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

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) digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   (b) integer → digit+
   (c) alpha → a | b | ... | z
   (d) identifier → alpha ( alpha | digit )* 
   (e) real → digit+ . digit+ [ E 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 <digit>, and they write => or ::= instead of →.

   (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. alphaNum → alpha | digit
      ii. id → alpha alphaNumList
      iii. alphaNumList → ϵ | alphaNumList alphaNum

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

8. One can use BNF to show the syntax of the whole program. Example from C:
(a) program → ( declaration | procedure )*  
(b) declaration → (int | real) id (, id )* ;  
(c) procedure → header block  
(d) header → (int | real) id ‘(’ id (, id )* ’)  
(e) block → { declaration* statement* }  
(f) statement → ( assignment | for | while | if | block)  
(g) assignment → id = 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, 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)

\[
\{B \land I\} S \{I\} \vdash \{I\} \text{while } B \text{ do } S \{\neg B \land I\}
\]

but no guarantee of completion.

9. Extended example: factorial

```plaintext
1 {true}
2 {1 = 1!}
3 count := 1;
4 {1 = count!}
5 answer := 1;
6 {answer = count!}
7 while count != n do
8     {answer = count!}
9     count := count + 1;
10     {answer = (count-1)!}
11     answer := answer * count;
12     {answer = count!}
13 end;
14 {answer = count! \land count = n}
15 {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.

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

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 (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.


6. Unfortunate syntax in C++ (see http://madebyevan.com/obscure-cpp-features/)
   (a) \texttt{int bar(int (x));}
       \textit{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

```c
1 & & and &= and_eq & bitand | bitor ^ compl ! not
2 != not_eq || or |= or_eq ^ xor ^= xor_eq
3 { } [ ]
4 < % $> < : :>
```
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): \( \text{count} = \text{count} + 5 \). 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.
   (b) The compiler can generate correct code.
   (c) Programming errors can often be detected as type violations.

   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 Static chain example (From Finkel, p. 24)

```
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.

Midterm, 2/28/2019

25 Scope

1. [Lecture 15, 3/5/2019]

2. 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.

3. 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 \textbf{for} loops (Java, Ada).

(c) Scopes can be nested (Java classes, Algol functions and blocks).
   i. Identifiers can be considered \textbf{local}, \textbf{nonlocal}, or \textbf{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 \textbf{hides} the outer declaration (Pascal).
      C. Hidden declarations can be accessed by \textbf{qualified names}.
      D. If the two meanings can be distinguished by usage, both are available (Java) but must be \textbf{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 \textbf{forward references} (Java methods, and to a limited extent, C and Pascal)

(e) Some languages do not require declaration at all, which violates \textbf{– 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.

4. \textbf{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 \textbf{– reliability}
   (c) It is impossible to statically check the type of nonlocals \textbf{– reliability}
   (d) Access to nonlocals tends to be slow, either because it requires runtime search or extra data structures set up during subroutine call.
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.

3. 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.

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. 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.

3. complex

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

4. Boolean

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

5. 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

28 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.

   (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) Lecture 19, 3/26/2019
   (b) Operations: match against a pattern by regular expression, substitute, adjust case, concatenate, extract substring, search for character.
   (c) Java instances of String are immutable; instances of StringBuilder are like character arrays.

5. 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.

29 Enumeration types

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

30 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) \( B := \text{cast to } B \) \( A \) — allowed; explicit static or dynamic constraint check.

31 Arrays

An array is an indexed sequence of values.

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

3. **Lecture 20, 3/28/2019**

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. \(+0, 1, \infty\) + 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) + regularity

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 `a[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).

32  Prolog by examples

33  Pointers

1.  Lecture 22, 4/4/2019

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. − impossible error out-of-bound errors.

9.  It is not required that a pointer reference a structure. Consider these types:
type
intPtrType = `integer;
strangePtrType = `strangePtrType;
var
intPtr : intPtrType;
sp : strangePtrType;
begin
new (intPtr);
intPtr^ = 4;
new (sp);
new (sp^);
sp^^ = sp;
end;

Lecture 23, 4/9/2019

11. 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. 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.

