CS 354: Programming Languages
Slides

Chapter 1: Introduction to Programming Language

1.01 CS 354: Programming Languages
1.02 Why study programming languages (PLs)?
1.03 There are different "levels" of PL
1.04 Why are there so many PLs?
1.05 What makes a PL successful?
1.06 Why do we have PLs? What is a PL for?
1.07 Why study PLs?
1.13 PL paradigms
1.18 Euclid's greatest common divisor algorithm
1.19 Compilation versus interpretation
1.24 Implementation strategies
1.31 Unconventional compilers
1.32 Other programming-environment tools
1.34 Phases of compilation
1.35 Scanning
1.37 Parsing
1.38 Semantic analysis
1.39 Intermediate code generation
1.40 Machine-independent optimization
1.41 Code generation
1.42 Machine-dependent code optimization
1.43 Symbol table
1.44 A scanning example
1.45 A parsing example
1.46 A parse-tree example

1.47 A syntax-tree example
1.49 A code-generation example
Chapter 2: Programming Language Syntax
2.01 Important definitions
2.03 Recursive definition of regular expressions (REs)
2.04 A regular-definition for numeric literals in Pascal
2.05 Context-Free Grammars (CFGs)
2.06 A CFG for simple expressions
2.07 Parse tree for `3+4*5`
2.08 Parse tree for `10-4+3`
2.09 Derivation of `3+4*5`
Chapter 3: Names, Scopes, and Bindings
3.01 Names, scopes, and bindings
3.02 Binding time
3.03 Run time
3.04 Binding
3.06 Scope
3.07 Lifetime and storage-management events
3.08 Scope and lifetime
3.09 Storage-allocation mechanisms
3.10 Linux `x86_64` Process Memory
3.11 Static allocation
3.12 Stack allocation
3.13 Stack frames
3.15 Stack-frame allocation and deallocation
3.16 Heap allocation
3.17 Introduction to scope rules
3.18 Static Scope
3.27 Dynamic Scope
3.29 Implementing scope

3.30 The meaning of names within a scope
3.33 Binding referencing environments
3.36 Macro expansion
3.37 Separate Compilation in C
3.38 Chapter conclusions
Chapter 4: Semantic Analysis
4.01 Semantic analysis
Chapter 5: Target Machine Architecture
Chapter 6: Control Flow
6.01 Control-flow paradigms
6.02 Ordering in expression evaluation
6.04 Precedence and associativity
6.05 Parameter-evaluation order
6.06 Arithmetic identities
6.07 Short-circuit evaluation
6.08 Variables as values or references
6.09 Values, references, or pointers
6.10 Orthogonality
6.13 Side effects
6.17 How can we avoid side effects?
6.18 Statement ordering: unstructured flow
6.19 Nonlocal gotos
6.20 Statement sequencing
6.21 Selection
6.24 Iteration
6.28 Iterators
6.29 Generators
6.30 Recursion
6.31 Tail recursion
6.32 Argument evaluation order (revisited)

6.33 Nondeterminancy
Chapter 7: Type Systems
7.01 Data Types
7.02 What is a type?
7.03 Why do we want types?
7.04 Type system
7.06 Built-in types
7.07 User-defined types
7.08 Orthogonality (revisited)
7.09 Type checking
7.10 Is type equivalence obvious?
7.11 Type equivalence
7.12 Type-equivalence compromise
7.13 Type conversion and coercion
7.14 Coercions
7.16 Casts
7.18 Records (Structures) and Variants (Unions)
7.20 Alignment
7.21 With statements
7.22 Discriminated and free unions
7.23 Unions and type-system subversion
7.24 Arrays
7.26 Array allocation
7.29 Array layout
7.30 Array traversal and caches
7.31 Other layout strategies
7.32 Element-address computation
7.33 Character strings
7.34 String-oriented PLs
7.36 Sets

7.37 Pascal sets
7.38 Sets in other PLs
7.39 Pointers and recursive types
7.40 Pointers and trees
7.41 C pointers and arrays
7.42 Pointer problems
7.43 Run-time dangling-reference solutions
7.45 Garbage collection
7.46 Garbage-collection pitfalls
7.48 Alternative to explicit deallocation andgarbage collection
7.49 Lists
7.50 Dotted pairs
7.51 Comprehensions
7.52 Files and input/output
7.53 Equality testing and assignment
Chapter 8: Composite Types
8.01 Subroutines and Control Abstraction
8.02 Where are we?
8.03 Parameter allocation: We saw ···
8.04 Calling sequences
8.05 Typical stack-frame structure
8.06 GCC (4.8.2) x86_64 stack-frame structure
8.07 Registers
8.08 Parameter passing
8.09 Pass by-value
8.10 Pass by-reference
8.11 Pass by-value/result
8.12 Pass by-name
8.13 A menagerie of parameterish features
8.20 Closures

8.23 Generics (aka, Templates)
8.24 Prehistoric Generics
8.25 Java and C++ Examples
8.26 Interesting Generic Variations
8.29 Implementation
8.32 Constraints (aka, Bounds)
8.34 Exception Handling
8.35 Prehistoric Exception Handling
8.36 Exception Handling Semantics
8.37 Exceptional Examples

CS 354: Programming Languages

- Roster and passwords
- Our `pub` directory:

  `onyx:~jbuffenb/classes/354/pub`

  `pub/ch1/gcd/gcd.scm`

  Canvas
- Our lecture slides, table of contents, and code:

  `pub/slides/slides.pdf`

  `pub/slides/code.tar`
- Review syllabus:

  `http://csweb.boisestate.edu/~buff`

  `pub/syllabus-1/syllabus.pdf`

  `pub/syllabus-2/syllabus.pdf`
- 1: Introduction

Why study programming languages (PLs)?

- Knowing multiple PLs is good.
  - Every PL does not have every feature.
  - There are different paradigms.
  - Different PLs are good for different tasks.
- Knowing the theory behind their design is good.

There are different "levels" of PL

- Microcode languages
- Machine languages

  `pub/sum/x86_64/sum.l`
- Assembly languages

  `pub/sum/x86_64/sum.s`
- Intermediate/Internal PLs (e.g., Java byte code)
- High-level PLs (e.g., Java)

Why are there so many PLs?

- Evolution: we've learned better ways of doing things over time.
- Socio-economic factors: proprietary interests and commercial advantage.
- Some PLs are oriented toward special purposes.
- Some PLs are oriented toward special hardware.
- We have diverse ideas about what is pleasant to use.
- Some PLs are easier to learn.
- Some PLs are easier to implement a translator for.
- Newer PLs enforce programmer behaviors that reduce program-maintenance costs.
- Translation technology has improved.
- Some PLs match problem domains.
- Expressive power is important, even if computability power is equal.
- Some PLs have a huge base of previously written programs and libraries.

What makes a PL successful?

- Some are easy to learn (BASIC, Pascal, LOGO, Scheme).
- Some are more "expressive," easier to use once fluent, and "powerful" (C, Common Lisp, APL, Algol-68, Perl).
- Some are easy to implement (Pascal, BASIC and Forth).
- Some can compile to very good (fast/small) code (Fortran).
- Some have the backing of a powerful sponsor (COBOL, PL/I, Ada, and Visual Basic)
- Some enjoyed wide dissemination at minimal cost (Pascal, Turing, and Java).

Why do we have PLs? What is a PL for?

- A PL (or any language) helps determine the way you think, express algorithms, and solve problems.
- Some PLs try to match the user's point of view.
- Some PLs try to match the developer's point of view.
- They are an abstraction of a machine or virtual machine.
- They help you specify what you want the hardware to do without getting down into the bits.

Why study PLs? (1 of 6)

- Knowing multiple PLs makes it easier to choose an appropriate PL:
  - C versus C++, Go, or Rust for systems programming
  - Fortran versus APL or Ada for numerical computations
  - C versus C++ or Ada for embedded systems
  - Common Lisp versus Scheme or ML for symbolic data manipulation
  - Java versus C/CORBA or C/COM for networked programs

Why study PLs? (2 of 6)

- Knowing multiple PLs makes it easier to learn new PLs, because many are similar. This is true for natural languages, too.
- PL concepts have even more similarity. For example, most PLs have iteration, recursion, and abstraction. They might only differ in syntax and semantic details.

