1 Intro

Lecture 1, 1/14/2015

1. Handout 1 — My names
2. Plagiarism — read aloud
3. E-mail list: cs450001@cs.uky.edu
4. Assignments on web. First assignment — Fortran (simple database)
5. Accounts in MultiLab
6. Text (Sebesta, 10th edition) — we will follow somewhat
7. Extra 5 minutes every lecture?

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 Java Puzzlers

4 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)

5 Fortran by examples

Lecture 2, 1/16/2015 examples.f

6 Fortran jokes (from the net)

1. [Lecture 3, 1/21/2015]
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.
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. 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.

2. rule-based or logic (Prolog, lparses, aspps, CP): rules with conditions and consequences; predicates

3. text-oriented (HTML, XML, TeX, nroff): not programming languages, but might have macros and nested structures.

4. other (RPG, APT, GPSS, SQL)
9 Compilation and interpretation

Stages in program preparation
1. compile: program $\rightarrow$ **relocatable object code** (ROC)
2. link: multiple ROCs and libraries $\rightarrow$ ROC
3. load: fully resolved ROC $\rightarrow$ **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, 1/23/2015

5. Pseudocodes
   (a) Include operations such as \texttt{sqrt}, \texttt{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: \texttt{integer} and \texttt{float}
   (c) Fortran II (1958)
   (d) structure: independent compilation of subroutines
   (e) Fortran IV (ANSI: 1966)
      i. control: logical \texttt{if}, procedure-valued parameters
   (f) Fortran 77 (ANSI: 1978)
      i. data: string handling
      ii. control: \texttt{while} loops, \texttt{if} with optional \texttt{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: \texttt{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#.

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. Lecture 5, 1/26/2015

3. Most syntax descriptions use BNF (Backus-Naur Form) or some variant; this formalism was introduced around 1960 for Algol-60.

4. 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).

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) $\text{real} \rightarrow \text{digit}^+ \cdot \text{digit}^+ \ [ \varepsilon \text{digit}^+ ]$

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 $\text{digit}$ makes use of alternation; one may write separate rules for each possibility instead.
(d) The rule for $\text{identifier}$ 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. $\text{identifier} \rightarrow \text{alpha} \\text{alphaNumList}$
   iii. $\text{alphaNumList} \rightarrow \epsilon \mid \text{alphaNumericList} \text{alphaNum}$
(e) The rule for $\text{real}$ 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:

(a) $\text{program} \rightarrow ( \text{declaration} \mid \text{procedure} )^*$
(b) $\text{declaration} \rightarrow ( \text{int} \mid \text{real} ) \text{identifier} \ ( , \text{identifier} )^* \ ;$
(c) $\text{procedure} \rightarrow \text{header} \ \text{block}$
(d) $\text{header} \rightarrow ( \text{int} \mid \text{real} ) \ \text{identifier} \ ( ' ( , \text{id} )^* )'$
(e) $\text{block} \rightarrow \{ \ \text{declaration}^* \ \text{statement}^* \ \}$
(f) $\text{statement} \rightarrow ( \text{assignment} \mid \text{for} \mid \text{while} \mid \text{if} \mid \text{block} )$
(g) $\text{assignment} \rightarrow \text{identifier} \ = \ \text{expression} \ ;$
(h) if → if '(' expression ')' statement [ else statement ]

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) and yacc (bison) 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 | identifier | “(" 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:

\[
\begin{align*}
\text{if } & (x<0) \\
& \text{if } (y<0)
\end{align*}
\]
\[ y = y - 1; \]
\[ \text{else} \]
\[ y = 0; \]

iv. C, Java, and Pascal: \texttt{else} always attaches to the closest preceding unmatched \texttt{if}.

v. Algol: \texttt{then} part must not be a nested \texttt{if}. Sebesta p. 132 shows a BNF for a slight generalization: the \texttt{then} part must not be a non-\texttt{else} version of \texttt{if}.

vi. Go: both the \texttt{then} and \texttt{else} parts must be compound statements (surrounded by brackets).

vii. Ada, Modula: \texttt{if} must be closed by \texttt{end if} (or some other similar syntax), and deep nesting is avoided by \texttt{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 **Formal semantics**

1.  [Lecture 6, 1/30/2015]

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.

14 **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.

15 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.

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 \rightarrow x \rightarrow 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. Axiom of selection (if statements):
\{B \textbf{and} P\} S_1 Q, \{(\textbf{not} B) \textbf{and} P\} S_2 \{Q\} ⊢ 
\{P\} \textbf{if} B \textbf{then} S_1 \textbf{else} S_2 \{Q\}
7. Extended example: factorial

{true}
{1 = 1!}
count := 1;
{1 = count!}
answer = 1;
{answer = count!}
while count < n do
  {answer = count!}
count = count + 1;
  {answer = (count-1)!}
answer := answer * count;
{answer = count!}
end;
{answer = count! and count >= n}
{answer = n!}

To fix the disconnect between the last two lines, we need to also have an invariant that count <= n throughout.

8. 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.

16 Pascal by examples

1. Lecture 7, 2/2/2015

17 Denotational semantics (Scott and Strachey 1971)

1. Lecture 8, 2/4/2015
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}
   \quad e = E[T] \ u
   \in
   \quad \text{if } e \text{ then } S[St_1]u \ \text{else } S[St_2]u
   \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 (**while** loops).

   (c) Complete denotational descriptions cover all erroneous cases, clarifying 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? No in Fortran, Lisp; Yes in most Algol-derived languages. In Prolog, case determines role as a variable or a 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.
4. Regular form: typically \textit{alpha} (\textit{alpha} | \textit{num} | \textunderscore)^*, but some languages might disallow multiple contiguous underscores.

5. Conventions: Separate words in a variable name by underscore: \texttt{big\_num}, or by camel notation (internal capitals): \texttt{bigNum}. Use all caps for constants, initial caps for classes.

19 Names: Semantic issues

1. \textbf{Variable}: Name used to abstract a memory cell or cells.

   (a) Attributes
   
   i. address: (static, often as offset from start of a \texttt{frame}). Also called the \textbf{L-value} of the variable. Can refer to multiple adjacent addresses, which together we call a \textbf{memory cell}. If two variables access the same address, they are \textit{aliases}. This situation is error-prone.
   
   ii. value (dynamic): contents of the addressed cell. also called the \textbf{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) \textbf{Early binding} is usually cheaper (time, space) than late binding.
   
   (c) \textbf{Late binding} often provides more facility than early binding.
   
   (d) Example: When is the type of a variable determined?
   
