1 Intro

Lecture 1, 1/10/2019

1. Handout 1 — My names
2. Plagiarism — read aloud
3. E-mail list: CS450001@cs.uky.edu
4. Assignments on web. First assignment — Fortran
5. Accounts in MultiLab
6. Text (Sebesta, 10th edition) — we will follow somewhat

2 Software tools

<table>
<thead>
<tr>
<th>Use (client)</th>
<th>Spec</th>
<th>Programmer</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>→</td>
<td>Compiler</td>
<td></td>
</tr>
</tbody>
</table>

3 Fortran by examples

examples.f

4 Fortran jokes (from the net)

1. Lecture 2, 1/15/2019
2. God is REAL unless declared INTEGER.

3. Question: What will the scientific programming language of 2050 look like? Answer: No one knows, but it will be called FORTRAN.

4. CS without FORTRAN and COBOL is like birthday cake without ketchup and mustard.

5. Consistently separating words by spaces became a general custom about the tenth century CE, and lasted until about 1957, when FORTRAN abandoned the practice.

6. The primary purpose of the DATA statement is to give names to constants; instead of referring to pi as 3.141592653589793 at every appearance, the variable PI can be given that value with a DATA statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of pi change.

5 Java Puzzlers

6 Language evaluation criteria

1. Readability: important for maintenance as well as coding.
   
   (a) simplicity: small size
      i. number of basic constructs
      ii. number alternative ways to say the same thing (Consider incrementation in C, or conditionals in Perl)
      iii. number of meanings an operator (like +) might have
   
   (b) orthogonality: all combinations of basic features allowed.
      i. example (Algol): all statements have values (itself problematic: What is the value of a for loop?)
      ii. counterexample (C): functions cannot return struct values.
   
   (c) nested (Algol-like) control structures and name spaces
   
   (d) wide set of helpful data types and programmer-defined data types
   
   (e) readable syntax

2. writability: important for coding
(a) Support for abstraction: “ability to define and use complicated structures or operations in ways that allow many of the details to be ignored.” Abstraction is needed to manage the complexity of programming.

(b) expressivity (which is different from “power”; all programming languages can program Turing machines, so all are equally powerful): convenient ways to specify computations. Example: (Prolog) built-in backtracking.

3. reliability: important for debugging and maintenance

   (a) type checking
   (b) exception handling
   (c) restricted aliasing
   (d) (not in book) automatic memory allocation (as in Java, as opposed to C)

4. cost

   (a) training programmers (time, money)
   (b) writing programs (time, money)
   (c) compiling programs (time and space)
   (d) executing programs (time and space)
   (e) providing a compiler (time, money)
   (f) maintaining programs (time, money)

5. portability
6. generality (but beware of the Ada syndrome of over-complexity)
7. well-definedness (syntax is easy to specify, but semantics is harder)
8. But: a designer often has to trade one criterion for another.

   (a) reliability vs. cost of execution (array subscript checks)
   (b) expressivity vs. readability (APL)
   (c) writability vs. reliability (pointers)
   (d) generality vs. simplicity (Ada)
7 MacLennan’s principles

A related set of principles is given by [MacLennan slide], with principles such as

1. Labelling: Do not require the programmer to know the absolute position of an item in a list.
2. Structure: The static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations.

8 Language categories (programming paradigms)

A programming paradigm is a way to represent algorithms.

1. Lecture 3, 1/17/2019
2. procedural: procedure calls with parameters, return values
   (a) imperative (Fortran, Algol, Pascal, C): Variables hold values and have scope. Control structures based on statements, including sequences, assignments, compound statements, loops, procedure calls, exception handling.
      i. object-oriented (Java, C++, C#): imperative, with data and associated procedures organized in hierarchical classes.
      ii. visual (Visual BASIC, .NET languages): drag-and-drop generation of code, easy generation of GUIs.
      iii. scripting (Perl, Python, Ruby): string manipulation, invoking programs and manipulating results.
      iv. web-oriented (JavaScript, PHP, JSP): creating and manipulating document content.
   (b) functional (Lisp, ML): There are no variables, but there are named read-only parameters and possibly named constants. Control structures are based on expressions, high-order functions, and a heavy use of recursion.
3. declarative (or rule-based or logic) (Prolog, lparse, aspps, CP): rules with conditions and consequences; predicates
4. text-oriented (HTML, XML, TeX, nroff): not programming languages, but might have macros and nested structures.
5. other (RPG, APT, GPSS, SQL)

9 Compilation and interpretation

![Diagram of program preparation stages]

Stages in program preparation
1. compile: program → relocatable object code (ROC)
2. link: multiple ROCs and libraries → ROC
3. load: fully resolved ROC → absolute object code (AOC) (in memory)
4. execute: hardware treats AOC as program, not data.

