CSE583:
Programming Languages

David Notkin
4 January 2000
notkin@cs.washington.edu
http://www.cs.washington.edu/education/courses/583

Central focus

- Study of major concepts in programming languages
- A particular focus on non-standard languages, concepts and constructs
- Not especially
  - Implementation oriented or theoretically oriented, although we'll necessarily touch on both

Tonight

- A little administrivia
  - Ask questions now or at the end
- Basic intentions for the class
  - Why study languages, importance of languages, a bit of language history, etc.
- Language design principles
- A whirlwind tour of types and some non-standard programming paradigms

Distance learning

- You need to be involved in the lecture
  - I'll try to help
- *Let us know if there are technical problems*
  - Then and there, and on an ongoing basis

Administrivia: see web page

- Readings
- Assignments (written and programming)
- Mini term papers
- Take home final
- Some pair work permitted (not required)

- TA: Adam Carlson

My prejudices

- Programming languages are a key part of developing better software
- I'm a software engineering researcher
  - There are many other factors that contribute to quality software
  - (Aside: my knowledge of programming languages is somewhat limited)

Two most important benefits

- Higher-level languages give productivity improvements
  - The lines of code you can produce is roughly independent of the programming language
  - It is not clear whether the quality remains the same
- Explicit interfaces
  - The structure of a program is at least as important as the way the computation is written

Sapir-Whorf hypothesis

- Programming languages do influence the way we write software
- Hypothesis: the language we have influences how we think (as well as how we communicate what we think)
  - Hypothesized with respect to natural language, not programming language, but plausible nonetheless
  - "First language" theory

What was your first language?

- What programming language did you learn first
  - Use your own definition of "learn" and "first"
- At each site, the person who was born closest to UW should gather the information

Flon's Law

- A good programmer will program well in any language, and a bad programmer will program poorly in any language

Why study programming languages?

- I'm not likely to convince you to stop using YFPL and start using MFPL, my new distributed, concurrent, web-based, object-oriented, interactive, enterprise, constraint-based, rule-oriented, parallel, Y2K-compliant, heterogeneous, 128-bit, buzzword-based language
- Your Favorite Programming Language

Why study programming languages?

- You are probably not going to try to write a new programming language intended to replace C++, Java or any other prevalent programming language
- If you are going to try, good luck!

So why?

- Some of the stuff is very, very cool
- It may help you better use the language(s) you currently use
- It may possibly help you select a language for a new project
- It may help you design “little”, domain-specific, languages better

A Partially Correct History of Programming Languages

- *[Edited without permission from Dartmouth’s CS68 97W]*
- Konrad Zuse's Plankalkul (1945) was perhaps the first language designed for expressing computation on a computer; but never implemented
- FORTRAN (FORmula TRANslator) was the first high-level language implemented, with an emphasis on efficiency of compiled code; design and implementation team led by John Backus (1954-57)
- LISP (LISt Processor) was a language designed for symbolic processing (mostly for AI users); McCarthy at MIT (1958); introduced symbolic computation and automatic memory management

More…

- ALGOL-60 (ALGOrithmic Language) designed for clearly expressing algorithms both to the computer and in computer science literature; first report on issued in 1958, with subsequent meetings in 1959 and 1960 revised the specification; primary ancestor of Pascal and C, introduingd block structure, compound statements, recursive procedure calls, nested if, loops, arbitrary length identifiers
- COBOL (COmmon Business-Oriented Language) designed around 1960 for business applications, pioneering sophisticated record structures; designed to be readable by managers, so has a strong English-like flavor

More…

- BASIC (Dartmouth) was perhaps the first language designed for time-sharing systems
- PL/I was IBM's attempt to tie together concepts from FORTRAN, ALGOL, COBOL (and a little LISP) and to add more features; introduced concurrency and exceptions
- Simula/67 that introduced objects and inheritance
- Pascal (Wirth, 1971), a ALGOL-like language with a deep understanding of implementation issues
- C (1972), designed in part for portable OS design