2. 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, $\lor$, $\land$, $\oplus$.

3. Parentheses: (Almost) universally available to explicitly specify precedence (within parentheses happens first) and associativity (within parentheses is grouped).
4. Smalltalk and Ruby: Operators are just shorthands for calls on methods, so they all have the same precedence and are all left-associative.

5. Conditional expressions: \[ b \ ? \ x \ : \ y \] (in ML: \textit{if} \ b \ \textit{then} \ x \ \textit{else} \ y).

6. Side effects: order of evaluation can influence meaning and even correctness of execution: \[ a + \text{fun}(a) \]. In C: \textit{f}(j++, j++)

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

8. Overloading
   (a) Can be confusing, as with \texttt{&} in C, which means both “address of” and “bitwise and”.
   (b) Another confusion is the \texttt{/} operator, which can be overloaded for integer and real (as in C, Java, and Perl) 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.

9. Type conversion from X to Y
   (a) Narrowing: Y cannot store even approximations of all values of X. Example: \texttt{float} has a much smaller range than \texttt{double}. Often unsafe.
   (b) Widening: Y can at least store approximations of X. Example: \texttt{float} can approximate \texttt{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) Lecture 24, 4/11/2019
   (f) Non-converting cast: C++ \texttt{reinterpret_cast<TYPE>}; C pointer casts: \* (\texttt{((Y*) \& X})

10. 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. Six standard Boolean-returning relational operators: 

\[
< \quad \leq \quad = \quad > \quad \geq \quad !=
\]

Perl adds a comparison operator \(<=\) and its non-numeric version `cmp`, but they produce values \{-1, 0, 1\}, not Booleans.

2. Boolean combining operators: `not`, `and`, and `or`, in order of decreasing precedence.

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

   (a) Short-circuiting changes semantics, because computing \(B\) 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 `and` and `or`. Java: `&&` is short-circuit; `&` is not.

36  **Assignment statements**  

1. Syntax: usually either `=` or `:=`; sometimes `←`.


3. Assignment expression: `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): `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 += 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: `x = x++`. 
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

This example is in Python.

```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. [Lecture 25, 4/16/2019]

2. General policy: only one entry (no \texttt{goto} into an \texttt{if} statement!), one typical exit, with possible extra error exits.

3. Two-way selection: \texttt{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 10), solved either by disallowing \texttt{if} in the \texttt{then} part (Algol), matching \texttt{else} with the nearest preceding unmatched \texttt{if} (Pascal), or closing syntax \texttt{end if} (Ada), which requires \texttt{elseif} to avoid deep nesting.

   (c) as a postfix operation (Perl), which is nice if the body is a single statement.

4. Multiple selection: \texttt{case} (Pascal), \texttt{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) Label type: integer, string (Java 7, Go), any discrete type (Pascal).
(d) 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. [Lecture 26, 4/18/2019]

3. Counter-based loops (for)

4. Arrangement

  (a) Specific (Fortran, Pascal): Loop variable (control variable), initial, terminal values, stepsize.
  (b) General (C, Java): initialization (can include declaration of loop variables), termination test, advancing statement
  (c) Array-based (Perl): Loop variable, array.
  (d) Iterator-based (CLU, Python): Loop variable, iterator

5. scope of loop variable: the recent trend is to make it a local declaration.

6. value of the loop variable after termination: generally unwise to depend on any particular value.

7. modifying the loop variable (or the initial or terminal values) within the loop: generally unwise; some languages prevent.
8. are the loop parameters evaluated once before the loop, or after each iteration? — once (Pascal), after each iteration (C).

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

10. 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.
   (c) Algol 60: baroque structure, including explicit values, values advancing by some formula.

11. 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.

12. 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 { b a := b ∨ a;
      | b a := a ∨ b;
  }
```

### 41 Subprograms

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

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

3. Syntax: **header** and **body**.

   (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 (in most languages), 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, the 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.