Why study PLs? (3 of 6)

- It helps you use your PL more effectively, and understand its obscure features. For example:
  - In C/C++, you can understand unions, arrays, pointers, separate compilation, varargs, and exception handling.
  - In Lisp/Scheme, you can understand first-class functions/closures, streams, exception handling, and symbol internals. For example:

    `pub/etc/Y.scm`

    `pub/etc/Y.sh`

Why study PLs? (4 of 6)

- It helps you understand implementation costs and choose between alternative ways of doing things, based on knowledge of what will be done underneath. For example:
  - Use `x<<1` instead of `x*2`, or `x*x` instead of `x**2`.
  - Use C pointers or Pascal's `with` statement, to factor address calculations.
  - Avoid call by value with large data items in Pascal.
  - Avoid call by name in Algol 60.
  - Choose between computation and table lookup.

Why study PLs? (5 of 6)

- It helps you figure out how to do things in PLs that don't support them explicitly. For example:
  - Old Fortran lacks suitable control structures, but you can use gotos, comments, and programmer discipline.
  - Some PLs lack recursion (e.g., Fortran and CSP), but you can write a recursive algorithm and use mechanical recursion elimination.

Why study PLs? (6 of 6)

- It helps you figure out how to do things in languages that don't support them explicitly. For example:
  - Fortran lacks named constants and enumerations, but you can use variables that are initialized once, then never changed.
  - C and Pascal lack modules, but you can use comments and programmer discipline.
  - Most PLs lack iterators, but you can fake them with functions or member functions.

PL paradigms (1 of 5)

- imperative
  - von Neumann (e.g., Fortran, Pascal, Basic, and C)
  - object-oriented (e.g., Smalltalk, Eiffel, C++, C#, and Java)
  - scripting languages (e.g., Bash, AWK, Perl, Python, JavaScript, and PHP)
- declarative
  - functional (e.g., Scheme, Haskell, ML, Lisp, and FP)
  - logic, constraint-based (e.g., Prolog, VisiCalc, and RPG)

PL paradigms (2 of 5)

- You know a few imperative PLs. These PLs use multiple-assignment variables. An assignment statement can cause a *side-effect*, which influences future computation.
- An imperative program tells the computer *how* to solve a problem in a particular way (e.g., Newton's, or the Babylonian, method for computing the square root of a number).

PL paradigms (3 of 5)

- You are not expected to know any declarative PLs. A declarative program tells the computer *what* problem to solve. It describes all acceptable solutions. This is especially true of the logic PLs (e.g., Prolog). For example:

  $$sqrt(x^2) = x \pm \epsilon$$

  is a declarative program for computing an approximation to the square root of a number. The PL translator is free to choose any way of finding an acceptable solution.
- SQL and Make can be placed in this family.

PL paradigms (4 of 5)

- Functional PLs are based on recursive-function definition and invocation, without side-effects.
- You can see that these families are not always clear cut: You can write a C program in a functional style, if you constrain yourself.

PL paradigms (5 of 5)

- Dataflow PLs model computation with nodes that transform input data streams into output data streams (e.g., Sisal and `jq`). Arcs between nodes model the streams.
- Scripting PLs were originally used to glue-together other programs (e.g., `sh`, the Unix shell). They've morphed into von Neumann and OO PLs.
- OO PLs model computation as the message-passing simulation of real-world entities. The other imperative PLs are identified with John von Neumann.

Euclid's greatest common divisor algorithm

- C

  `pub/ch1/gcd/gcd.c`
- Scheme

  `pub/ch1/gcd/gcd.scm`
- Prolog

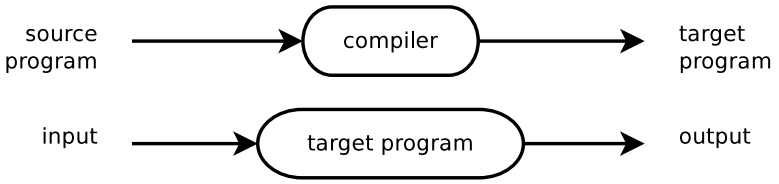
  `pub/ch1/gcd/gcd.pl`

Compilation versus interpretation (1 of 5)

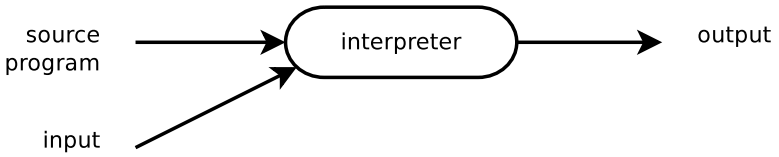
- These are not opposites, and, sometimes, there is not a clear-cut distinction.
- Pure compilation: The compiler translates the high-level source program into an equivalent target program, perhaps in assembly language, and then exits.
- Pure interpretation: After the interpreter translates *each* statement or construct, it immediately executes the translation.
- Compiled programs execute with higher performance: time and space.
- Interpretation is more flexible and provides a better debugging environment.

Compilation versus interpretation (2 of 5)

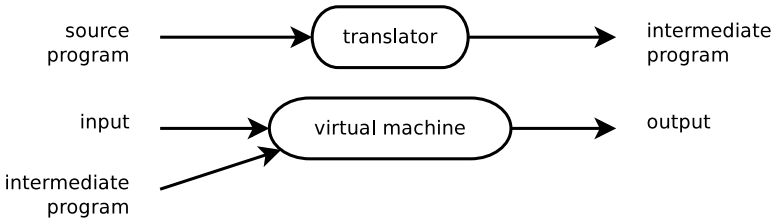
- Compilation



- Interpretation



- Hybrid



Compilation versus interpretation (3 of 5)

- C is typically compiled, Lisp is typically interpreted, and Java is typically a hybrid.
- With compilation, performance (i.e., speed) may be ten times better.
- The Java byte-code interpreter (i.e., the JVM) is a virtual machine.
- The intermediate program (e.g., byte code) is also called intermediate code.

Compilation versus interpretation (4 of 5)

- Compilation does not have to produce machine language for some sort of hardware.
- Compilation is translation from one PL into another, with full analysis of the meaning of the input.
- Our textbook says compilation entails "semantic understanding" of what is being processed.
- Preprocessing is a textual, rather than grammatical, transformation.

Compilation versus interpretation (5 of 5)

- Many compiled languages have interpreted pieces. For example, Fortran has FORMAT statements, and C has printf/scanf function calls:

  `pub/ch1/format.c`

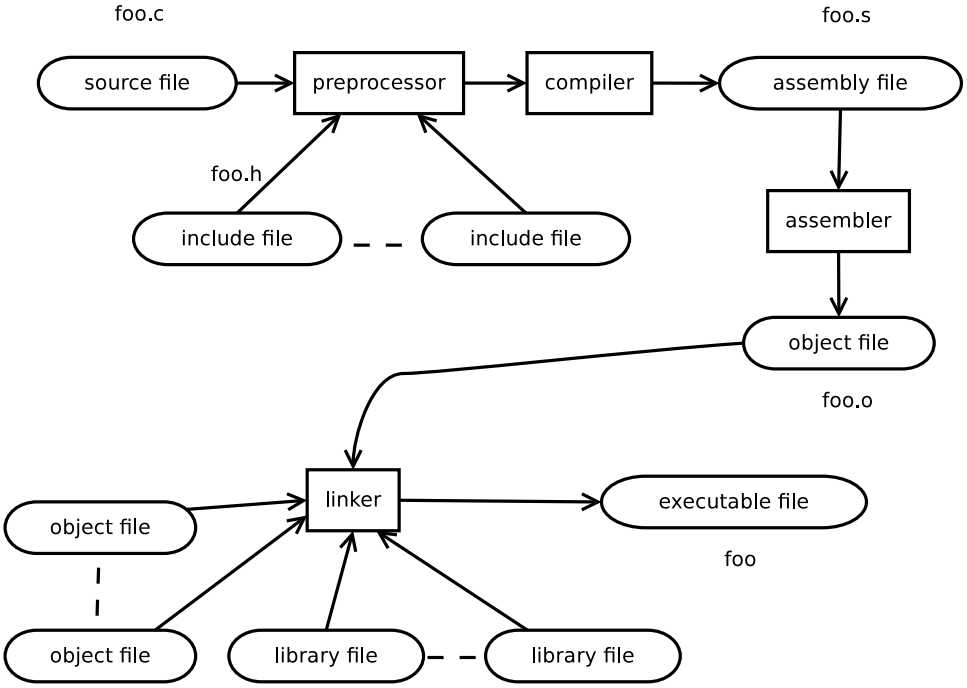
  `pub/ch1/format.f`
- Most use "virtual instructions":
  - set operations in Pascal
  - string manipulation in Basic
- Some compilers produce nothing but virtual instructions (e.g., Pascal P-code, Java byte code, and Microsoft COM+).