More…

- Prolog designed and implemented in early 1970s
- ML late 1970s; as broadly used as any functional programming language
- Smalltalk late 1980s; uniformly object-oriented language
- Ada in early 1980s, intended to be a uniform language for government applications
- C++ in mid 1980's
- Java developed by Sun in early 1990s

Sammet’s view

- Over 200 programming languages were developed between 1952 and 1972, but only about 13 were significant

Another chronology of influential languages

- 1957 FORTRAN
- 1958 ALGOL
- 1960 LISP
- 1960 COBOL
- 1962 APL
- 1962 SIMULA
- 1964 BASIC
- 1964 PL/I
- 1966 ISWIM
- 1970 Prolog
- 1972 C
- 1975 Pascal
- 1975 Scheme
- 1977 OPS5
- 1978 CSP
- 1978 FP
- 1980 dBASE II
- 1983 Smalltalk-80
- 1983 Ada
- 1983 Parlog
- 1984 Standard ML
- 1986 C++
- 1986 CLP(R)
- 1986 Eiffel
- 1988 CLOS
- 1988 Mathematica
- 1988 Oberon
- 1990 Haskell

What is your primary programming language?

- What programming language do you use most regularly and extensively?
- At each site, the person who was born closest to New York City should gather the information

Tradeoff between expressiveness and performance

- Perlis epigram #54: “Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.”
  - http://www.cs.yale.edu/~perlis-alan/quotes.html
- We’re not talking about computability in this class
  - In principle, you can (somehow or another) write any program you care about in any programming language
- We’re talking about effectiveness

Basic language design principles

- Many of the following are from B.J. MacLennan
  - *Principles of Programming Languages: Design, Evaluation and Implementation*
- Two of the assigned papers (by Wirth and by Hoare) are on programming language design principles
- I’ll give a few examples; you can fill in others from your experience

Abstraction

- Avoid requiring something to be stated more than once; factor out the recurring pattern
  - Procedures and functions, modules and classes, macros, etc.
- FORTRAN I (in the 1950s) did not include subprograms in its preliminary description
  - Libraries (for I/O and math) but not user-defined subprograms
- The original and central use of subprograms was to save memory

Automation

- Automate mechanical, tedious, or error-prone activities
- (This rule is at least as important for tools as for languages.)
- High-level language loops are a great example
- Parameter passing is another

Labeling

- Avoid arbitrary sequences more than a few items long
- Do not require the user to know the absolute position of an item in a list
  - Instead, associate a meaningful label with each item, allowing to occur in any order
- case statements vs. computed gotos
- property lists (p-lists) in Lisp

  -
    ```
    (age 45 ssn 123456789
     name (Smokey Bear)
     office (Sieg 123))
    ```

Parameter labeling

-
  ```
  call stokes ( a, bc_tag, detmap, eqn, g, ierror,
  indx, ipivot, jac, maxelm, maxeqn, maxnp,
  maxquad1, maxquad2, maxside, nelem, neqn, nlband,
  node, np, nquad1, nquad2, nrow, nside, penalty1,
  penalty2, phi, region, region_ymax, res, res2,
  reynold, side_basis, side_elem, side_eqn,
  side_etam, side_etap, side_indx, side_xsim,
  side_xsip, squad1, wquad1, wquad2, xc, yc )
  ```
- 45 parameters in this FORTRAN call
  - Taken from John Burkhart's web page at the Pittsburgh Supercomputing Center
  - http://www.psc.edu/~burkardt/flow6.html
- I've heard tell of Cobol programs with 100s of parameters

Parameter labeling in Ada

- Ada 83 permits position-independent parameters (and default values)