10 Evolution of programming languages, according to Sebesta

1. See genealogy: [book Figure 2.1, page 37]
   (a) syntax: line oriented: 3 lines per statement (one for types, one for subscripts)
   (b) data: bits, integer, floating-point, arrays, records (nested)
   (c) control: *for*, multi-level *break, if* (without *else*)
   (d) assertions
3. Assembler language with macros.
   (a) Sebesta thinks these languages did not contribute to the main line of development of programming languages.
   (b) syntax: one line per operation, with symbols instead of opcodes and addresses + labelling
   (c) macros (typically for subroutine linkage)

4. Lecture 4, 1/22/2019

5. Pseudocodes
   (a) Include operations such as sqrt, sine, branches, I/O conversions.
   (b) Short code (Machuly 1949, Univac)
   (c) Speed coding (interpretive, Backus, IBM 701, 1954)

6. Fortran (IBM 704, 1954-60)
   (a) Constraints: small memories, unreliable computers, primary use is scientific, speed of code more important than cost of programmers.
   (b) Fortran I (1956)
      i. control: based on IBM 704 instructions
      ii. data: implicit typing only: integer and float
   (c) Fortran II (1958)
   (d) structure: independent compilation of subroutines
   (e) Fortran IV (ANSI: 1966)
      i. control: logical if, procedure-valued parameters
   (f) Fortran 77 (ANSI: 1978)
      i. data: string handling
      ii. control: while loops, if with optional else
   (g) Fortran 90 (ANSI: 1992)
      i. syntax: remove rigid position-based syntax; convention becomes that first letter only is capitalized in identifiers.
   (h) Fortran 95 (ISO: 1997)
      i. control: forall to aid parallelization
(i) Evaluation: Very influential. Showed that efficiency is possible with higher-level languages. Still in use, primarily in scientific code.

7. Functional programming: Lisp
   
   (a) We will skip this material for now.

8. Algol 58, Algol 60
   
   (a) Designed by committees in Europe.
   (b) data: dynamic-sized arrays (Sebesta calls them stack-dynamic)
   (c) control: block structure; parameter passing by name and by value; recursive procedures
   (d) Evaluation
      
      i. Used very heavily to describe algorithms, but not heavily used in USA.
      ii. Lack of I/O led to multiple versions.
      iii. Ancestor of very heavily used languages: C, C++, Java, C#, JavaScript, Go.

9. Cobol 60
   
   (a) syntax: macros (define); long names (30 characters)
   (b) data: hierarchical records (first appeared in Plankalkül, then here)
   (c) control: weak. No functions, no parameters to subroutines.
   (d) Evaluation: led to mechanization of accounting; still in very heavy use in business.

11 Syntax: Grammars

1. Grammars are a formal way to define the syntax of a programming language, which means how a program is composed, and the forms of its components, independent of their meaning.

2. Most syntax descriptions use BNF (Backus-Naur Form) or some variant; this formalism was introduced around 1960 for Algol-60.
3. Formal language theory defines a **language** as a set of (valid) **sentences** built out of **lexemes** (irreducible units). But for our purposes, a programming language is a set of (syntactically valid) **programs** built out of **tokens** (such as 1.232 or **while**).

4. **Lecture 5, 1/24/2019**

5. A BNF description is a collection of **productions** defining a **nonterminal** on the left-hand side in terms of both **terminals** and other nonterminals on the right-hand side.

6. One can use BNF to show what constitutes a **token**. Such a description can use recursion, but usually the Kleene star (*) makes such usages unnecessary. Such BNF actually defines a simpler set of possibilities known as a **regular language**.