   (e) Example (from Sebesta): \texttt{count = count + 5}. \textbf{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).
5. Binding names (or more generally, expressions) to types

(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, @foo is an array variable, %foo is a hash variable.

(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 Address bindings for 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) Lecture 11, 2/11/2015
(e) 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.
(f) The lifetime is the entire execution, so variables retain values.
(g) Run-time addressing is efficient.
(h) 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 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.

5. 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.

21 Type checking

1. Types serve several purposes.
   (a) The compiler can allocate the right amount of space.
   (b) The compiler can generate correct code.
   (c) Programming errors can often be detected as type violations.

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. 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.

5. 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).

22 Type equivalence

1. [2/16/2015] Cancelled due to snow

2. Lecture 13, 2/18/2015

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

4. 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.

5. 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.

23 Scope

1. [2/20/2015] Cancelled due to snow

2. [Lecture 14, 2/23/2015]

3. 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.

4. 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.

5. **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.

### 24 Static chain example (From Finkel, p. 24)

```
1. procedure A(X: integer, G: procedure) {
   procedure B() {
      write(X); // writes 2
   } // B()
   switch (X) {
      case 2: A(1, B); break;
      case 1: A(0, G); break;
      case 0: G(); break;
   } // switch
} // A()

procedure dummy() {}; // never called

// main
A(2, dummy);

2. [Lecture 15, 2/25/2015]

3. main → A(2, dummy) → A(1, B) → A(0, B) → B().

4. 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.
```
25 Data types — Overview

1. Some languages provide almost no datatypes (BCPL). Others provide many (PL/I). Most languages provide a few datatypes and a way to introduce new ones.

2. Each type is described by a **descriptor**.
   
   (a) For integers, the descriptor might indicate number of bytes.
   
   (b) For arrays, the descriptor indicates subscript and element types (as pointers to other descriptors) and per-dimension ranges.
   
   (c) For records, the descriptor indicates fields and their types (as pointers to other descriptors).
   
   (d) The compiler stores type descriptors in the symbol table (ST).
   
   (e) Some type descriptors need to be dynamic, at least in part. Example: dynamic-sized arrays (Pascal). Dynamic type descriptors are on the stack.

26 Primitive data types

1. **integer**
   
   (a) Some languages have varieties of different storage sizes (Fortran-IV, C, Ada, Java), which might be called **short**, **int**, **long**, **long long**.
   
   (b) Usually stored in **twos complement**.
   
   (c) MININT = -MININT, in two’s complement.
   
   (d) Unsigned variants of integer are available (C).
   
   (e) Operations include **arithmetic** (+, −, *, **div**, **mod**, sometimes **) and **comparison** (including <=> in Perl).
   
   (f) One must carefully define **div** and **mod** to accommodate negative operands.
   
   (g) Arithmetic overflow is possible, treated by truncation or exception. The result has the wrong sign (and value).
   
   (h) Division by 0 causes an exception or results in NaN (not a number).

2. [Lecture 16, 2/27/2015] Smalltalk by examples
4. real
   (a) Different storage sizes are often available.
   (b) Different representations (fixed, float) are available in Ada.
   (c) The IEEE 754 standard (1980) suggests (in single precision)
       i. one sign bit
       ii. 8-bit exponent $e$ representing $-127 \ldots 128$ (in excess-127 notation)
       iii. 23-bit mantissa, with an assumed initial 1 bit (hidden)
   (d) The IEEE 754 standard also defines longer precisions, and it can represent both $\infty$ and NaN.

5. complex
   (a) Stored as two reals, usually representing real and imaginary parts (but $\rho, \theta$ representation is possible).
   (b) Quite rare; only in Fortran.

6. Boolean
   (a) Can be packed into 1 bit, but usually expanded to 8. (C and Perl: not distinct from integer – impossible error).

7. character
   (a) can be packed into integers (Fortran).
   (b) encodings
       i. ASCII (7 bits)
       ii. ASCII plus a second “code page” for extended alphabets (8 bits)
       iii. FIELDDATA (obsolete: Univac)
       iv. EBCDIC (obsolete: IBM)
       v. Unicode (originally 16, now 32 bits), often represented by UTF-8, which uses multiple 8-bit chunks (Perl, Java).
   (c) operations include comparison, which may involve locale-specific rules.
   (d) Python, Perl: string of length 1
27 Strings

1. Length restrictions

(a) static: immutable, length fixed at creation time (Java).
(b) limited dynamic: up to the allocated size (C, C++)
(c) dynamic: no maximum, varying length (Perl, JavaScript, Snobol)

2. Fortran: possible to pack 6 characters into an integer; Hollerith constants in FORMAT statements.

3. Lecture 18, 3/4/2015


(a) Doesn’t work well for UTF-8.
(b) To allocate: malloc(strlen(theString)+1) to leave room for the null terminator. – impossible error
(c) Assignment in C is pointer copy, not shallow copy. One needs to use strcpy(3) or strncpy(3) instead.
(d) C, C++: There is no protection against indexing past the end of the array – impossible error


(a) Operations: match against a pattern by regular expression, substitute, adjust case, concatenate, extract substring, search for character.
(b) Java instances of String are read-only; instances of StringBuilder are like character arrays.

6. Storage organization

(a) Compile-time descriptor might contain length.
(b) Run-time descriptor might contain current length, start address, maximum length.
(c) For dynamic length strings: modifications might be implemented by complete copy into fresh heap.
28 **Enumeration types**

1. (Pascal, C, Java) + labelling + impossible error
2. Comparable, discrete.
3. How to define I/O?
4. Convertible to integer?

29 **Subtypes**

1. A **subtype** is a type with (more) constraints placed on its values.
2. Members of the subtype inherit all operations of the base type.
3. Examples
   
   (a) Pascal: `type smallInt = 1 .. 10`
   
   (b) Ada: `subtype Weekend is Day range Saturday .. Sunday`
   
   (c) Java: subclasses
4. Assignment compatibility, where A is a variable of some type, and B is a variable of its subtype.
   
   (a) A := B — always allowed.
   
   (b) B := A — maybe allowed; implicit static or dynamic constraint check.
   
   (c) B := (cast to B) A — allowed; explicit static or dynamic constraint check.

30 **Midterm**

1. **Snow day, 3/6/2015**
2. **Midterm, 3/9/2015**
3. **Midterm review, 3/11/2015**
31 Lisp introduction

1. [Lecture 19, 3/13/2015] Lisp by examples

32 Arrays

An array is an indexed sequence of values.