  -
    ```
    procedure DRAW_AXES(X_ORIGIN,Y_ORIGIN:COORD:=0;
                        X_SCALE,Y_SCALE:REAL:=1.0;
                        X_SPACING,Y_SPACING:NATURAL:=1;
                        X_LOG,Y_LOG:BOOLEAN:=FALSE;
                        FULL_GRID:BOOLEAN:=FALSE);
    ```

  -
    ```
    DRAW_AXES(500,500,Y_SCALE=>0.5,Y_LOG:=TRUE,
              X_SPACING=>10,Y_SPACING=>10);
    ```
- Complicates overloading

  -
    ```
    procedure P(X:INTEGER;Y:BOOLEAN:=FALSE);
    procedure P(X:INTEGER;Y:INTEGER:=0);
    P(3);
    ```

Defense in Depth

- Have a series of defenses so that if an error isn't caught by one, it will probably be caught by another

-
  ```
  DO 20 I = 1.100
  DO 20 I = 1,100
  ```
- Boom, there goes the Venus probe!
  - Apparently urban legend, but still!
- Interaction of implicit declarations and ignoring of blanks as lexical units

Localized cost

- Users should only pay for what they use; avoid distributed costs
- (I prefer to call this "manifest cost", where all costs should be apparent)
- Algol 60 for-loops reevaluated its loop parameters on each iteration
  - In the absence of a smart compiler, even simple loops became very costly