Implementation strategies (1 of 7)

- Some translation systems employ a preprocessor:
  - removes comments and white space
  - groups characters into tokens (keywords, identifiers, numbers, and punctuation)
  - expands abbreviations in the style of a macro assembler
  - may identify higher-level syntactic structures

Implementation strategies (2 of 7)

- Many translation systems (e.g., GCC) employ an assembler and linker:

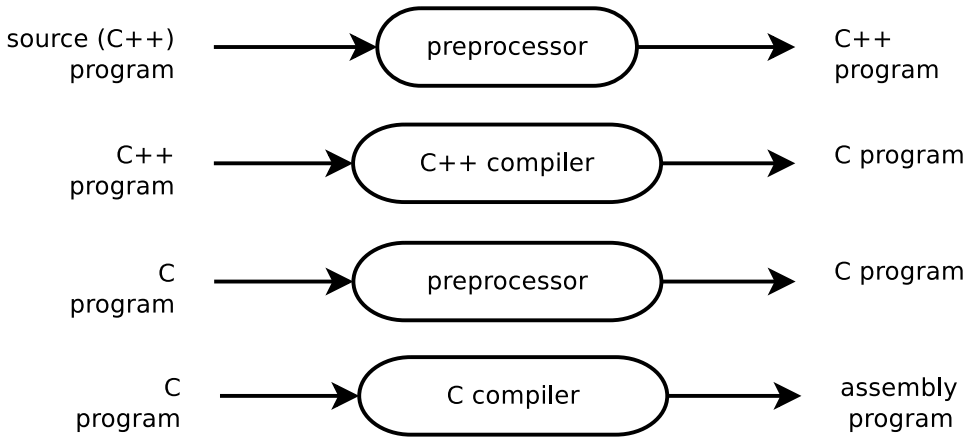


Implementation strategies (3 of 7)

- The C preprocessor has several features:
  - conditional compilation: `#if` and friends
  - file inclusion: `#include`
  - macro definition and expansion: `#define`
  - pragmas
  - line control

Implementation strategies (4 of 7)

- Source-to-source translation:



Implementation strategies (5 of 7)

- Porting a PL like Pascal to a new computer:
  - Participants:
    - A pcode interpreter (assembly)
    - B pcode interpreter (binary)
    - C pascal-to-pcode compiler (pcode)
    - D pascal-to-assembly compiler (assembly)
    - E pascal-to-assembly compiler (binary)
    - F pascal-to-assembly compiler (pascal)
    - G pascal-to-assembly compiler (pcode)
  - Possible steps:

    pub/ch1/bootstrap.pdf

Implementation strategies (6 of 7)

- Compiling typically interpreted PLs:
  - Delay some operations until run time.
  - Such operations are often in a library.
  - In a pinch, revert to the interpreter.
- A virtual machine can perform a second kind of compilation, called Just-In-Time or On-Request (JIT or ORC) compilation. The resulting machine code is much faster.
- A Lisp or Prolog program can invoke the translator, to process dynamically created source code.

Implementation strategies (7 of 7)

- There are two other (opposing) variations:
  - A processor's machine language can be an otherwise virtual machine language (e.g., Java byte code).
  - A processor's machine language might be interpreted internally:
    - Machine-instructions are implemented, within the CPU, in firmware.
    - The interpreter is written in lower-level instructions (i.e., microcode), stored in read-only memory and executed by hardware.
    - This is common, these days, but allowing users to change the microcode is uncommon (e.g., AMD 2900 bit-slice processors).

Unconventional compilers

- structure editors
- pretty printers
- syntax highlighters
- static analyzers
- text formatters
- documentation generators
- database-query interpreters
- debuggers
- silicon compilers

Other programming-environment tools (1 of 2)

- documentation (e.g., `web`, `javadoc`, and `doxygen`)
- design documents (e.g., `dia`)
- testing (e.g., `JUnit` and `dejagnus`)
- edit/compile/debug cycle (e.g., `emacs` and `eclipse`)
- pretty printing and syntax highlighting (e.g., `emacs`, `vim`, `cb`, `vgrind`, and `clang-format`)
- cross referencing (e.g., `cxref`, `web`, `javadoc`, and `doxygen`)
- static analysis (e.g., `lint` and `FindBugs`)
- version control (e.g., `rcs`, `cvs`, `subversion`, and `git`)
- building (e.g., `make`, `ant`, and `maven`)

Other programming-environment tools (2 of 2)

- report generators (e.g., `grg`: GNU Report Generator)
- screen, GUI, and application generators (e.g., `screengen`: Java Screen Generator)
- frameworks and libraries (e.g., X11 and Swing)
- debugging (e.g., `gdb` and `ddd`, `objdump`, and `readelf`)
- dynamic analysis (e.g., `valgrind` and `strace`)
- bug tracking (e.g., `bugzilla` and `GNATS`)

Phases of compilation

- Lexical analysis (scanning) inputs a sequence of characters and outputs a sequence of tokens.
- Syntax analysis (parsing) outputs a syntax or parse tree.
- This is the *front-end/back-end* border.
- Semantic analysis (type checking) outputs a modified tree.
- Intermediate code generation outputs an intermediate representation of the target program.
- Machine-independent code optimization outputs modified intermediate code.
- Code generation outputs a target program.
- Machine-dependent code optimization outputs a modified target program.

Scanning (1 of 2)

- Adjacent characters are grouped into *tokens*: the smallest meaningful units in a program.
- The next phase, parsing, only knows about tokens.
- Example tokens are:
  - numeric literals
  - character literals
  - string literals
  - keywords
  - identifiers
  - operators (e.g., + and ++)
- Comments and whitespace are typically not tokens!

Scanning (2 of 2)

- Scanning is recognition of a *regular language*, the strings of which match *regular expressions*.
- Theoretically, a scanner is a *deterministic finite automaton* (DFA): a machine that changes from state to state as it consumes input symbols (e.g., characters).

Parsing

- Tokens are organized into a *parse tree*, according to the syntax/grammar of the PL.
- Parsing is recognition of a *context-free language*, the strings of which are token sequences that match a *context-free grammar*.
- Theoretically, a parser is a *pushdown automaton* (PDA): a DFA with a stack.

Semantic analysis

- The parse tree is traversed to understand its static characteristics.
- Much of this analysis is type checking and conversion.
- Execution (i.e., dynamic) behavior cannot be checked statically (e.g., array-indexing errors and expression values).
- Theoretically, dynamic semantics is undecidable.

Intermediate code generation

- The parse tree is again traversed to produce instructions for a virtual (i.e., imaginary and idealized machine).
- A so-called *intermediate form* (IF) is often chosen for machine independence, ease of optimization, or compactness.
- Java's byte code can be thought of as an intermediate form.

Machine-independent optimization

- Optimization transforms an IF program into a faster or smaller one.
- We're talking about CPU-independent optimizations (e.g., loop unrolling).
- The term is a misnomer: "improvement" would be better.
- Optimization is optional, often selected by a command-line argument (e.g., `-O`).

Code generation

- Code generation transforms an IF program into assembly language for a particular CPU.
- Some translators don't produce native code, just virtual code.