   (a) \( \text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \)
   (b) \( \text{integer} \rightarrow \text{digit}^+ \)
   (c) \( \text{alpha} \rightarrow a \mid b \mid \ldots \mid z \)
   (d) \( \text{identifier} \rightarrow \text{alpha} \ (\text{alpha} \mid \text{digit})^* \)
   (e) \( \text{real} \rightarrow \text{digit}^+ \ . \ \text{digit}^+ \ [ \ E \ \text{digit}^+ \ ] \)

7. Comments on the grammar above

   (a) The exact syntax for BNF varies from book to book (and program to program). Some versions write nonterminals in braces, like \(<\text{digit}>\), and they write \(\rightarrow\) or \(::=\) instead of \(\rightarrow\).

   (b) We are using various extensions to ordinary BNF, namely:

   (c) The rule for **digit** makes use of **alternation**; one may write separate rules for each possibility instead.

   (d) The rule for **identifier** makes use of grouping parentheses and the Kleene star; one can avoid parentheses by introducing another nonterminal, and one can avoid the Kleene star by recursion:

      i. \( \text{alphaNum} \rightarrow \text{alpha} \mid \text{digit} \)
      ii. \( \text{id} \rightarrow \text{alpha} \ \text{alphaNumList} \)
      iii. \( \text{alphaNumList} \rightarrow \epsilon \mid \text{alphaNumList} \ \text{alphaNum} \)

   (e) The rule for **real** uses \([\ldots]\) for optional and Kleene +, both of which can be removed by alternation, \(\epsilon\), and recursion.

8. One can use BNF to show the syntax of the whole program. Example from C:
9. One can use a BNF in various ways.

(a) To derive valid programs (“sentences of the language defined by the BNF”). [build a derivation]

(b) Given a program, to determine how to derive it.
   i. The result looks like a tree; it is called a parse tree.
   ii. There are tools, such as lex (flex, jflex) and yacc (bison, javaCUP) that automatically generate a tokenizer and a parser from the BNF.
   iii. BNF is powerful enough to describe associativity (subtraction proceeds left-to-right, but exponentiation proceeds right-to-left) and operator precedence (multiplication occurs before subtraction).
      A. expression → (expression (+ | -) expression) | term
      B. term → term (* | / | %) factor | factor
      C. factor → primary ** factor | primary
      D. primary → integer | real | id | “(" expression “)"

(c) Notes on this grammar
   i. The rule for term is left-recursive, which gives us left-associativity for multiplication. The rule for factor is right-recursive, giving us right-associativity for exponentiation.
   ii. The rule for expression is ambiguous; there are two parses for the sentence “3 - 4 - 7”. Associativity is unspecified, because the rule is both left-recursive and right-recursive.
   iii. We can fix that rule by replacing the second use of expression by term to retain only left-recursion (and thereby left-associativity).

(d) If there can be more than one parse tree, the grammar is ambiguous.
i. Ambiguity is usually a mistake in the BNF.
ii. Ambiguity is sometimes allowed, so long as the parser always chooses the right version and the language definition agrees.
iii. Example: **dangling else**:

   ```
   if (x<0)
     if (y<0)
       y = y-1;
     else
       y = 0;
   ```

iv. C, Java, and Pascal: **else** always attaches to the closest preceding unmatched **if**.
v. Algol: **then** part must not be a nested **if**. — regularity.

   Sebesta p. 132 shows a BNF for a slight generalization: the **then** part must not be a non-**else** version of **if**.
vi. Go: both the **then** and **else** parts must be compound statements (surrounded by brackets).
vii. Ada, Modula: **if** must be closed by **end if** (or some other similar syntax), and deep nesting is avoided by **elsif**.

12 Theory of formal languages: the Chomsky hierarchy

1. regular languages (Chomsky’s type 3)
   (a) extended BNF without recursion.
   (b) insufficient for arbitrary nesting.
   (c) sufficient for defining tokens such as floating-point literals, identifiers.
   (d) parseable by finite-state machines.

2. context-free languages (Chomsky’s type 2)
   (a) extended BNF (including recursion).
   (b) sufficient for the syntax of programming languages except that scope rules (some people call that the **static semantics**) are not included.
   (c) parseable by a push-down automaton (a single stack).
(d) Earley’s algorithm (Jay Earley, 1970) can parse in $O(n^3)$ for ambiguous grammars and $O(n^2)$ for unambiguous grammars.
(e) Actual programming languages are more restrictive (in particular, they need very little lookahead), allowing $O(n)$ parsers.