1. [Lecture 20, 3/23/2015]

2. Notation: the index is of the subscript type, and the values are of the element type.

3. Homogeneity

   (a) Homogeneous (typical for statically typed languages): all the values have the same element type.
   (b) Inhomogeneous (typical for dynamically typed languages): the values may have different element type.

4. Dimension: the number of components to the index.

   (a) Fortran: 1, 2, or 3 dimensions only. $[-0, 1, \infty]$
   (b) Algol: Any positive number of dimensions. $[+0, 1, \infty]$
   (c) Pascal: One dimension, but the element type may itself be an array type. $[+0, 1, \infty] + \text{regularity}$
   (d) APL: 0-dimensional array is a simple scalar.
   (e) C: One dimension, but actually represented by a pointer. The element type can itself be a pointer, leading, as in Pascal, to higher-dimensional arrays.

5. Layout: applies to higher-dimensional arrays. What is placed after $A(x, y)$ in memory?

   (a) Row-major: $A(x, y+1)$ (or the next row): Most languages
   (b) Column-major: $A(x+1, y)$ (or the next column): Fortran
   (c) Why does it make a difference?

6. Bounds

   (a) subscript type is integer; always starts at 0 (C).
(b) subscript type is integer; always starts at 1 (Fortran).
(c) subscript type is integer; programmer specifies lower and upper bounds (Algol)
(d) subscript type is any discrete, finite type; programmer specifies lower and upper bounds (Pascal)

7. Sizing

(a) Static size
   i. Bounds are known at compile time (and are part of the type).
   ii. The array can be allocated statically or stack-dynamically.

(b) Dynamic size
   i. Bounds computed at elaboration time (but may still be part of the type).
   ii. Usually allocated stack-dynamic, but can be heap-dynamic
      (C: explicit call to malloc(3) and free(3); Java: always)
   iii. Where is the data itself placed in the activation record? A
        pointer (location vector) is placed at a static offset; the data
        are placed later in the activation record (stack-dynamic allocation).
   iv. What if a dynamic type is elaborated in one scope, and a
        variable is declared of that type in a deeper scope? The type
        may be needed at runtime (for bounds checking and even
        for index calculation); store it in its activation record (dope
        vector)

(c) Flexible size: bounds not determined; as cells are assigned values, they become defined (Perl). Allocation is heap-dynamic. Leads to neat features:
   i. Ability to push, pop, shift, unshift values on/off arrays.
   ii. Ability to concatenate arrays.

8. Indexing

(a) Arbitrary expression of the subscript type (Fortran allows only limited expressions).
(b) Actual address calculation is based on bounds, lengths of each dimension, and size of the element type. It can include bounds checking (static or dynamic).
(c) Negative subscript in a 0-based array means “from the end” (Perl).

(d) **Array slice**: a set of adjacent cells, such as \([3..10]\). Array slices are usually only allowed in the last dimension (for Row-major), but APL allows array slices in any dimension.

(e) An array element or slice is a valid L-value.

(f) Pascal: Array assignment is valid; shallow-copy semantics.

(g) Java: Array assignment is valid; pointer-copy semantics.

(h) C: Array assignment is invalid.

9. Initialization

   (a) No initialization: Pascal, C.

   (b) Initialized to element-type specific default: Java, Perl.

   (c) Explicit initializer syntax (C, Java).

33 Pointers

1. Lecture 21, 3/25/2015

2. Do not interfere with strong typing.

3. Address-of (**referencing**) operator (in C: \&). Can lead to dangling references to the stack. Not allowed in Pascal [+ impossible error]

   (a) Pascal strictly distinguishes heap objects (which are only accessible through pointers) and stack objects (which are never accessible by pointers)


5. Requires dynamic memory management: explicit **new** and **free**, which can be error-prone.

6. Java therefore does not have explicit pointers. [+ impossible error]. It does have **reference types**, though, for all class instances. Likewise, Smalltalk, Python, and Ruby variables are all references.

7. Assignment operator: **pointer assignment**, as opposed to **shallow copy** or **deep copy**.
8. C arrays are represented by pointers, leading to new operators: array subscripting and addition of integers. Out-of-bound errors.

9. It is not required that a pointer reference a structure. Consider these types:

```typescript
// type
intPtrType ^integer;
strangePtrType ^strangePtrType;

// var
var
intPtr : intPtrType;
sp : strangePtrType;

begin
new(intPtr);
intPtr^ = 4;
new(sp);
new(sp^);
sp^^ = sp;
end;
```

10. Heap management
(a) Lock and key to make sure that a pointer always refers to a valid region of the heap.
(b) Reference counts, which fail to deallocate circular structures, use extra space, and add to the cost of all reference operations.
(c) Garbage collection, which usually interrupts regular processing while reclaiming memory in several phases: mark and sweep.

### 34 Arithmetic expressions

1. Lecture 22, 3/27/2015

2. operator precedence: \( a + b * c \) must have a well-defined meaning. Most languages have a set of precedence levels, with unary minus > multiplication and division > addition and subtraction > boolean operators.

3. operator associativity: \( a + b + c \) must be done in some order. Most languages specify left-associative for all operators except exponentiation. APL is right-associative for all operators. Associativity makes
a difference for subtraction, division, exponentiation, but not (necessarily) for addition, multiplication, or, and, xor.

4. Parentheses: Universally available to explicitly specify precedence (within parentheses happens first) and associativity (within parentheses is grouped).

5. Smalltalk and Ruby: Operators are just shorthands for calls on methods, so they all have the same precedence and are all left-associative.

6. Conditional expressions: b ? x : y (in ML: if b then x else y).

7. Side effects: order of evaluation can influence meaning and even correctness of execution: a + fun(a).

8. In Algol-W, which has a result parameter-passing mode, a call like fun(x,x) can place one of two values in x depending on the compiler’s choice.

9. Overloading
   (a) Can be confusing, as with & in C, which means both “address of” and “bitwise and”.
   (b) Another confusion is the / operator, which can be overloaded (as in C and Java) or always return real (as in Pascal). JavaScript avoids the problem by having no integers!
   (c) Abstract data types (and their generalization, classes) can introduce programmer-defined overloading.

10. Type conversion from X to Y
    (a) Narrowing: Y cannot store even approximations of all values of X. Example: float has a much smaller range than double. Often unsafe.
    (b) Widening: Y can at least store approximations of X. Example: float can approximate integer, even if not exactly. Generally safe, but precision can be lost.
    (c) Explicit (converting cast): Programmer indicates conversion.
    (d) Implicit (coercion): Compiler chooses conversion.
        i. Mixed-mode expressions: Languages differ in how willing they are to coerce types. Ada is very strict; Java is very lenient.
(e) Non-converting cast: C++\texttt{reinterpret\_cast<type>}; C pointer casts: \texttt{* ((\texttt{Y}*) \&\texttt{X})}

11. Underflow and overflow: Usually not detected, even at runtime, due to the expense. But division by 0 is usually detected.