Orthogonality

- Independent functions should be controlled by independent mechanisms
- Single statements vs. blocks in FORTRAN or C

  -
    ```
    if c then S
    if c then {S1;S2}
    ```
  - Complicates change
- Inheritance for sharing code vs. inheritance for sharing behaviors

Regularity

- Regular rules, with exceptions, are easier to learn, use, describe, and implement
- In Smalltalk-80, everything is an object
  - Integers, points, user-defined objects, even the class definitions themselves
  - So you manipulate everything the same way

Security

- No program that violates the definition of the language, or its own intended structure, should escape detection
- Pascal’s type hole for non-discriminated union types
- Ada 83 and non-compliant programs

Simplicity

- A language should be as simple as possible
- There should be a minimum number of concepts with simple rules for their combination
- C++, need I say more?
- Smalltalk-80
- Functional languages
  - Function (λ) definition
  - Function application

Structure

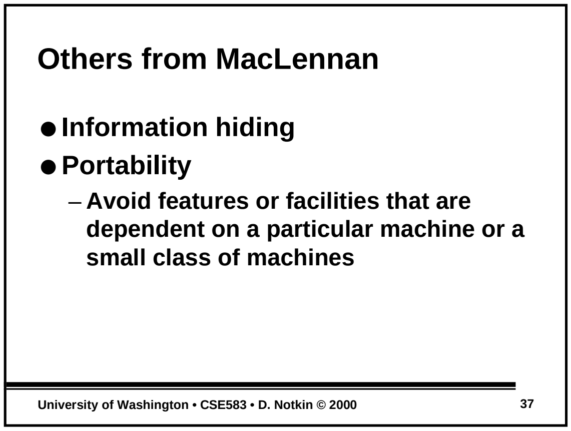
- The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations
- gotos
- Dynamic scoping

Syntactic consistency

- Similar things should look similar
- Different things should look different
- computed gotos vs. assigned gotos
  - `GOTO (L1,L2…),I` `GOTO N,(L1,L2,…)`
  - goto L_I vs. branch to statement whose address is in `N`
    - the list is documentation

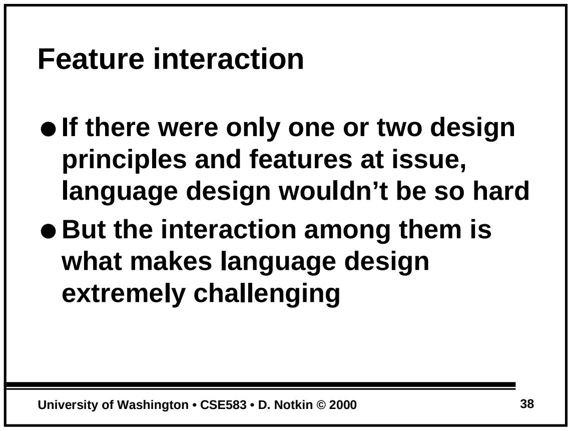
Zero-one-infinity

- The only reasonable numbers are zero, one, and infinity
- Six character identifiers



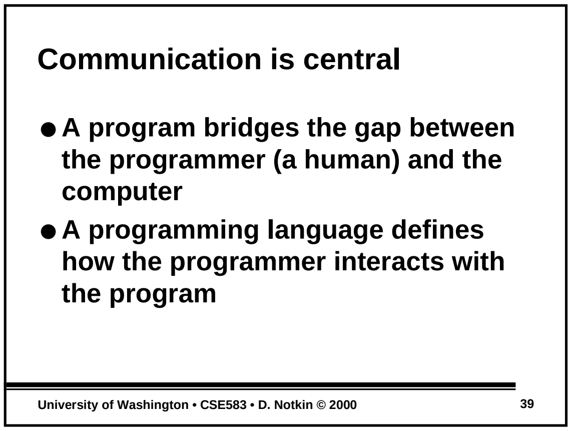
Others from MacLennan

- Information hiding
- Portability
  - Avoid features or facilities that are dependent on a particular machine or a small class of machines



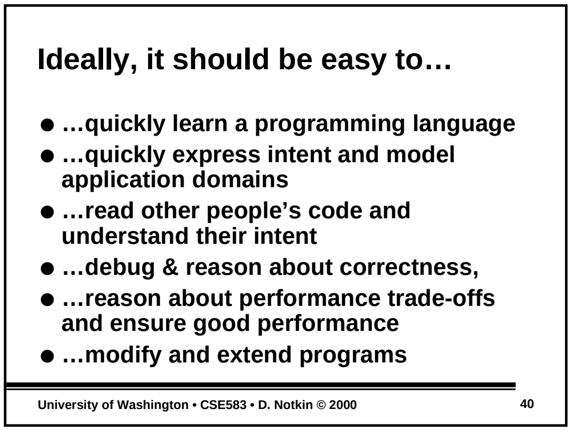
Feature interaction

- If there were only one or two design principles and features at issue, language design wouldn't be so hard
- But the interaction among them is what makes language design extremely challenging



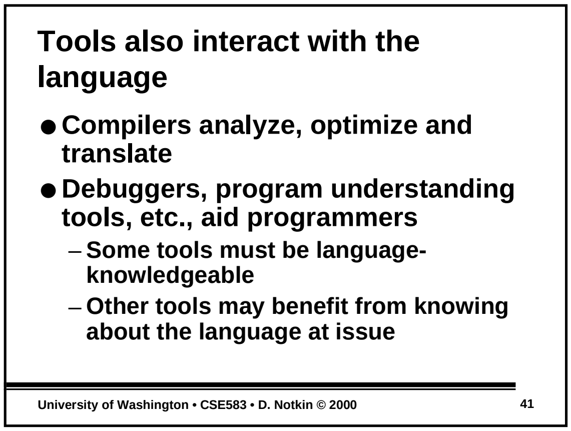
Communication is central

- A program bridges the gap between the programmer (a human) and the computer
- A programming language defines how the programmer interacts with the program



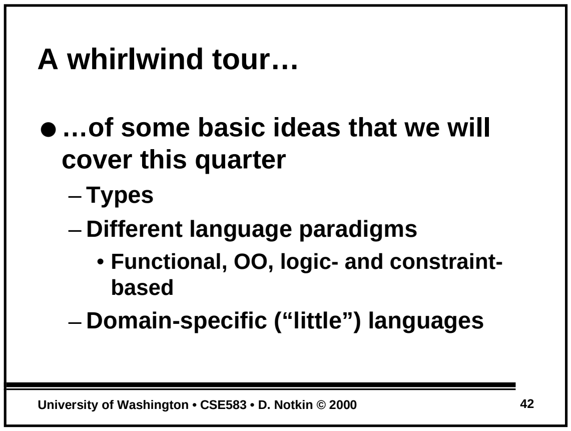
Ideally, it should be easy to…

- …quickly learn a programming language
- …quickly express intent and model application domains
- …read other people's code and understand their intent
- …debug & reason about correctness,
- …reason about performance trade-offs and ensure good performance
- …modify and extend programs



Tools also interact with the language

- Compilers analyze, optimize and translate
- Debuggers, program understanding tools, etc., aid programmers
  - Some tools must be language-knowledgeable
  - Other tools may benefit from knowing about the language at issue



A whirlwind tour…

- …of some basic ideas that we will cover this quarter
  - Types
  - Different language paradigms
    - Functional, OO, logic- and constraint-based
  - Domain-specific ("little") languages

Types

- Types are one of the most powerful notions developed in programming language research
  - Rich in theory and rich in practice
  - There are lots of disagreements about what is the right way to handle types