3. context-sensitive languages (Chomsky’s type 1)
   
   (a) BNF, but allowing context terminals on the left-hand side of rules. (They are repeated on the right-hand side.)
   (b) sufficient for the syntax of programming languages, including scope rules.
   (c) parseable by a linear-bounded automaton, but very slowly.
   (d) Attribute grammars are an attempt to formalize scope information as part of parsing. They were of research interest in the 1970s and 1980s.

4. recursively enumerable languages (Chomsky’s type 0)
   
   (a) Rules may have arbitrary left-hand and right-hand sides.
   (b) Recognizable by Turing machines.

13 Pascal by examples

1. Lecture 6, 1/29/2019

14 Formal semantics

1. Lecture 7, 1/31/2019

2. The semantics of a programming language describes what programs mean, that is, what they do when running, as opposed to how they look.

3. Three ways of approaching semantics
   
   (a) Axiomatic semantics: each statement is defined by an axiom linking preconditions to postconditions, which are logical statements about the values of variables.
   (b) Operational semantics: each statement is defined by what it does to the state of a virtual machine
(c) **Denotational** semantics: the meaning of a program is a function linking inputs to outputs, composed of individual functions for each statement.

15 **Operational semantics**

1. Basic idea: translate programs (or statements) into a simpler **intermediate language** with its own interpreter.

2. Levels of use
   
   (a) Natural: See the final result of executing the whole program.
   
   (b) Structural: Inspect the translation of single components (such as statements)

3. Designing the intermediate language
   
   (a) Algol style: reduce control constructs to `goto` and `if-then` (without `else`); reduce expressions to single operators, introducing new variables to hold intermediate results.

16 **Axiomatic semantics (Hoare 1967)**

1. Background

   (a) Does not prove termination.
   
   (b) Only as good as the preconditions and postconditions
   
   (c) Led to a fad of proving programs correct
   
   (d) Led to a fad of teaching programming by precondition/postcondition/loop invariant, still evidenced by Eiffel.
   
   (e) Extension: weakest preconditions (Dijkstra 1975). Can prove termination, but it’s hard to discover loop invariants.

2. Based on placing **assertions** in the program and providing **axioms** that allow one to prove statements of the form \{P\} S \{Q\} meaning if predicate \(P\) is true before statement \(S\) starts, then after statement \(S\) completes, if it does, then \(Q\) must hold."

3. Axiom of assignment: \{Q_{x=E}\} x := E \{Q\}

4. Example: \{y = 12\} x := y + 2 \{x = 14\}
5. Weak and strong predicates

(a) If $P \Rightarrow Q$, we say that $P$ is **stronger than** $Q$.

(b) Strengthening a precondition $P$ in $\{P\} S \{Q\}$ **weakens** the entire statement; weakening the precondition **strengthens** the statement.

(c) Axioms try to show the strongest statements, that is, the weakest preconditions for which the statement always holds.

6. Lecture 8, 2/5/2019

7. Axiom of selection (**if** statements)
\[
\{B \land P\} S_1 \{Q\}, \{\neg B \land P\} S_2 \{Q\} \vdash \{P\} \text{if } B \text{ then } S_1 \text{ else } S_2 \{Q\}
\]

8. Axiom of iteration (**while** statements)
\[
\{B \land I\} S \{I\} \vdash \{I\} \textbf{while } B \textbf{ do } S \{\neg B \land I\}
\]
but no guarantee of completion.

9. Extended example: factorial

\[
\{true\}
\{1 = 1!\}
count := 1;
\{1 = count!\}
answer := 1;
\{answer = count!\}
\textbf{while } count != n \textbf{ do }
\{answer = count!\}
count := count + 1;
\{answer = (count-1)!\}
answer := answer * count;
\{answer = count!\}
\textbf{end;}
\{answer = count! \land count = n\}
\{answer = n!\}
\]

10. But the loop might not terminate: if $n < 1$.

11. Evaluation

(a) It is possible to prove small programs correct.
(b) Complex control structures (like `break` and concurrency) are very hard to model.
(c) Designing the proper overall preconditions and postconditions of a piece of code is at least as hard as designing the code.

17 Denotational semantics (Scott and Strachey 1971)

1. Lecture 9, 2/7/2019

2. Basic idea

   (a) One defines a complicated function that maps program fragments onto mathematical objects.
   (b) The denotation of a program is the mathematical object that the program maps onto.