35 Relational and Boolean expressions

1. Lecture 23, 3/30/2015
2. Six standard Boolean-returning relational operators:
   
   \begin{verbatim}
   < <= = > >= !=
   \end{verbatim}

   Perl adds a comparison operator \texttt{<>} and its non-numeric version \texttt{cmp}, but they do not produce Booleans.

3. Boolean combining operators: \texttt{not}, \texttt{and}, and \texttt{or}, in order of decreasing precedence.

4. Short-circuit (lazy) evaluation: \texttt{A or B} is known to be true as soon as \texttt{A} is known to be true; \texttt{A and B} is known to be false as soon as \texttt{A} is known to be false. In these cases, there is no need to compute the value of \texttt{B}.

   (a) Short-circuiting changes semantics, because computing \texttt{B} might have a side effect (modifying a global variable or causing an error).

   (b) Some languages (Ada, Java) provide different operators for short-circuit versions of \texttt{and} and \texttt{or}.

36 Assignment statements

1. Syntax: usually either = or :=; sometimes \texttt{←}.

2. Conditional target (C++): \texttt{flag ? var1 : var2 = value}.

3. Assignment expression: \texttt{var1 = var2 = value}. Treat assignment as an expression returning the value of the RHS; the assignment operator is right-associative. Can be error prone (C): \texttt{if (x = y)} is legitimate but has a surprising semantics. (Java and C# distinguish integer from Boolean, making such an error much less likely.)
4. Compound assignment: \( x += \text{value} \), where various operators are available in addition to +. Makes it easier to generate efficient code, and the program can be clearer. Java warning: not quite the same semantics as the full form with respect to coercions [Java Puzzlers #9].

5. Unary assignment: \( x++ \) and \( ++x \) (also with \( -- \)). Can be error-prone.

6. Multiple target (Perl, Ruby): \( x, y := \text{value1, value2} \). The L-values of the LHS are first evaluated, then the R-values of the RHS, so one may use multiple targets to swap two variables. If the same variable appears multiple times on the left, the order is obscure.

37 Using iterators to generate binary trees

Lecture 24, 4/1/2015
This example is in Python.

def binGen(size):
    if size > 0:
        for root in range(size):
            for left in binGen(root):
                for right in binGen(size - root - 1):
                    yield("cons(" + left + "," + right + ")")
    else:
        yield "-"

for aTree in binGen(3):
    print aTree

38 Control statements: Selection

1. General policy: only one entrance (no goto into an if statement!), one typical exit, with possible extra error exits.

2. Two-way selection: if

(a) Each branch is a statement (in Perl and Go, each branch must be a compound statement).

(b) dangling else problem (seen earlier, page 9), solved either by disallowing if in the then part (Algol), matching else with
the nearest preceding unmatched if (Pascal), or closing syntax end if (Ada), which requires elseif to avoid deep nesting.
(c) as a postfix operation (Perl), which is nice if the body is a single statement.

3. Multiple selection: case (Pascal), switch (C, Java)
   (a) Branches: must use break to avoid falling through (C); implicit termination of each branch (Pascal, Go); explicit fallthrough (Go)
   (b) Labels: static values (C), static ranges (Pascal), dynamic expressions (Go). Why static?
   (c) [Lecture 25, 4/3/2015]
   (d) Label type: integer, string (Java 7, Go), any discrete type (Pascal).
   (e) Generated code
      i. multiple tests, as if a series of conditionals: good if few cases.
      ii. jump table: good if the labels are dense
      iii. binary search tree: good for many, sparse ranges
      iv. hash table: good for many, sparse, values

39 Iteration

1. Basic questions: how is iteration controlled (logical or counting?), and where does the control appear in the loop construct (top=pretest or bottom=posttest)?
2. Counter-based loops (for)
3. Arrangement
   (a) Specific: Loop variable (control variable), initial, terminal values, stepsize.
   (b) General: initialization (can include declaration of loop variables), termination test, advancing statement
4. scope of loop variable: the recent trend is to make it a local declaration.
5. value of the loop variable after termination: generally unwise to depend on any particular value.

6. modifying the loop variable (or the initial or terminal values) within the loop: generally unwise; some languages prevent.

7. are the loop parameters evaluated once before the loop, or after each iteration? — once (Pascal), after each iteration (C).

8. branches into the loop from outside: generally forbidden; certainly unwise.

9. Looping over non-contiguous or non-numeric sets: still have a loop variable.
   
   (a) (Python, Perl) may loop over members of a set (represented as an array, possibly computed on the spot)
   (b) (CLU, Python, JavaScript 1.7) may loop over values returned by an iterator. Example: generating binary trees.

10. Logically controlled loops (**while**)
   
   (a) both pretest and posttest (**until**) versions.
   (b) not as perspicuous as **for**, because the “increment” is hidden.
   (c) Go therefore only has **for**.

11. Programmer-located control mechanisms

   (a) **break** or **last**: exits loop; can be multilevel with labelled loops
   (b) **continue** or **next**: finishes this iteration only
   (c) **redo**: restarts this iteration

### 40 Guarded commands

1. Guarded **if**

   (a) Syntax: each branch has a Boolean **guard** and a statement; branches separated by a **fatbar** [], entire construct terminated by **fi**.
   (b) Semantics: Evaluate all guards. If all are false, error. Otherwise choose any true guard and execute its statement. Fairness is not required, but is good.

2. Guarded **do**
(a) Syntax: much like guarded `if`.
(b) Semantics: Evaluate all guards. If all are false, done. Otherwise choose any true guard, execute its statement, and repeat.
(c) Example: Euclid’s algorithm for greatest common divisor of A and B:

```plaintext
a, b := A, B;
do {
    a < b: b := b - a;
    | b < a: a := a - b;
}
```

## 41 Subprograms

1. **Lecture 26, 4/6/2015**

2. Also called procedures, functions (particularly when they return a value), methods, and subroutines.

3. Single entry point; caller is blocked for the duration (unlike threads); control returns once to the caller (unlike iterators).


   (a) Header indicates that it is a subprogram, possibly distinguishing functions (which return a value) from procedures (which don’t). It also names the subprogram, gives return type, lists formal parameters with types (in Perl, parameters are not listed; in Smalltalk, no types).

   (b) Usually, the declaration binds the name to the subprogram at compile time. But in some languages, declaration is executable and can bind at runtime (Python, ML).