Machine-dependent code optimization

- Some optimizations use CPU-specific features (e.g., special instructions or addressing modes).
- Another example of target-dependent optimization is the reordering of machine instructions, allowing them to be executed in parallel.

Symbol table

- All phases rely on a symbol table, which records information about the program's identifiers (e.g., variables).
- You can think of a symbol table as a stack of maps. Each map maps an identifier (i.e., a string) to a record containing everything the translator knows about the identifier (e.g., type and address). The stack represents nested scopes.
- A symbol table may be retained, in some form, for use by a debugger, even after compilation.

A scanning example

- Consider this implementation of Euclid's greatest common divisor algorithm:

  `pub/ch1/gcd/gcd.c`
- These are the tokens, one per line:

  `pub/ch1/gcd/tokens`
- For the scanner, the program might as well be:

  `pub/ch1/gcd/ugly.c`

A parsing example

- A PL's parser parses according to a grammar for the language.
- We'll learn more later, but a grammar is a set of rules, with a start symbol. Here's part of a grammar for C:

  `pub/ch1/c-grammar`
- Here's a complete grammar for C:

  `pub/ch1/c-grammar.y`
- Here are complete grammars for Java:

  `pub/ch1/java1.y`

  `pub/ch1/java2.y`

A parse-tree example

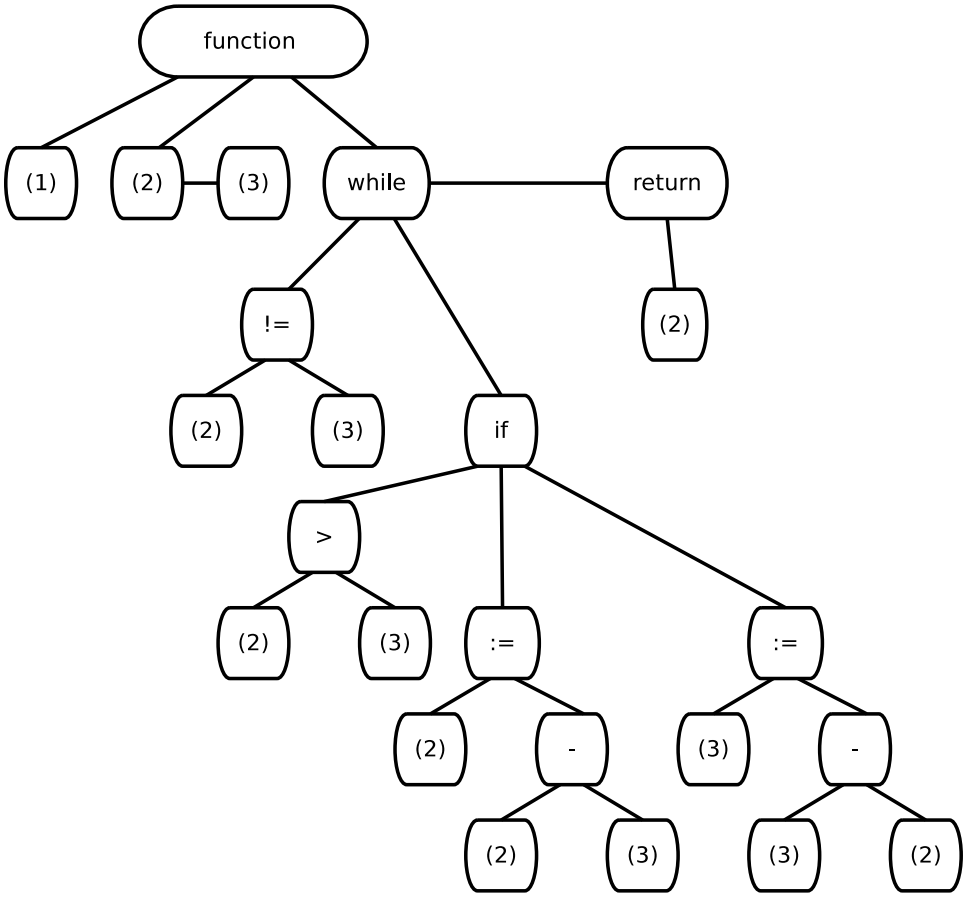
- This is just the top of the tree for the GCD program, according to the grammar:

```
translation_unit
|-external_declaration
| |-function_definition
|   |-declaration_specifiers ···
|   |-declarator ···
|   |-declaration_list ···
|   |-compound_statement ···
|-translation_unit
  |-external_declaration
    |-function_definition
      |-declaration_specifiers ···
      |-declarator ···
      |-declaration_list ···
      |-compound_statement ···
```

A syntax-tree example (1 of 2)

- Since a parse tree is so huge, an *abstract syntax tree* (AST) it sometimes used, instead.
- It omits information that is not needed after parsing.
- It can even be fairly PL independent: an internal form.
- This is for the GCD program's `gcd()` function definition:

A syntax-tree example (2 of 2)



Symbol Table		
1	`gcd`	int int $\rightarrow$ int
2	`a`	int
3	`b`	int

A code-generation example

- The (final) generation phase outputs code in the target language (e.g., assembly).
- Here are examples for a small C program:

  `pub/sum/x86_64/sum.c`
- compiled for my laptop (Intel Pentium):

  `pub/sum/x86_64/sum-gcc.s`
- compiled for my Raspberry Pi (ARM):

  `pub/sum/armv8-a/pi.jpg`

  `pub/sum/armv8-a/sum-gcc.s`

Important definitions (1 of 2)

- An *alphabet* is a finite set of symbols (e.g., characters or tokens).
- A *string* (e.g., lexeme or program) is a sequence of symbols from an alphabet.
- A *language* is a set of such strings.
- The *syntax* of a language is a set of rules that specify the set of strings in the language. For a PL, these strings are the set of "legal-looking" programs.
- The *semantics* of a language is a set of rules that specify the meaning of each string in the language. For a PL, the semantics further specifies which programs are legal, and what a program "does" when it executes.

Important definitions (2 of 2)

- This chapter is about syntax. The syntax of a formal language, like a PL, is specified at two levels, with two different notations: a regular expression (RE) and a (typically context-free) grammar (CFG).
- The part of a translator that recognizes tokens, according to a regular expression, is called a *scanner*. It is a deterministic finite automaton (DFA).
- The part of a translator that recognizes token sequences, according to a grammar, is called a *parser*. It is a push-down automaton (PDA).

Recursive definition of regular expressions (REs)

- An RE is one of the following (in order of increasing precedence):
  - the empty string, denoted by ϵ
  - an alphabetic character
  - two regular expressions separated by | (alternation)
  - two regular expressions juxtaposed (concatenation)
  - a regular expression followed by $*$ (zero or more repetitions)
- Parentheses allow grouping.
- There are other operators (e.g., +), but this is the minimal set.

A regular-definition for numeric literals in Pascal

- `1`
- `123`
- `001`
- `1.2`
- `1.2e3`
- `1.2e00`

`pub/ch2/numbers`

Context-Free Grammars (CFGs)