3. Small example: function $S$ from statements and environments to updated environments, assuming no errors occur.

   $$S[\text{if } T \text{ then } St_1 \text{ else } St_2] \ u =$$
   $$\text{let } e = E[T] \ u$$
   $$\text{in }$$
   $$\text{if } e \text{ then } S[St_1]u \text{ else } S[St_2]u$$
   $$\text{end}$$

4. Evaluation

   (a) The semantic domains onto which one maps programs are recursively defined and therefore mathematically suspect.
   (b) It is very awkward (much harder than Sebesta indicates) to capture indefinite iteration (\texttt{while} loops).
   (c) Complete denotational descriptions cover all erroneous cases (at a terrible cost to readability), specifying exactly what an erroneous program means.
   (d) Denotational semantics is of little use to programmers.
   (e) One can try to automatically convert a denotational description of a language into a compiler.
18 Names: Syntax issues

1. Case sensitive?
   (a) Fortran, Lisp: no
   (b) Most Algol-derived languages: yes, but it is wise to follow capitalization conventions (as in Java)
   (c) Prolog: case determines role: variable or constant.

2. Keywords? In most modern languages, some words are reserved to be used only in their keyword role. Some early languages used delimiters (like dots) to show that a word was a keyword, such as `.begin.`., or depended on the context to determine if the word was a keyword.
   (a) **Predefined** names, like `int` in Pascal, are not reserved, but it is foolish to redefine them.

3. Valid length? Fortran II limited to 6, Fortran 95 limited to 31; Snobol and Ada have no limit. Java class files restrict length to 64K.

4. Regular form: typically `alpha (alpha | num | _)*`, but some languages disallow multiple contiguous underscores.


   (a) `int bar(int x));`
      declares a function `bar` that returns an int and has one parameter, not a variable `bar` initialized to `int(x)`.
   (b) All operator tokens have equivalent names
      ```
      && and & = and_eq & bitand | bitor ^ compl ! not
      != not_eq || or |= or_eq ^ xor ^= xor_eq
      { } [] <>
      <% %> <: :>
      ```
19 Names: Semantic issues

1. Lecture 10, 2/12/2019

2. Variable: Name used to abstract a memory cell or cells.

   (a) Attributes
   
   i. address: (static, often as offset from start of a frame). Also called the L-value of the variable. Can refer to multiple adjacent addresses, which together we call a memory cell. If two variables access the same address, they are aliases. This situation is error-prone.
   
   ii. value (dynamic): contents of the addressed cell. Also called the R-value of the variable.
   
   iii. type (usually static): set of values that can be stored in the address and how those values are interpreted.
   
   iv. lifetime (dynamic)
   
   v. scope (usually static)

3. Binding: associating a name (like a variable) to an attribute (like its location).

   (a) This definition is extremely general.
   
   (b) Early binding is usually cheaper (time, space) than late binding.
   
   (c) Late binding often provides more facility than early binding.
   
   (d) Example: When is the type of a variable determined?
   
   (e) Example (from Sebesta): count = count + 5. When are the pieces bound?
   
   (f) Static binding: occurs before run time (therefore at language definition time, compilation time, or link time); remains unchanged during program execution.
   
   (g) Dynamic binding: occurs during run time (therefore at load time, name-scope entry (elaboration), or statement execution).

4. Binding types to names (or more generally, expressions)

   (a) Names of what?
   
   i. constants: R value but no L value (then how are they passed in Fortran?)
   
   ii. variables
iii. procedures and functions: the type (called a **signature**) is dictated by their **prototype** or **header**. Usually the type is static, but in JavaScript it can be dynamic.

iv. expressions: syntactic sugar for (possibly nested) function calls. Have (dynamic) R value, no L value (how are they passed in Fortran?)

v. labels, as in Fortran, C, and Pascal.

vi. types, as in Pascal and C.

vii. classes, as in Smalltalk (or Java by reflection).

(b) **static**: by declarations

i. explicit, as in Pascal

ii. implicit, as in Fortran, PL/I, Basic. Good practice now is to say **IMPLICIT NONE** to prevent such declarations.

iii. limited and enhanced declarations

A. only binding a name to a type, not to an L value: C **extern**, Pascal **const**.

B. only introducing a name as valid and binding it to an L value, but not binding it to a type: Smalltalk instance variables.

C. also binding a value: initialized variables, constants, procedures and functions.

(c) **static**: by context of usage, as in Perl: $foo is a scalar variable (which can hold an integer, float, string, or pointer!), @foo is an array variable (holding only scalars), %foo is a hash variable (holding only scalars).