   (c) Functional languages (Lisp, ML) allow anonymous procedures. So do Go, JavaScript, Java 8.

   (d) Prototypes: In languages in which a function must be declared before use (Pascal) or may be in order to introduce types (C, C++), one can build a prototype, which is the header alone.

5. Parameters
(a) **Formal**: the names used inside the procedure body. Usually allocated space at invocation time on the stack.

(b) **Actual**: the values (in general, expressions) used by the call.
   
i. Usually **positional**: presented as an ordered, comma-separated list.
   
ii. Some languages (Ada, Python) allow **keyword** syntax, in which the caller names the formal for each actual, in any order.
   
iii. Hybrids are allowed in Ada and Python, but once you have a keyword actual, all the following ones must also be keyword.

(c) Default values for formals (C++, Python, Ada) when the call omits that formal. Must be last if positional.

(d) Variable number of parameters. Perl, Java, C#: The actuals form an array, and there is only one formal. C has its own method, used by `printf(3)`.

6. Local variables

   (a) Usually allocated in stack-dynamic fashion (Algol), but sometimes statically (Fortran, **static** in C).

   (b) Nested subprograms are allowed in Algol, Pascal, Python, but not in C.

7. Parameter-passing modes

   (a) semantic modes: **in**, **out**, **in-out**. These are all that Ada provides. The actual for **out** mode must be a variable; beware parameter collisions.

   (b) semantic marker: **readonly** (Ada), **final** (Java). Can be enforced by the compiler.

   (c) implementation modes

      i. copy the R-value (in: **value**, out: **result**; in-out: **value-result**) C only provides value mode. This mode is good for short values, but copy cost is high for long values.
      
      ii. copy the L-value (address): **reference**. Fortran only provides reference mode, even for actual expressions, which must be evaluated to a temporary location, whose address
is then passed. Reference mode allows the subprogram to modify the actual immediately. It is slightly more expensive to access reference-mode parameters.

iii. name mode (Algol): every access to the L-value or R-value of the formal is equivalent to an access to the L-value or R-value of the actual.

A. Use: Jensen’s device:

```c
int sum(name int a, j) {
    int answer = 0;
    for (j = 0; j < 10; j++) {
        answer += a;
    }
}
```

```c
int sum = sum(B[k], k); // sum of B[1..10]
print(sum(k*k, k)); // sum of first 10 squares
```

B. Implemented by thunks: Little procedures running in the referencing environment of the caller returning the L-value or the R-value of the actual.

C. Puzzle: try to build a swap routine using only name-mode parameters. This solution does not work:

```c
void swap(name int x, y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```c
swap(B[j], j); // works ok
swap(j, B[j]); // fails
```

8. Arrays (especially multidimensional) as parameters

(a) Original Pascal: Must completely specify type

(b) Standard Pascal and Ada: compliant arrays, indicating number of dimensions, index and base type, but leaving ranges open. Later binding, therefore more expensive; the compiler must pass more information, and the syntax must provide a way to access the bounds. In Ada, you say `myArray’first[3]` to get the low bound of the 3rd dimension. In Fortran, the program must pass such information in extra parameters.
42  Prolog by examples

Lecture 27, 4/8/2015

43  First, second, third class values

1. [Lecture 28, 4/13/2015]

2. First-class values: may be assigned to variables (in languages that have variables), may be returned from functions. Integers are always first-class.

3. Second-class values: may be passed as actuals to procedures, but are not first-class. Procedures are often second-class.

4. Third-class values: may be used in their intended way, but are not second-class. Labels and types are usually third-class.

44  Second-class procedures

1. Type checking

   (a) The full type of a procedure is its protocol: the number and types of its parameters and its return type (but not the names of the parameters and not the procedure name. You might call protocol equivalence a form of “lax structural equivalence”.)

2. Passing a nested procedure P: what is its nonlocal referencing environment (NLRE)? Assume that P’s lexical parent is Q and that P is finally invoked by R.

   (a) Shallow binding: P’s NLRE is bound when R calls P as the top-most frame for Q. This choice is very difficult to implement, although possible.

   (b) Deep binding: P’s NLRE is bound when P is elaborated by Q and is transmitted along with P’s code pointer when P is passed as a parameter. The package containing the code pointer and the NLRE pointer is called a closure. When R calls P, R initializes P’s frame from the closure.

function sub1() {
    var x;
    function sub2() {
        alert(x);
    }
    function sub3() {
        var x;
        x = 3;
        sub4(sub2);
    }
    function sub4(subx) {
        var x;
        x = 4;
        subx();
    }
    // sub1 body:
    x = 1;
    sub3();
    sub1();
}

4. main → sub1 → sub3 → sub4 → sub2.

5. Deep binding: When sub3 calls sub4, it passes a pointer to sub1’s frame as the NLRE in the closure. When sub4 calls sub2, sub2 gets the right NLRE, namely, the one belonging to sub1.

6. Shallow binding: There is only one frame for sub1 on the stack, so we use it. (*The book is wrong here!*)

7. More informative example (From Finkel, p. 24)

procedure A(X:integer, G:procedure) {
    procedure B() {
        write(X); // writes 2
    } // B()
    switch (X) {
        case 2: A(1, B); break;
        case 1: A(0, G); break;
        case 0: G(); break;
    } // switch
} // A()

procedure dummy() {}; // never called
// main
A(2, dummy);

8. main → A(2, dummy) → A(1, B) → A(0, B) → B(

9. 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.

45 First-class procedures?

1. One must represent a first-class procedure as a closure, but there is a risk that the NLRE in the closure is stale.
   
   (a) Don’t let procedures be first-class (Pascal).
   (b) Only let top-level procedures be first-class (Modula-2).
   (c) Only have top-level procedures (C)
   (d) Retain frames until no references remain, carving all frames from the heap (Smalltalk, JavaScript).

46 Second-class labels?

1. Jump to a label means unwinding the stack.
2. Therefore the label must be a closure: address in code, frame owning the label.