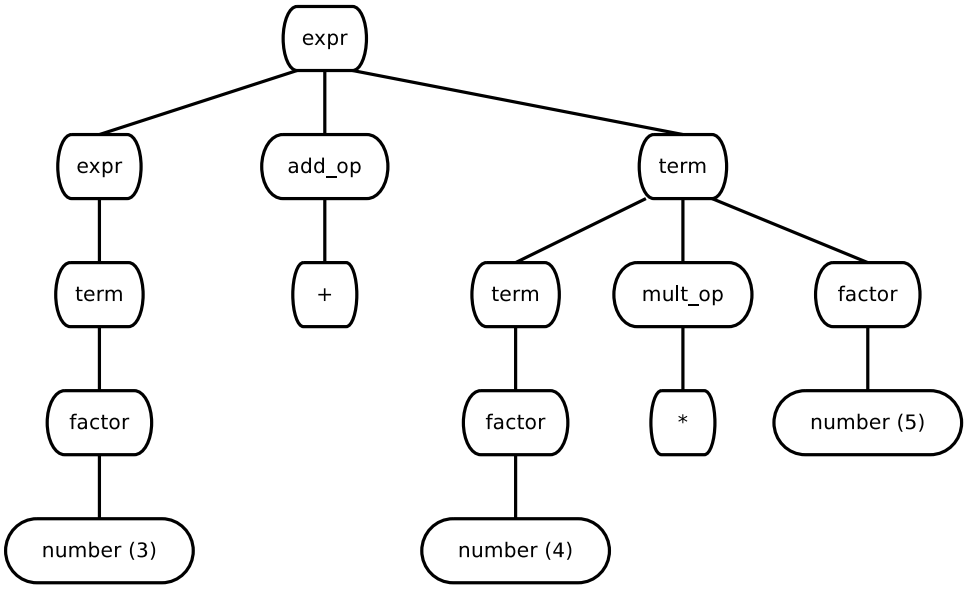
- The PL notation for CFGs is also called Backus-Normal Form or Backus-Naur Form (BNF).
- A CFG has four parts:
  - The *terminal* symbols (aka, *tokens*) are a subset of the grammar symbols. Each represents a string of characters in a string in the language.
  - The *nonterminal* symbols are the rest of the grammar symbols. Each represents a string of terminals.
  - The *start* symbol is a nonterminal.
  - The *productions* each have a nonterminal on the left hand side (LHS) and a string of symbols (terminal and/or nonterminal) on the right hand side (RHS).

A CFG for simple expressions

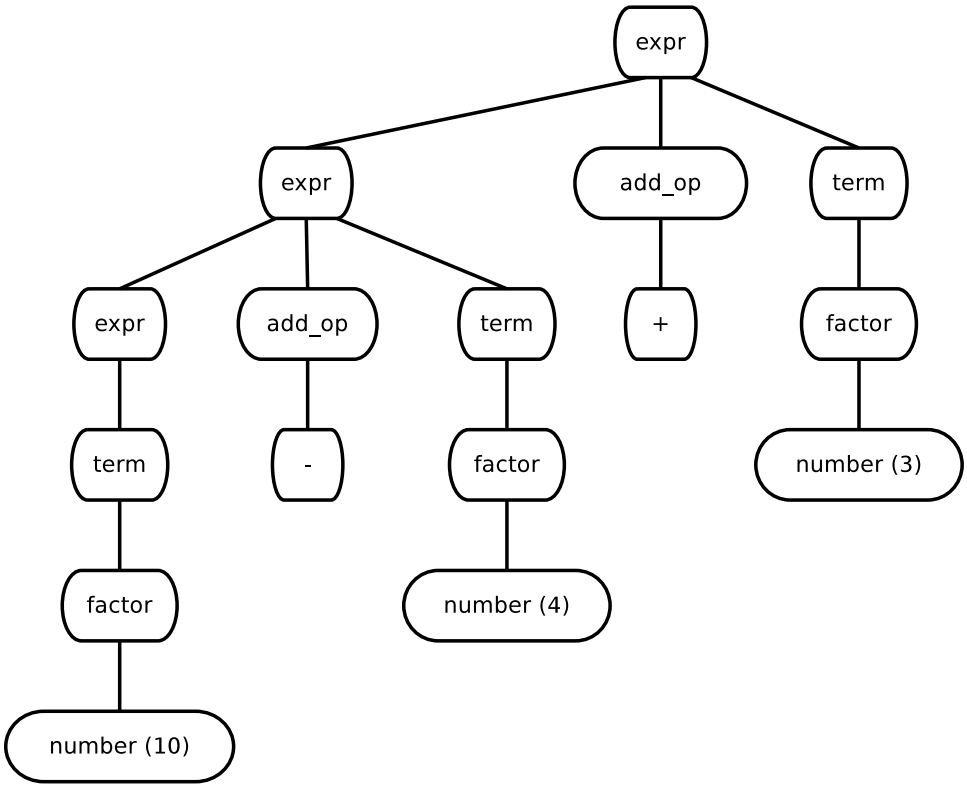
- We want the expected precedence and associativity.
- `x`
- `123`
- `x*(12+-34)`
- `---123`
- `---(123)`

`pub/ch2/exprs`

Parse tree for 3+4*5



Parse tree for `10-4+3`



Derivation of 3+4*5

```
expr => expr add_op term
     => term add_op term
     => factor add_op term
     => number add_op term
     => number '+' term
     => number '+' term mult_op factor
     => number '+' factor mult_op factor
     => number '+' number mult_op factor
     => number '+' number '*' factor
     => number '+' number '*' number
```

Names, scopes, and bindings

- A *identifier* is a (hopefully) mnemonic character string representing something else (e.g., `totalAmount`).
- A *name* is often just an identifier, but can be more (e.g., `x.y.z` or `Foo::bar`).
- A name provides abstraction. We can refer to the higher-level name (e.g., a variable), rather than the lower-level object it represents (e.g., its address).
- Here, "object" has nothing to do with object orientation.
- Abstraction is very important. It allows us to conquer complexity.
- A *binding* is an association between two things, such as a name and the object it names.
- The *scope* of a binding is the textual part of the program in which the binding is active.
- An *environment* is a set of bindings.

Binding time

- *Binding time* is the point at which a binding is created. More generally, it's the point at which any implementation decision is made. Here are some times and bindings:
  - PL design time: syntax, type system, and semantics
  - PL implementation time: CFG, input/output, and numeric precision.
  - program-development time: algorithms, design, and names
  - compile time: intra-module bindings and data layout
  - link time: inter-module bindings and whole-program layout
  - load time: virtual-memory addresses
  - run time: hardware addresses and variable values

Run time

- program-startup time
- module-entry time
- elaboration time: when a declaration is first "seen"
- procedure-entry time
- block-entry time
- statement-execution time
- procedure-exit time
- program-exit time

Binding (1 of 2)

- *Static* binding generally refers to one created before run time.
- *Dynamic* binding refers to one created at or during run time.
- Many of the important differences between PLs are due to when various kinds of bindings are created (e.g., when a variable's type is bound).

Binding (2 of 2)

- Earlier binding can improve performance.
- Later binding can improve flexibility.
- Compiled PLs tend to have earlier binding.
- Interpreted PLs tend to have later binding.

Scope

- All PLs allow a programmer to name data. The name can then be used, rather than the data's address.
- Not all data is named (e.g., some is referenced by pointers).
- PLs have scope rules, which determine name/variable bindings.

Lifetime and storage-management events

- creation of an object
- creation of a binding
- references to a variable, via a binding
- temporary deactivation of a binding
- reactivation of a binding
- destruction of a binding
- destruction of an object

Scope and lifetime

- The time between the creation and destruction of a binding is called its *lifetime* (aka, *extent*).
- The time between the creation and destruction of an object is *its* lifetime. They need not coincide.
- If an object outlives all of its bindings it's *garbage*.
- If a binding outlives its object it's a *dangling reference*.
- The textual region of a program in which a binding is active is its scope.
- Lifetime and scope rules vary significantly across PLs.

Storage-allocation mechanisms

- A PL can have three kinds of allocation:
  - *static*: a fixed-sized chunk of memory, reserved by the compiler/linker
  - *stack*: the CPU's call/return stack
  - *heap*: a managed pool of memory, from which run-time allocations and deallocations can occur (e.g., via Java's `new` keyword)
- They determine where, in a process's virtual address space, data is stored.