(d) **dynamic**: by right-hand side of assignment (Snobol, Smalltalk, JavaScript)

i. Late binding, so more expensive in time and space: operators must check the type before acting,

ii. More error-prone.

iii. More common in interpreted languages than compiled languages.

iv. The value is usually represented as a pointer behind the scenes.

(e) **dynamic**, by type inference (ML, Miranda, Haskell)
20 Smalltalk by examples

Lecture 11, 2/14/2019
Lecture 12, 2/19/2019

21 Bindings addresses to variables

1. Notation
   (a) **allocation**: Taking a cell from available memory and binding it to a variable.
   (b) **deallocation**: Returning the variable’s cell to available memory.
   (c) **lifetime**: Period (typically dynamic) between allocation and deallocation.

2. Static variables
   (a) The compiler/linker fixes the address, typically in a region called the **data segment**. In Unix, there are two data segments: initialized data (contents are stored in the object file) and uninitialized data (only the total size is specified by the object file).
   (b) Fortran: Every program and subroutine has its own static variables. The variables are stored in a per-subroutine **frame** that the compiler allocates. The frame also includes the (dynamic) return address, which is why recursion is not allowed.
   (c) C: Global variables (marked **extern**) are static.
   (d) Algol: Local variables marked **own** are static, even though they may have dynamic type, which is an unfortunate collision of features that is very hard to implement.
   (e) The lifetime is the entire execution, so variables retain values.
   (f) Run-time addressing is efficient.
   (g) Memory-intensive, because no sharing in space of values not needed at the same time.

3. Stack-dynamic variables
   (a) Usually stored on a single stack, which we call the **central stack**, but there can be multiple stacks (for concurrency).
(b) Allocated during elaboration of a scope, typically as a routine is instantiated.
(c) The allocation unit is a frame (or activation record), whose size is dependent on the routine (and possibly by sizes of dynamic types).
(d) Variables declared after statements might not yet be visible, but they are usually already allocated as the scope starts.
   i. C++ and Java: declarations may be anywhere in a scope.
   ii. C: new blocks can introduce declarations with limited scope, but the implementation usually allocates them at routine-elaboration time.
(e) Needed by recursion so each instance of a routine can have its own copy of local variables.
(f) Each stack frame is also used for linkage of routines. Its contents:
   i. Parameters (at static frame offsets)
   ii. Return address (points to code space)
   iii. Dynamic pointer, forming the dynamic chain: points to the start of the previous frame
   iv. Static pointer, forming the static chain: points to the frame of the lexical parent so that code can access non-local variables (and parameters).
   v. Local variables (at static frame offsets) (including hidden variables such as temporaries that don’t fit in registers)
(g) The cost of allocation and deallocation is trivial.
(h) The cost of access is slightly more than for statically allocated variables, typically as offsets from a register that points to the start of the current frame.

Lecture 13, 2/21/2019

4. Heap-dynamic variables

(a) Usually stored in a memory region called the heap, not to be confused with the heap data structure.
(b) Pascal, Java: allocation by new.
(c) C: allocation by malloc(3)
(d) Pascal, C: deallocation by free.
(e) JavaScript, Perl: value constructors can allocate.
(f) Java: automatic deallocation when value no longer in use.
(g) Can be accessed by **pointer**-valued variables. The pointers themselves can be stack-dynamic.
   i. Pascal: Heap-dynamic variables are exactly those accessible by pointers.
   ii. C: Any variable can be accessed by pointers, leading to insecurity.
   iii. Java, Smalltalk: No explicit pointer variables.

22 Type checking

1. Types serve several purposes.

   (a) The compiler can allocate the right amount of space + automation
   (b) The compiler can generate correct code. + impossible error
   (c) Programming errors can often be detected as type violations. + defense in depth However, not all type errors can be caught.
      i. If we use integers to represent colors, we might multiply the integers, although multiplying the colors is meaningless.
      ii. If we store both distance and time in reals, division makes sense (we get velocity), but not addition. My work on **dimensions** tries to remedy this problem.

2. A type error arises when an operation is attempted with parameters of a type for which it is not defined. Such errors are common in assembler programming.