47 Overloaded procedures

1. A form of polymorphism (overload polymorphism, although the book calls it ad-hoc polymorphism)
2. Usually easy to resolve statically based on procedure protocol.
3. However, type coercions and default parameters can enable multiple competing versions of a procedure.
4. Disambiguation can be exponentially hard even without type coercions and default parameters (Ada).
5. From www.coderanch.com/t/417622/java-programmer-SCJP/certification/Golden-Rules-widening-boxing-varargs:

(a) 1. Primitive Widening > Boxing > Varargs.
(b) 2. Widening and Boxing (WB) not allowed.
(c) 3. Boxing and Widening (BW) allowed.
(d) 4. While overloading, Widening + vararg vs Boxing + vararg
can only be used in a mutually exclusive manner i.e. not to-
gether.
(e) 5. Widening between wrapper classes not allowed

Call: doX(5)
Overloaded methods Called method
doX(Integer i) & doX(long l) long (by Rule 1)
doX(int...i) & doX(Integer i) Integer (by Rule 1)
doX(Long l) & doX(int...i) int...i (Rule 2 & 1)
doX(Long l) & doX(Integer...i) Integer...i (R. 2&1)
doX(Object o) & doX(Long l) Object o (Rule 2&3)
doX(Object o) & doX(int...i) Object o (Rule 3&1)
doX(Object o) & doX(long l) long l (Rule 3&1)
doX(long...l) & doX(Integer...i) ambiguous (Rule 4)
doX(long...l) & doX(Integer i) Integer (Rule 1)

Call: doX(Integer(5))
doX(Long l) error (Rule 5)
doX(Long l) & doX(long...l) long...l (Rule 5 & 1)

48 Generic code

1. A form of polymorphism: parametric polymorphism
2. The term generic is from Ada; C++ calls it a template.
3. Something like compile-time second-class types.
4. The code is instantiated zero or more times at compile time with the
types desired; each instance leads to separate compiled code.
5. Available in Ada, C++, C#, and Java 1.5.
6. The formal type parameter might include constraints, such as inter-
faces that the type must implement.
7. The formal type parameter might be a template, with a wildcard (Java):

```java
void printCollection(Collection<?> c) { ... }
```

49 Aside: forms of polymorphism

1. Overload: Static procedure overloading with compile-time resolution (Ada, Java). Here, the type of the procedure (its signature) determines whether it is a viable candidate for resolving the overloading.

2. Parametric: Generic packages (Ada), templates (C++). This sort of polymorphism allows types to be statically second class.

3. [Lecture 30, 4/17/2015]

4. Dynamic method binding: (Java, Smalltalk, C++ deferred binding). Here, the dynamic type (class) of the value (object) determines which procedure (method) to invoke.

5. Type-identifier: Types described by type identifiers (perhaps with the compiler inferring the types of values) (ML). Here, the dynamic type constraints on the parameter and return value determine the effective type of the function. For instance, the type of a sorting procedure might be this:

```plaintext
('a list * ('a * 'a -> int)) -> 'a list
```

Here, 'a is a type identifier. The type above indicates a procedure that takes two parameters: a list of some element type ('a) and a comparison procedure that takes two elements and returns an integer (−1, 0, 1) to show their relative order. The procedure returns a list of the same element type. Similarly, a function-composition procedure would have this polymorphic type:

```plaintext
('a -> 'b) * ('b -> 'c) -> ('a -> 'c)
```

50 Coroutines

1. [Lecture 31, 4/20/2015]

2. Multiple entry points, controlled by the code.
3. Simula-67: Every record has initialization code, which may detach before completing.

4. The program may resume an incomplete initialization, which pauses the caller and causes the resumed thread to continue.

5. This technique is needed for efficiently comparing the fringes of two trees for equality.

6. See [Finkel: pp 37-38]

51 Implementing subprograms

1. **Linkage**: call and return operations.

2. The linkage is the joint responsibility of the caller and subprogram; the compiler writer must design who does which steps.

3. Stack holds **frames** (activation records).

4. Call

   (a) Save the current execution status: place partial expression evaluation results and volatile registers in temporary locations (in current frame).

   (b) Establish the frame of the called procedure, including

      i. Dynamic pointer to the caller’s frame (either the top or the bottom, so the new frame can be removed)

      ii. Static pointer (if the language allows non-local references, so they can be resolved)

      iii. Parameters (initialized according to their mode)

   (c) Jump to the subroutine.

5. Return

   (a) Copy any **out** parameters to the caller.

   (b) Copy the return value to a place accessible to the caller (global, or in caller’s frame)

   (c) Restore registers.

   (d) Jump to return address.

6. Blocks (compound statements with local declarations)
(a) The compiler can treat them as immediate calls to anonymous parameterless procedures, thereby introducing a frame.
(b) The compiler can hoist the declarations to the level of the current subroutine and avoid the cost of introducing a frame.
   i. This design works because anonymous procedures cannot recurse.
   ii. The space allocated for variables of different blocks can overlap.

7. [Lecture 32, 4/22/2015]
8. Dynamic scope

   (a) Simple idea for accessing a variable: search through the dynamic chain, from most recent to least recent, for the desired variable. This method is called deep access (do not confuse with deep binding).
   (b) A faster method for access (but slower linkage) makes a stack of values for each variable; access each variable by accessing the top of that stack. This method is called shallow access (do not confuse with shallow binding).

52 Abstract data types (ADTs)

1. Abstraction means only viewing the most significant attributes of an entity.
   (a) Process abstraction: reducing a complex operation to a single name.
   (b) Data abstraction: reducing a data type to its manipulation methods.

2. General idea
   (a) Declare data values along with operations (functions and procedures, typically) that pertain to those values.
   (b) client = importer. Does not see the concrete implementation, although the compiler might need to see at least the sizes of the data.
   (c) server = exporter. Does not see the client’s usage pattern.
(d) Separate **specification** (a syntactic unit seen by both) from the **implementation** (a syntactic unit seen by the exporter) from the **invocation** (seen by the importer).

(e) These ideas are based on Simula-67 classes: Data fields and procedure fields are grouped together.

(f) Information hiding, leading to reliability, because the importer cannot deal with the underlying representation directly.

(g) Allows separate compilation

3. Typical use of abstract data type: **structure manager**

   (a) The package exports a single opaque type and many procedures that take values of that type.

   (b) The client is expected to declare variables (or arrays) of the exported type.

4. Everyone’s favorite example: Stack

   (a) Representation might be an array or a linked list; the importer doesn’t know or care.

   (b) The specification might include `push()`, `pop()`, `top()`, and needs some way of creating and destroying stacks (which might be part of the specification)

5. Other possible use of an abstract data type: **data container**.

   (a) The specification exports variables, acting as restricted-use global data.

   (b) Example: complex numbers

   (c) Each package gives you just a single collection of data; the only way to get multiple copies is by statically instantiating a generic package several times.

53 **Abstract data types in Ada**

1. The construct is called a **package**.

2. Packages generally have two parts that can be compiled separately: the **specification** and the **implementation** (marked by **body**).

3. Typically, a package exports a single type.
4. The type (such as Stack) is declared to be private in the specification. Still, the compiler needs to know some of its properties (in particular, size), so the specification also has a private section giving more detail.