Linux `x86_64` Process Memory

- A 64-bit (8-byte) address has 16 hex digits (e.g., `7fff ffff ffff ffff`). It can address any byte in a 64-exabyte region:

  pub/ch3/mem-proc.pdf

- The 256-terabyte user-space region needs the low 48 bits:

  pub/ch3/mem-user.pdf

- For example, run the Bash command:

```
pmap -x $$
```

Static allocation

- instructions and operands
- global variables
- `static` or `own` variables
- literals and constants

Stack allocation

- subroutine formal parameters and return variables
- subroutine-local variables: scalars and aggregates
- temporaries
- block-local variables, which can reuse space

  `pub/ch3/blockvars.c`

Stack frames (1 of 2)

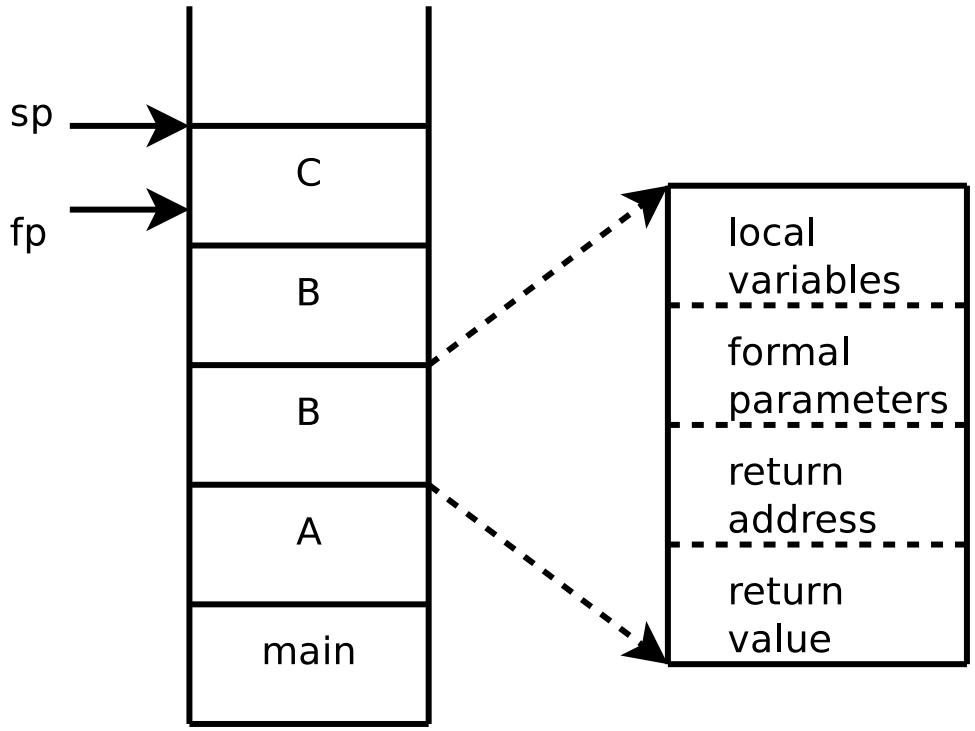
- When a subroutine is called, or when a block is entered, the data pushed onto the stack is called a *frame*.
- Frame content and organization is an important part of a translator's design, but a frame typically contains:
  - formal parameters
  - return values
  - local variables
  - scope information
  - return address
  - saved registers
- A subroutine's code accesses stack data via offsets from the beginning of the frame, which are computed at compile time.
- A frame is bracketed by a *frame pointer* (`fp`) and the *stack pointer* (`sp`).

Stack frames (2 of 2)

- Suppose `main` has called `A`, `A` has called `B`, `B` has called itself, the second call to `B` has called `C`, and `C` is about to call `D`:

  `pub/ch3/frames.c`

- Today, our stack grows upwards, even though a push typically reduces the address in `sp`.



Stack-frame allocation and deallocation

- Caller and callee both help maintain the callee's stack frame.
- The caller's work is the *calling sequence* (e.g., pushing actual parameters).
- The callee's initial work is the *prolog* (e.g., allocating space for local variables).
- The callee's final work is the *epilog* (e.g., deallocating space for the frame).
- Finally, the caller retrieves the return values from just above the top of the stack, according to `sp`. This allows the caller to ignore the return values.

Heap allocation

- During execution, a program can explicitly request (e.g., `malloc()`) an arbitrary-sized block of contiguous memory.
- The allocated block is at least as big as the request. This causes *internal fragmentation*.
- A reference (i.e., pointer) to the beginning of the block is returned.
- The block's lifetime extends until:
  - explicit deallocation (e.g., `free()`)
  - no references to the block remain accessible to the program (i.e., it becomes garbage)
- Since allocations and deallocations are temporally independent, free blocks cannot be completely coalesced. This causes *external fragmentation*.

Introduction to scope rules

- A PL's scope rules determine the declaration bound to an identifier.
- The two broad styles are static (aka, lexical) scope and dynamic (aka, fluid) scope. Static scope is far more common.
- Traditionally, Lisp has dynamic scope, which appears to have been a "mistake." Scheme has static scope.
- Bash has dynamic scope!
- Some PLs have both (e.g., Scheme, Perl, Common Lisp, and Emacs Lisp).
- First, we consider static scope.

Static Scope (1 of 8)

- A *scope* is a portion of a program's source code, of maximal size, in which a binding does not change.
- A scope need not be contiguous.
- In most PLs, a scope is created upon subroutine (and sometimes block) entry:
  - create local-variable bindings
  - temporarily deactivate bindings for "shadowed" variables (creating a "hole" in the outer scope)
- Upon subroutine/block exit:
  - destroy local-variable bindings
  - reactivate previously deactivated bindings

Static Scope (2 of 8)

- An identifier's declaration can be determined by examining a program's source code; you don't have to execute it.
- In other words, a compiler can determine bindings.
- Most PLs use static scope. Essentially, all compiled PLs do.
- Basically, the last declaration of an identifier, seen by the compiler, is the active one.
- Most mainstream PLs (e.g., C, C++, Pascal, and Java) are descendants of the *block-structured* PL Algol 60, which uses static scope.

Static Scope (3 of 8)

- In a block-structured PL, an identifier is known (i.e., bound) in its declaration's block, and in each enclosed block, unless it is redeclared in an enclosed block.
- To resolve a reference to an identifier, check the scope of the referencing block, and then the scopes of statically enclosing blocks, until a binding is found.

Static Scope (4 of 8)

- PLs with modules, classes, and/or packages have additional rules.
- For example, in Java, the scope of a `private static` class variable is the entire enclosing class, even before the declaration, but its lifetime is an entire execution.
- This is similar to an Algol `own` or C `static` local variable.
- C example:

  `pub/ch3/label.c`

Static Scope (5 of 8)

- Class-member visibility in Java:

Modifier	*Class*	*Package*	*Subclass*	*World*
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
none	Y	Y	N	N
`private`	Y	N	N	N

- Which is missing?

Static Scope (5 of 8)

- Recall that formal parameters and nonglobal variables are on the stack, in stack frames.
- Immediately local variables are in the top frame.
- Which frame contains a particular, visible, nonlocal variable? Where, exactly, is that frame? How does recursion complicate this problem?
- Nested blocks in C:

  `pub/ch3/blocks.c`

Static Scope (6 of 8)

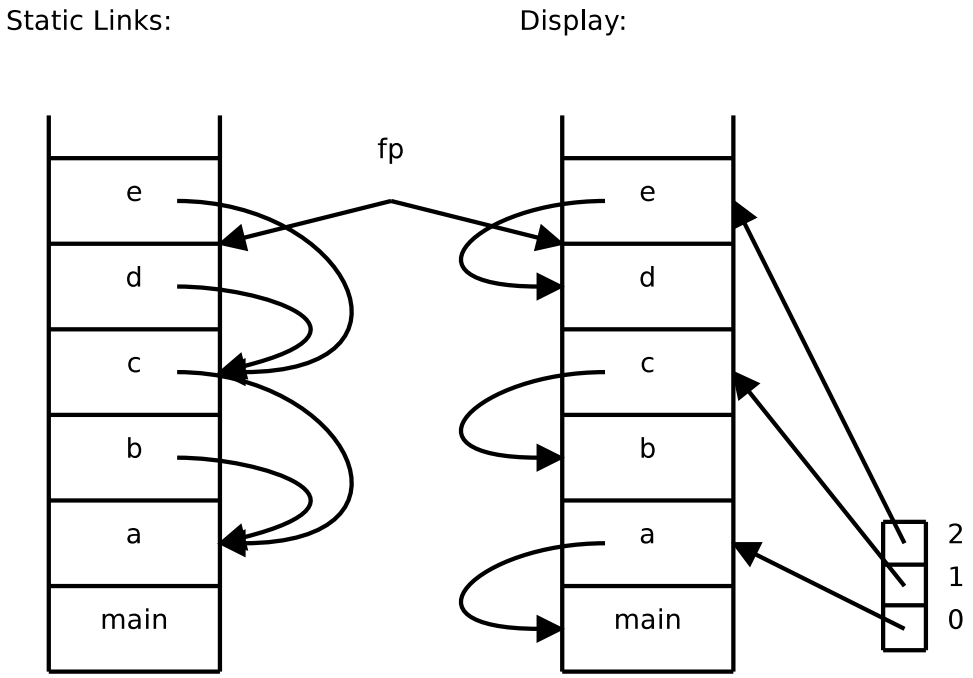
- One solution: *static links*.
  - During compilation, compute the static nesting depth of each declaration (d) and reference (r).
  - During execution, maintain an entry in each frame, pointing to the nearest frame with the previous nesting depth.
  - To accomplish the reference, follow $k = r - d$ static links, and access that frame.
- A better solution: *displays*.
  - During compilation, compute the static nesting depth of each declaration (d).
  - During execution, maintain an array of frame pointers, indexed by depth.
  - To accomplish the reference of a name declared at depth d, access the frame at `display`[d].
  - For a same-depth (e.g., recursive) call, store the previous display entry in the frame, and restore it upon return.