3. A type system defines the bindings between a variable’s type, its values, and the operations on those values.

4. **Lecture 14, 2/26/2019**

5. A language is **strongly typed** if

   (a) Every value has a type. Expressions have values, and procedures and labels are also values, albeit second or third class (to be defined later).
   (b) Assignment and formal-actual bindings are restricted to **compatible** types, introducing type conversions if necessary.
   (c) All type errors can be detected, typically statically.
6. Algol-like languages try to be strongly typed.

(a) Pascal is mostly strongly typed, but it is possible to bind a formal procedure-valued parameter to an actual with a different signature. Untagged variants also introduce an explicit hole in strong typing.

(b) C is mostly strongly typed, but it is possible to invoke a procedure with the wrong number or types of actual parameters. Union types also introduce an explicit hole in strong typing.

(c) Ada and Java are strongly typed (with explicit casting loopholes).

23 Type equivalence

1. The compiler must reject any assignment or parameter binding with incompatible types.

2. Types are compatible if they are equivalent or if the language is willing to coerce the R-value to a type equivalent to the L-value’s type.

3. When are types equivalent?

   (a) Name equivalence (Pascal, Ada, Java): The types have the same name, or can be traced back to the same name.

      i. A type generator (type constructor) like array, record, pointerTo, or derived creates a new internal type name.

      ii. Strict (Ada): a declaration of multiple variables is a shorthand for multiple declarations; any type generator in the declaration is therefore expanded to multiple (different) types.

      iii. Lax (declaration equivalence: Pascal): a declaration of multiple variables shares any type generator among the variables.

   (b) Structural equivalence (Ada unconstrained arrays, Modula-3): The types have the same memory layout.

      i. Strict: arrays have the same bounds, same subscript type; record fields have the same names, records are not flattened.

      ii. Can be implemented inexpensively by a combination of compile-time effort (compute canonical representation and hash it) and run-time effort (compare actual hash with expected hash).
iii. Very useful for extending strong typing to data output by one program and input by another.

24 Scope

1. The scope of an identifier is the collection of statements that can access that identifier. An identifier is a name, which could refer to a constant, type, procedure, label, or variable.

2. Static scope: The scope of an identifier is based on where the statements are in the source program. Also called lexical scope.

   (a) Very common, including Fortran and all Algol derivatives.
   (b) Scope can be delimited by compilation units (C), packages (Java, Ada), classes (Java), functions (Algol), blocks (Algol), and for loops (Java, Ada).
   (c) Scopes can be nested (Java classes, Algol functions and blocks).
      i. Identifiers can be considered local, nonlocal, or global.
      ii. If the same name is declared twice (typically in an outer and inner scope), languages take different stances.
         A. Disallow.
         B. Inner declaration hides the outer declaration (Pascal).
         C. Hidden declarations can be accessed by qualified names.
         D. If the two meanings can be distinguished by usage, both are available (Java) but must be resolved, typically statically.
      iii. Nested scopes can lead to an overabundance of global variables.
   (d) Some languages require that all identifier declarations precede any statements in a scope (C, Pascal, Fortran); others allow intermingling, so long as each identifier is declared before use (C++, Java variables); some allow forward references (Java methods, and to a limited extent, C and Pascal)
   (e) Some languages do not require declaration at all, which violates — impossible error: Perl, Fortran.
   (f) Not all languages require that variables have a declared type, even though they allow or require that variables be declared: Perl, Smalltalk.
3. **Dynamic scope**: The scope of an identifier is based on where execution has been on its way to the statement.

   (a) Quite uncommon in modern languages; was present in Lisp 1.5 and is an option in Perl.
   (b) Subprograms have access to all variables in the dynamic path — reliability
   (c) It is impossible to statically check the type of nonlocals — reliability
   (d) Access to nonlocals tends to be slow, either because it requires runtime search or extra data structures set up during subroutine call.

25 **Static chain example (From Finkel, p. 24)**

```haskell
procedure A(X: integer, G: procedure);
    procedure B;
    begin
        writeln(X); { writes 2 }
        end; { B }
    begin { A }
        case X of
            2: A(1, B);
            1: A(0, G);
            0: G();
        end { case }
    end; { A }

procedure dummy; begin end; { never called }

A(2, dummy); { main }
```

1. main → A(2, dummy) → A(1, B) → A(0, B) → B().
2. Deep binding: When A(1) calls A(1), it passes B as a closure. When that B is finally called, the X it needs is the original 2.