5. Alternatively, the specification can declare the exported type to be a pointer to a completely unspecified type. Now the compiler has enough information, and less is revealed to the client.

6. However, pointer types can be mis-used (such as failure to initialize or attempt to test equality, which becomes pointer equality).

7. Ada provides limited private, which has no operations at all, including assignment and equality test.


54 Syntax for encapsulation

1. Lecture 33, 4/24/2015

2. We want to be able to organize large programs in compilation units that can be separately compiled.

3. Nested subprograms can be used for organizing programs, but that organization does not address the separate-compilation issue.

4. C: Header files contain common type definitions and procedure protocols.

   (a) Each source file uses #include to include the relevant header files.

   (b) There are tools (like makedef) for constructing dependency graphs in large projects with multiple header files so that the make program recompiles when necessary, but only as much as is necessary.

5. C++: The programmer may place declarations placed within namespaces.

   (a) To declare a namespace:

      namespace theNamespace {
          ...
      }


(b) To refer to an identifier in a namespace by full qualification:
\[\text{theNamespace::theIdentifier}\]
(c) To import a single identifier into scope from a namespace:
\[\text{using theNamespace::theIdentifier;}\]
(d) To bring the entire namespace into scope:
\[\text{using namespace theNamespace;}\]

6. Java: Classes can be collected into packages, which have implicit privileges over the protected or default-permission members of all the classes in the package.

(a) To indicate that a compilation unit belongs to a package:
\[\text{package thePackage;}\]
(b) To reference an identifier (typically a class name) in a package:
\[\text{thePackage.theIdentifier}\]
(c) To import a particular identifier from a package:
\[\text{import thePackage.theIdentifier;}\]
\[\text{...}\]
\[\text{theIdentifier}\]
(d) To import all identifiers from a package (not considered good practice):
\[\text{import thePackage::*;}\]

55 Object-oriented programming

1. Major components of object-oriented programming
   (a) abstract data types (data encapsulation), possibly with generics
   (b) inheritance, good for specialization and for code reuse
   (c) dynamic method-binding polymorphism (not just overload polymorphism, which is statically resolvable)

2. Basic idea: The concept of type is expanded to class; the concept of value is expanded to instance.

3. A class defines
   (a) instance variables: like fields in a record type.
4. In some languages, a class also can define

(a) **class variables**: shared among all instances; generally treated as at the same lexical level as instance variables.

(b) **class methods**: methods that are not bound to an instance; they may not access instance variables. Java: **static methods**.

(c) **constructors**: methods automatically invoked when an instance is created. In Simula, the constructor looks like a main block of the class; it is allowed to **detach**, leading to coroutines. Constructors are in charge of initialization.

(d) **destructors**: methods automatically invoked when an instance leaves scope, is explicitly deallocated, or is garbage collected. Destructors are in charge of finalization.

(e) **inner classes**: classes defined within a class definition, leading to arbitrarily deep non-local referencing environments.

5. **Lecture 34, 4/27/2015**

6. Inheritance

(a) A class $S$ can **extend** another class $B$. We say that $S$ is a **subclass** of $B$, and that $B$ is the **superclass** of $S$.

(b) All methods and variables of $B$ are available in $S$.

(c) $S$ may define additional members.

(d) $S$ may declare members that override members of $B$.

(e) resolution of methods (dynamic method-binding polymorphism)

i. **static** (early binding): fast, but limiting. Default in C++, C#. Available in Java: **final** methods.

ii. **deferred** (late binding): slower, but allows subclasses to override a method declaration successfully. Default in Java; obligatory in Smalltalk; optional in C++: **virtual**. Not necessarily inefficient (C++: only five additional memory references).
(f) **Multiple inheritance**: a subclass that extends more than one superclass. The language must deal with conflicts of inherited members, including “diamond inheritance” where the same member is inherited in two different paths upwards.

i. Java does not have multiple inheritance, but it allows a class to advertise that it implements multiple interfaces.

ii. An **interface** is like a class, but it is a promise of the existence and type of certain members, not an implementation. Classes that implement an interface are obliged (by the compiler) to fulfil the promise, providing all the members.

iii. Some languages (Haskell) allow the interface itself to provide some implementation details. For example, an ordered interface might require a definition of `less` but itself implement `greater` based on `less` and `equal`.

(g) Inheritance is *not* the same as **nesting** a class within a class, which is useful if the nested class is only needed by the surrounding class.

7. Abstract methods and classes

   (a) A member might be so abstract that it is only meant as an advertisement, to be fulfilled in subclasses.

   (b) Such a member is declared **abstract** (in C++, it’s called **virtual**; in Java, the entire class must also be declared **abstract**).

   (c) One may not generate an instance of a class that has an abstract member, but one may generate instances of subclasses (if they implement the abstract member).

8. Forms of name conflict in Java

   (a) **Overriding**: Instance method > accessible instance methods with the same name in superclasses. Generally OK.

   (b) **Hiding**: field, static method, member type > accessible things respectively with the same name (for methods: same signature) in superclasses. A hidden member is not inherited. Bad.

   (c) **Overloading**: Method > other method with different signature; resolution is at compile time. Often unclear.
(d) **Shadowing**: field, method, type > similar things respectively in an enclosing scope. The shadowed thing might be accessible by a qualified name. Generally bad except for constructor idiom.

(e) **Obscuring**: variable > type > package with the same name if several interpretations are possible. Very bad; comes from violating naming conventions.

9. **Lecture 35, 4/29/2015**

10. **Information hiding**

<table>
<thead>
<tr>
<th>language</th>
<th>mode</th>
<th>other instance of same class</th>
<th>instance of friend class (or same package)</th>
<th>inherited instance</th>
<th>unrelated instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smalltalk</td>
<td>variable</td>
<td>n</td>
<td>-</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>method</td>
<td>y</td>
<td>-</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>C++/Java</td>
<td>public</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>C++/Java</td>
<td>protected</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>C++/Java</td>
<td>private</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Java</td>
<td>default</td>
<td>y</td>
<td>y</td>
<td>n?</td>
<td>n</td>
</tr>
</tbody>
</table>

11. **Allocation and deallocation concerns.**

(a) Objects can be **value variables**: static (C++) and stack-dynamic (C++), or **pointer variables**: heap-dynamic (Java, Smalltalk, C++).