Static Scope (7 of 8)

- GCC extends C with nested function definitions.
- Suppose we have this delightful program:

  pub/ch3/nonlocal.c

  and we are about to call `printf()`.



Static Scope (8 of 8)

- Saying "a binding's scope is the block in which the declaration occurs" sounds simple, but there are complications:

  `pub/ch3/order.c`
- Scheme provides three forms with varying semantics: `let`, `let*`, and `letrec`.
- Some PLs allow declarations to appear anywhere in a block, treating them as if they are all at the beginning of a block, and allowing them to refer to themselves.
- Other PLs (e.g., C, C++, and Pascal), allow "forward" declarations, which support recursive types (e.g., for a linked list).
- Modules (e.g., classes) typically provide a semi-global/semi-local scope mechanism.

Dynamic Scope (1 of 2)

- With static scope, bindings can be determined by looking at the source code.
- With dynamic scope, the execution path must be known.
- Bash examples:

  `pub/ch3/dynamic1`

  `pub/ch3/dynamic2`
- To resolve a reference, use the most recent active binding made at run time.
- Dynamic scope is typically used in interpreted PLs with no compile-time type checking.

Dynamic Scope (2 of 2)

- Suppose Java had dynamic scope:

  `pub/ch3/Dynamic1.java`

  `pub/ch3/Dynamic2.java`

Implementing scope

- For static scope: At compile time, use a stack of maps (each map is for one scope). Generate code that, at run time, manages stack frames. Recall:

  `pub/ch3/blocks.c`

- For dynamic scope: While interpreting, maintain a stack (i.e., an *association list* (a-list)) of bindings. Recall:

  `pub/ch3/dynamic1`

The meaning of names within a scope (1 of 3)

- *Aliasing* is when multiple names refer to the same object:
  - Fortran `EQUIVALENCE` and `COMMON`
  - C `union`
  - pointers and references
  - subroutine parameters
- Examples:

  `pub/ch3/aliasing.f`

  `pub/ch3/aliasing.c`
- Aliasing can save space, by sharing, but stack allocation is better.
- It can subvert a PL's type system (e.g., unions).
- It can be used to create linked and cyclic structures.

The meaning of names within a scope (2 of 3)

- *Overloading* is when a name refers to multiple objects:
  - integer/real arithmetic operators
  - Pascal built-in input/output subroutines
  - Fortran, Algol, and Pascal function-return variables
  - C++ user-defined operator overloading
  - Java user-defined method overloading (not overriding)
- Pascal example:

  `pub/ch3/overload.p`

The meaning of names within a scope (3 of 3)

- Similar polymorphism concepts:
  - *overloaded*: two objects with the same name:

    `pub/ch3/Overloaded.java`
  - *polymorphic*: one object with multiple implementations, via overriding

    `pub/ch3/Polymorphic.java`
  - *generic*: one object that can be instantiated, at compile time, in more than one way

    `pub/ch3/Generic.java`

Binding referencing environments (1 of 3)

- Static scope: Each scope can be represented by a map from identifiers to declarations. Nested scopes can be represented by a stack of scope maps. This symbol table represents a referencing environment. This is all done at compile time.
- Dynamic scope (slow access, fast calls): Bindings can be represented by a list of pairs (i.e., an association list). Each pair is an identifier and its declaration. The a-list is maintained in a stack-like way. This is all done at run time.
- Dynamic scope (fast access, slow calls): Instead we could maintain a table, indexed by identifier. Each entry is a stack of declarations for that identifier.

Binding referencing environments (2 of 3)

- Higher-order PLs allow you to (perhaps):
  - assign a function to a variable
  - pass a function to another function
  - return a function from a function
  - construct a function from "data"
- What bindings should exist when that function is later called?
  - If bindings are fixed early, when the function is assigned, it's called *deep binding*.
  - If bindings are fixed late, when the function is called, it's called *shallow binding*.
  - Neither is always best.

Binding referencing environments (3 of 3)

- When a function is, for example, passed as an actual parameter, it can be bundled with a referencing environment. This self-contained object is called a *closure*. It implements deep binding. The closure is what's assigned, passed, or returned.
- Scheme example:

  `pub/ch3/closure1.scm`
- Emacs Lisp and Bash examples:

  `pub/ch3/closure1.el`

  `pub/ch3/binding.sh`
- What's the required lifetime of an object in the referencing environment of a closure? Unlimited!
- Scheme example:

  `pub/ch3/closure2.scm`

Macro expansion

- This feature is typically provided by a preprocessor.
- A macro is defined, sometimes with arguments.
- A macro can then be referenced, causing its definition to be expanded.
- Macros manipulate PL tokens, although some string operations are supported.
- Macros are tricky to get right, but they allow you to do extra-linguistic things. For example:

  `pub/ch3/macro.c`

Separate Compilation in C

- C "modules" have evolved over time.
- Rules for separate compilation are messy, and coupled to the behavior of the linker.
- The modifier `static`, in a top-level function or variable definition, means it is visible only in the current compilation unit (i.e., source file). Otherwise, it's global.
- The modifier `static`, in a local variable definition, means it has unlimited lifetime.
- The modifier `extern`, in a top-level function or variable declaration, means it is defined elsewhere, perhaps in another source file. In the latter case, the linker will resolve them.
- A function declaration can omit the `extern` modifier, but it's best to be explicit.
- So called "forward declarations" support recursion and recursive types (e.g., linked lists).

Chapter conclusions

- PL features can be surprisingly subtle.
- Design a PL for easy translation, and it'll probably be good in many other ways:
  - easier to understand
  - more and more-portable translators
  - better generated code
  - fewer compiler bugs
  - smaller, faster, cheaper translators
  - better error messages
  - more maintainable user-written programs

Semantic analysis (1 of 3)

- This is mainly just a review.
- Lexical analysis (aka, scanning) transforms a source program, a string of characters, into a sequence of tokens. Each token is a string that matches a regular expression. The symbol table is checked for identifier tokens; new ones are added. Lexical errors are detected (e.g., unterminated string literal).
- Syntax analysis (aka, parsing) transforms a sequence of tokens into a parse tree, according to a grammar. A parse-tree node for an identifier references the identifier's symbol-table entry. As more is learned about an identifier (e.g., its type), its symbol-table entry is updated. Syntax errors are recognized (e.g., missing semicolon).

Semantic analysis (2 of 3)

- The parser may build a complete parse tree. Then, semantic analysis and code generation might occur during a separate traversal of the tree.
- Or, it might interleave these tasks, after each encapsulation unit (e.g., function definition) is parsed.
- The rules that determine whether a program is legal or not, according to the PL's definition, are enforced collectively: by the scanner, parser, and semantic analyzer.
- The semantic analyzer does whatever the earlier phases cannot.
- For example, ensuring that a function is called with appropriate parameters cannot be done with a CFG.
- Semantic analysis has two parts: static (e.g., type checking) and dynamic (e.g., index-bounds checking).
- This chapter considers the static part.

