Programming Languages

• As difficult to discuss rationally as religion or politics.
• Prone to extreme statements devoid of data.

Examples:

"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration." (Dijkstra)

"The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence." (Dijkstra)

• Like anything else, decision making should be a rational process based on the priorities and features of the project.
Some Decision Factors

• Features of application:
  Hard real time?
    Not just efficiency
    Predictability (need to guarantee deadlines will be met)
  High assurance?
  Portability?
  Maintainability?
  Others?

• Features of development environment:
  Availability of programmers, compilers, development tools?
  Schedule constraints?
  Others?
Relationship between PL and Correctness

- Decreasing emphasis as an explicit design goal and research topic.

- Masterability
  - Not complex: programmers understand it in its entirety
    The most important decisions in language design concern what is to be left out."  (Wirth)
  - Powerful features OK only if easy to use correctly.
    - Balance against need to keep language simple
  - "Natural"
    - Language should not surprise us in any of its effects.
    - Should correspond to our experience with natural languages, mathematics, and other PLs
Relationship between PL and Correctness (2)

- Error Proneness
  - Language design should prevent errors.
    Should be difficult or impossible to write an incorrect program.
  - If not possible, then allow their detection (as early as possible)
  - Need for general principles and hypotheses so can predict
    error-prone features and improve language design.
  - Some hypotheses and data about:
    Go to
    Global variables
    Pointers
    Selection by position (long parameter lists)
    Defaults and implicit type conversion
    Attempts to interpret intentions or fix errors
  - Meaning of features should be precisely defined (not
    dependent on compiler.)
Relationship between PL and Correctness (3)

• Understandability
  - "The primary goal of a programming language is accurate communication among humans."
  - Readability more important than writeability.
    • Well "punctuated" (easy to directly determine statement types and major subunits without intermediate inferences)
    • Use of distinct structural words (keywords, reserved words) for distinct concepts (no overloading, e.g., = for equal, assignment)
    • Avoidance of multiple use of symbols unless serve completely analogous functions (e.g., commas as separators, parentheses for grouping).
  - Necessary to be able to see what is being accomplished at a higher level of abstraction.
    • Permit programmers to state their "intentions" along with instructions necessary to carry them out.
Relationship between PL and Correctness (4)

• Maintainability
  – Locality -- possible to isolate changes.

  – Self-documenting
    • Programming decisions should be recorded in program, independent of external documentation.
    • Good comment convention, freedom to choose meaningful variable names, etc.
    • User-defined types and named constants
      e.g., type direction=(north, south, east, west)

• Explicit interfaces
  • Should cater to construction of hierarchies of modules
Relationship between PL and Correctness (5)

• Checkability
  – Every error should transform a correct program into one whose errors are detectable by the system.
  – All error detection based on redundancy (but some forms can cause errors).
    Examples of useful redundancy:
    - type declarations and type checking
    - declarative redundancy
    - invariance conditions or assertions
  – Run-time assertions, exception handling
    checking subscripts vs. array bounds
    case selector vs. case bounds
Relationship between PL and Correctness (6)

• General
  - High-level languages take many decisions out of programmer’s hands.
    • One reason they are so fiercely resented by experienced programmers.
    • Language should restrict programmer to decisions that really matter.
    • Decisions should be recorded in program independent of external documentation.
  - Simplicity of language less important than ability to write conceptually simple programs.
Green: Design and Use of PLs

"Clarifying the psychological processes of using programming languages will, I believe, clarify the requirements of language design and of environmental support."

Some examples of structured programming hypotheses:

- Control structures should be hierarchical. Thus they should be nested, rather than allowed to have arbitrary branching. In this way, successive layers of detail can be added.

- The comprehensibility of hierarchically constructed programs will be easier, since they can be understood by a reverse process -- understand the outer layer, then the inner layers, etc.

- These programs will be easy to modify because the inter-relations between parts will be simple.

Have accepted these hypotheses but have never been validated.
Green: Program Comprehension

• Cites experiments ("atheoretical" ) that evaluate only current programming practice.

• More interesting question: Can we elucidate underlying psychological principles to allow generalization of results to other classes of information structure in programming?
  
  – Hypothesis 1: If one language is better than another, it is always better, whatever the context.

  – Hypothesis 2: Every notation highlights some type of information at the expense of others; the better notation for a given task is the one that highlights the information that given task needs.

• More generally, the comprehensibility of a notation may depend on the number and complexity of mental operations required to extract needed information.
Green: Program Comprehension (2)

- Cites results supporting second hypothesis better than first. Not predicted by arguments of structured programming, which are based solely on presence or absence of good structure.

  - Programmers were not simply decoding programming structure top down into some undescribed mental representation. Were reworking one structure into another.

  - Difficulty of answering questions depended not only on source structure but also on relation between source and target structures.

- Observes that result "appears to raise insuperable difficulties for those simple-minded computer scientists who attempt to measure ‘psychological complexity’ of a program by means of a single number, such as McCabe."
Statement of Problem:

Fry: everything that is juicy but not hard
Boil: everything that is hard
Chop and roast: everything that is neither hard nor juicy

<table>
<thead>
<tr>
<th>if hard go to L1</th>
<th>if hard then begin boil end else begin if juicy then begin fry end else begin chop roast end</th>
<th>if hard: boil not hard: if juicy: fry not juicy: chop roast end juicy end hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>if juicy go to L2</td>
<td>chop roast stop</td>
<td></td>
</tr>
<tr>
<td>L2: fry stop</td>
<td>else</td>
<td>end</td>
</tr>
<tr>
<td>L1: boil stop</td>
<td>begin</td>
<td></td>
</tr>
</tbody>
</table>

Using Dijkstra’s guarded command:

if hard: boil
if not hard, juicy: fry
if not hard, not juicy: chop roast
Green: Program Creation

- Programs as plans.

- Role expressiveness: Outcome of a programmer’s effort is a structure in which each part plays some role vis-a-vis the programmer’s original intention.
  - Easy program comprehension and creation requires role expressiveness.
    - Rapid chunking into components
    - Visible or easily inferred purposes for each part
    - Visible or easily inferred relationships between each part and the larger structure.
  - Important to alleviate mismatches between programmer’s task and program structure.
  - Hypothesis that role expressiveness tends to detract from reusability of program fragments.
    "When a program fragment makes its role and purpose very clear, it is probably not easy to transport it unchanged to a new environment, because its role may be slightly but significantly different."
Green: Program Creation (2)

• Linguistic consistency
  – Metarules vs. BNF (simplified syntax rules)
    Not the number of rules that counts but the consistency between the form of the rules.
  – Similar roles should be indicated by similar syntax: syntax should map roles consistently.

• Significant omissions (defaults)

• Perceptual cues
  – Humans not good at discerning structure of a string of arbitrary symbols but good at differentiating shapes, spatial positions, and other perceptual cues.
  – Dangling else: use of indentation the best solution
    Green asks: "Why did it take so long to find the solution?"
Green: Other Topics

- Choice of programming paradigm (procedural, OO, functional, ...)
  Few studies comparing programming paradigms

- Programming medium
  Will new mediums affect design of languages and development of programs?

- Comprehension aid and documentation
  - Different aids for different language types, different purposes
  - Doxy: User definable browsers and filters
    Supply particular views of program
    Hide complexity irrelevant to user’s purpose.