(b) Example (C++)

```c++
void p() {
    List x1, x2, *y1, *y2;
    y1 = new List();
    y1->insert(1);
    y2 = y1;  // pointer copy
    x1.insert(1);
    x2 = x1;  // shallow copy
}
```

(c) Assignment semantics

i. Value variables \((x_1, x_2)\): Assignment uses shallow copy. Subclass instances might be larger than the declared class, so enough space must be left for the largest possible subclass, or assignment must be restricted (run-time checking). C++ forces value variables to be statically typed (no subclass instances allowed).

ii. Pointer variables \((y_1, y_2)\): pointer copy.


(d) Allocation time

i. Stack-dynamic: when the surrounding block is elaborated.
ii. Pointer variables: explicit new operation.
iii. Initialization code might be implicitly called (Java, C++ constructors) or it might require explicit invocation (Smalltalk). Implicit constructors may call the constructors of the parent class.

(e) Deallocation

i. Value variables: no special work required to deallocate.
ii. Pointer variables: usually garbage collection (Smalltalk, Java), although one can have explicit deallocation (delete in C++), which can be error-prone.
iii. Rarely, the programmer wishes finalization code to run at deallocation time. Unfortunately, one cannot predict when that code will run if there is garbage collection.

56 Support for object-orientation in various languages

1. To start the search for a member under deferred binding in the superclass of an object: super (Smalltalk)

2. Integrated programming development environments: First appeared for Smalltalk (windows, mouse, menus).

3. Simple types are not objects, for efficiency reasons.

   (a) Smalltalk: special case for integers, but seamlessly appear to be objects.
   (b) Java, C++: Boolean, character, numeric. C++: enumerations, arrays (Both are objects in Java.)
   (c) boxing: wrapping (typically implicitly) a simple type like int into a class like Integer.

4. Members that cannot be overridden: final (Java)

57 Concurrency

1. [Lecture 36, 5/1/2015]
2. Background

(a) A thread is an independent locus of control. It may or may not share address space with other threads. If so, some call it a light-weight thread; if not, some call it a process.

(b) Synchronization is a mechanism that controls the order in which actions occur. It typically prevents (blocks) one thread from proceeding until some other thread has accomplished some action.

(c) Mutual exclusion is a form of synchronization that prevents two threads from simultaneously accessing a shared data structure in a way that might cause inconsistencies, such as when one thread is writing into the data while another thread is reading it. The book calls mutual exclusion competition synchronization.

i. Code that accesses shared data is a critical section.
ii. Conflicting critical sections are those that access the same data.
iii. Only conflicting critical sections must be mutually exclusive.

(d) The standard examples for synchronization are the bounded buffer (a shared data structure with independent producers and consumers) and the dining philosophers (five agents pairwise sharing five chopstick, two of which are needed to eat).

3. Starting threads

(a) Explicit call to fork() and maybe join(), which might be library routines.

(b) cobegin ... coend.

(c) Modula: process is like a void procedure, but calling it starts a new thread, which terminates when it “returns”.

(d) Ada: Tasks

i. Declared like procedures.
ii. When a block with tasks is elaborated, the tasks start.
iii. The block does not exit until its own code and all the tasks have completed or are mutually blocked.

4. Semaphores
(a) Two atomic operations: \texttt{up} and \texttt{down}. The book calls them \texttt{release} and \texttt{wait}. E. Dijkstra, who invented semaphores, called the operations \textit{V} and \textit{P}, based on Dutch words that he later couldn’t remember.

(b) \texttt{up} increments the semaphore’s count. If the count is now non-positive, it also unblocks the first waiting thread.

(c) \texttt{down} decrements the semaphore’s count. If the count is now negative, it also blocks the calling thread, placing it in a queue.

(d) Use for mutual exclusion
   
   i. Each shared data structure has its own semaphore.
   
   ii. Initialize each semaphore’s value to 1.
   
   iii. At the start of a critical section, perform \texttt{down} on the critical section’s associated semaphore.
   
   iv. At the end of a critical section, perform \texttt{up} on the critical section’s associated semaphore.

(e) Use for cooperation (wait-until) synchronization, where B must wait for A to accomplish some task
   
   i. Initialize the semaphore to 0.
   
   ii. When B needs to wait, B executes \texttt{down} on the semaphore.
   
   iii. When A wishes to allow B to continue, A executes \texttt{up} on the semaphore.

5. Locks: simple mutual exclusion based on a single semaphore.

6. Conditional critical regions

   (a) Each shared variable must be declared \texttt{shared}; compiler enforces the rule that only such variables may be accessed from multiple threads.

   (b) All access to shared variables must be within a \texttt{region} statement that names all the shared variables that are accessed in a single synchronization-atomic way.

   (c) The \texttt{region} statement may begin with \texttt{await} with a Boolean condition. The condition is checked after the thread acquires exclusive access to the region variables; if it is false, exclusion is released.

   (d) Whenever a thread exits from a region, all blocked threads are given a chance to re-check their condition.

(a) A form of abstract data type.
(b) The exported procedures are mutually exclusive. Hoare called them guard procedures.
(c) A new data type, used for private variables inside the implementation of the abstract data type: condition. Two operations on condition variables: wait and signal.
   i. wait blocks the caller (placing it on a queue associated with the condition variable) and releases exclusion.
   ii. signal unblocks the first thread, if any, in the associated queue, giving it exclusive use of the monitor; the caller either exits the monitor (best programming practice, enforced in some languages) or is placed in a high-priority wait queue to run only when it can regain exclusion.
   iii. broadcast is a version of signal that unblocks all waiters for the condition, but gives only the first one exclusive access.
(d) Bounded buffer implementation
   i. The bounded buffer is an abstract data type that exports put() and get(), to be used by producers and consumers, respectively.
   ii. The bounded buffer has two private condition variables: notEmpty and notFull.
   iii. put:
       if (buffer is full) wait(notFull);
       place data in buffer;
       signal(notEmpty)
   iv. get:
       if (buffer is empty) wait(notEmpty);
       take data from buffer;
       signal(notFull)

8. Ada rendezvous
(a) A thread may invoke another by calling an entry (with actual parameters).
(b) The called thread must explicitely receive such calls with an accept (with formal parameters).
(c) A caller is blocked until the called task reaches a matching accept; a task that executes accept is blocked until there is a matching call.

(d) Once the call is matched to an accept, the caller and called thread are in rendezvous.

(e) A select statement can pick one of several accepts based on guards (Boolean conditions) and whether a caller is currently trying to enter into a rendezvous.

9. Message passing

(a) Message passing is usually mediated by the underlying operating system, so programming languages only provide a way of invoking the operating system’s function.

(b) The basic operations are send, receive.

(c) Both send and receive are typically be explicit and blocking.

(d) receive might be non-blocking (in case no messages have arrived).

(e) receive might be implicit, tied to a procedure (called a callback procedure).