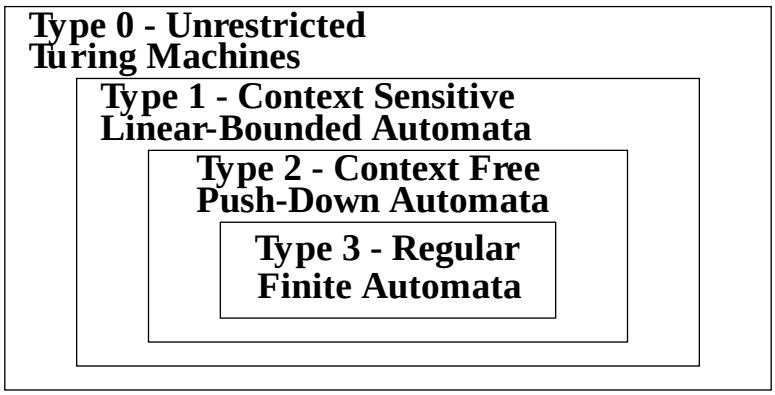

Figure 3. Chomsky classification of languages and their algorithms/processors

- Programming languages are not context-free (CF)
- Context-free are the most that can be effectively processed
- CF languages are defined with context-free grammar (CFG)
- PL are divided into CF and non-CF components
- PL is defined with grammar and semantics on elements called *tokens*
  - tokens are defined by regular language, with so called *lexical* definitions

    - for example, numbers starting with x are hex numbers, identifiers must start with letter, etc.
    - tokens are extracted using Finite Automaton (FA, FSM, FSA, etc.)
    - syntax is defined in terms of tokens
- Grammar (CFG) is processed by PushDown Automaton (PDA)
    - PDA is augmented with procedures to deal with static semantics (*semantic routines*)
        - variable binding
        - arguments types and number
        - compatibility rules etc.
    - CFG is expressed using Backus-Naur Form (BNF)
        - CFG can be augmented with static semantics information

BNF

- See 4250 textbook
- Standard BNF has *terminals* (tokens) and *non-terminals*
    - Terminals and nonterminals must be differentiated by notation such as token vs. <nonterminal> or a vs. A
    - One <nontermimal> is designated as the initial
    - BNF is a set of rules/productions of the form
      <nonterminal> -> any combination of terminals and <nonterminals>
- Extended BNF
    - Uses extra symbols to simplify the productions
        - {x} means repeat x any number of times
        - [x]means optional x
        - Etc.
    - The extended extra symbols must be clearly differentiated from the same symbols in terminals

      <program> -> program begin <variables> <statements> end .
      <variables> -> var id ; < variables> | ε
      <statements> -> <stat> ; {<stat>;}
      ...
      <ifStat> -> if <cond> <stat> [else <stat>]
      ...

Bindings

- See 4250 textbook
- Assigning property values to something
    - Assigning value to variable
    - Assigning address to variable
    - Assigning type to variable
- Static vs dynamic binding

- Static – translation time
- Dynamic – execution time
- Example from 4250 – static scoping vs. dynamic scoping

Assume function

```
int max(int, int);
```

then what bindings we have in this fragment:

```
int x;
x = 3;
x = max(3.5, 2+3);
```