- Most simply, a type represents a collection of values
  - Integers, cartesian points, polygons, employees, etc.
- Types can be useful to the programmer, to the compiler, and as documentation

Strong vs. weak typing

- Strong typing (type safe)
  - Never apply an operation to an inappropriate data value without signaling an error
  - Never misuse a bit pattern in memory
  - Array bounds checking? Divide-by-zero checking?
  - Ex: Scheme, ML, Haskell, Smalltalk, Java, Prolog, safe subset of Modula-3
- Weak typing: not strong
  - C/C++, Pascal, Fortran, assembly languages

Static vs. dynamic typing

- Static typing
  - Check for type safety statically (at compile time)
  - Impossible for some aspects
    - Ex: array bounds (value vs. type checking)
  - Ex: ML, Haskell, Java, C/C++, Pascal, Fortran
    - At least they *think* they know the types
- Dynamic typing: not static
  - Scheme, Smalltalk, Prolog
  - Some are mixed (static/dynamic): CLU, Cecil

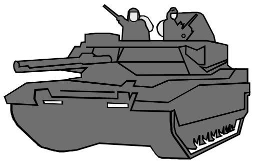
Types (miscellaneous)

- There is a tremendous amount of confusion in the terminology
  - Especially with”strong” vs. “static”
- Does static imply strong typing?
- Does strong imply static typing?

Imperative programming paradigm

- Most programming is done using the imperative paradigm
- Based on the Von Neumann machine with registers and modifiable memory
  - The memory is manipulated by the program through variables and assignments
  - Various constructs (especially control constructs) provide a way to structure the code and to order the manipulations of memory through variables and assignments

Object oriented paradigm



- We’ve had enough riots in Seattle lately, so let’s delay this discussion until later in the quarter
- “I am not a wuss”

Functional programming paradigm

- From the comp.lang.functional FAQ
  - "Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these language are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style."

Functional (con't)

- One central notion (of many) is referential transparency, which essentially means that computations are free of side-effects
  - That is, it's just like math, and you can always replace an expression by its value
- The following is always true in functional languages (but not in imperative ones)
  - `f(x) + f(x) = 2*f(x)`
- Makes I/O really fun!

Logic programming paradigm

- Use symbolic logic as a programming language
- This is good because logic is powerful and theorem proving can be used to "execute" programs
  - This will be clearer later in the term

```
app([], L, L).
app([A|X], Y, [A|Z])
  :- app(X, Y, Z).
```

- These two (Horn) clauses define list appending
- Can be used to compute in any direction or to check a property

Constraint logic programming paradigm

- Most logic programming is highly inefficient due to the needed search
- Some domains are more constrained, allowing efficient solutions to some limited but important classes of problems
  - Xerox paper flow

```
   DONALD
+  GERALD  =
   ROBERT
```

Domain specific languages

- "Programming" languages are written all the time
  - Jon Bentley called these "little languages"
  - Often without thought to programming language principles
- Examples?
- Tom Ball, Microsoft Research, will lecture on these later in the quarter

Questions?

- About content?
- About the course?
- About administrivia?