Semantic analysis (3 of 3)

- Semantic analysis transforms a parse tree into whatever the translator's back-end requires.
- This might be a modified parse tree, an abstract syntax tree, a sequence of virtual-machine instructions, or some other internal/intermediate form.
- For example, in GCC's back-end, the remaining pipeline to target code passes through three stages of internal form. A CPU's machine description is specified in a Lisp-like notation:

  `pub/ch4/ia64.md`
  `pub/ch4/aarch64.md`
- Static semantic errors are detected. This is mainly type checking, but includes less obvious errors (e.g., `break` outside of a loop or `switch` statement).
- Some type mismatches (e.g., integer/real) can be fixed with conversions, which might modify the parse tree.

Control-flow paradigms

- Sequencing (e.g., semicolon)
- Selection (e.g., `if`, `switch`)
- Iteration (e.g., `while`, `for`)
- Procedural Abstraction (e.g., function definition and call)
- Recursion: can replace iteration
- Concurrency: independent tasks, communication
- Exception Handling and Speculation: non-local jump
- Nondeterminacy: order is unimportant, but fairness often is

Ordering in expression evaluation (1 of 2)

- An operator can be unary, binary, or ternary.
- An operator can be prefix, infix, postfix, or mixfix (aka, distfix):

```
a+=-(x>y ? x++ : y++);
```

- An operator can be eager or lazy.
- With infix and mixfix operators, we need rules for precedence and associativity. Associativity breaks precedence ties.

Ordering in expression evaluation (2 of 2)

- A lazy operator can be "short circuit" (e.g., `&&`).
- Some PLs have "sequence points", which bound when side effects can happen. For example, in C, assignment is not a sequence point, so the final value of `i` is undefined:

```
int i=0;
i=i++;
```

Precedence and associativity

- Different PLs have different rules:
  - C has 15 levels (too many to remember).
  - Pascal has 4 levels (too few for expressiveness), but logicals are higher than relationals.
  - Fortran has 8 levels, but not many operators.
  - Ada has 6 levels, but `and` and `or` are at the same level.
  - Lisp and Scheme have zero levels (parentheses)
- Caveat codor: use parentheses!
- C rules (`http://www.inf.udec.cl/~leo/precedence.pdf`): pub/ch6/precedence.pdf

Parameter-evaluation order

- In what order are actual parameters evaluated?

  `f(a(),b(),c())`

- Most PLs (e.g., C and C++) don't specify evaluation order of `a`, `b`, and `c`. A translator gets to choose. Java evaluates from left to right. In C:

  pub/ch6/parmeval.c

- Is comma, like semicolon, a binary infix operator, with associativity? Yes, but the inter-parameter comma is not the comma operator:

  pub/ch6/comma.c

- But, there is an important sublety, as shown in:

  pub/ch6/opeval.c

Arithmetic identities

- Commutativity: Are these the same?

```
x=a+b;
x=b+a;
```

  What about?

```
x=a()+b();
x=b()+a();
```

- Associativity: Are these the same?

```
x=a+b+c;
x=(a+b)+c;
x=a+(b+c);
```

Short-circuit evaluation

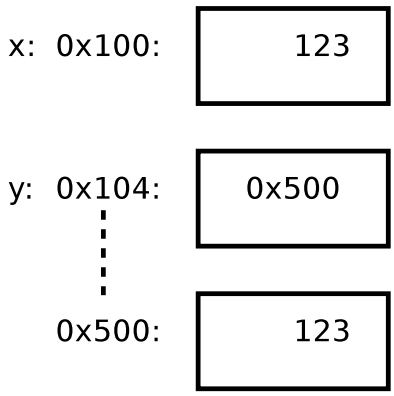
- Most operators are *eager* (aka, strict): all of their operands are always evaluated.
- All of a *lazy* (aka, nonstrict) operator's operands are not always evaluated (e.g., `?:`).
- A lazy logical operator is called a *short-circuit* operator (e.g., `&&`). Its result may depend on the values of only some of its operands.
- This can be very convenient, to avoid null-pointer dereferences and index-bounds errors:

  ```
  if (p && p->foo) ···
  ```

  Pascal forgot to do this:

  `pub/ch6/lazy.p`
- In Pascal, you could put an `if` in the `while`, or use a `for`.

Variables as values or references



- In some PLs, a variable is an identifier bound to the address of a value (e.g., Fortran, C, and Ada).
- In others, a variable is an identifier bound to the address of a location containing the address of a value (e.g., Lisp, Scheme, and Smalltalk).
- Some PLs are hybrids (e.g., Java). Scalar variables hold values. Object variables hold references.
- Some PLs allow both (e.g., Algol, Pascal, and C++).

Values, references, or pointers

- C++ and Pascal allow a variable to be declared as a value, a reference, or a pointer.
- In C++:

  `pub/ch6/values.cc`

Orthogonality (1 of 3)

- Wikipedia: *Orthogonality* in a PL means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the PL.
- Primitives can be used in any combination, but give consistent results.
- This is very important: it leads to simplicity and elegance.

Orthogonality (2 of 3)
examples

- An expression can be a statement:

```
x==y;
```

- A statement can be an expression, which returns a value:

```
x=while (...) ...;
```

- But, this leads to the common mistake:

```
if (a=b) ...;
```

Orthogonality (3 of 3)
counter examples

- Fortran:

```
c legal
  if (x.eq.y) x=0
c illegal
  if (x.eq.y) if (x.eq.z) x=0
```

- Pascal:

```
var i:integer;      { legal }
i:=123;             { legal }
var i:integer:=123; { illegal }
```

- Java, C, and C++:

```
int[] a={1,2,3}; // legal
a={1,2,3};       // illegal
```

Side effects (1 of 4)

- Imperative PLs provide assignment statements or assignment expressions. The most common operator is `=` or `:=`.
- Regardless of whether the operator produces a value, its fundamental purpose is to cause a side effect: it changes the value of one of its operands.
- More generally, a *side effect* is a noticeable change caused by an operation, beyond its return value.
- Many PLs have assignment operators that do more than assignment (e.g., `+=`, prefix/postfix `++`, and Icon's `~===:=`).

Side effects (2 of 4)

- Assignment operators are convenient and can lead to concise code, but they can be tricky (e.g., string copy, in C/C++):

```
while (*p++=*q++);
```

- They are not just abbreviations:

  pub/ch6/assgops.c

- Even Cobol has an arithmetic/assignment statement:

```
ADD Amount TO Total.
```

Side effects (3 of 4)

- Imperative PLs need side effects.
- Other, so called *pure* PL paradigms try to avoid them (e.g., functional PLs).
- Pure PLs can have variables, which you can assign values, *but only once*! You cannot change a variable's value. They are called *single-assignment* PLs.
- Benefits of pure programs:
  - more like mathematics
  - easier to understand
  - easier to optimize
  - easier to prove correct
- Side effects can be convenient (e.g., `rand()`).

Side effects (4 of 4)

- A side effect is especially bad if it causes different expression-evaluation orders to produce different results. For example:

  `pub/ch6/seorder.c`
- This interferes with optimization, portability, and opportunities for concurrency.
- A translator cannot completely check for, and warn about, side effects.

How can we avoid side effects?

- Let's revisit this very realistic example:

  `pub/ch3/label.c`
- We can use a single-assignment pattern:

  `pub/ch6/label.c`
- This pattern is part of the pure functional PL Haskell, which it calls *monads*:

  `pub/ch6/haskell/label.hs`

  You only need to understand the very high-level idea.

Statement ordering: unstructured flow

- In addition to the one-after-another semantics of the semicolon, or newline, early imperative PLs had labels and a `goto` statement:

  pub/ch6/goto.c
- This isn't too surprising. It was like an assembly-language `jmp` instruction.
- The "structured programming" movement of the 1970s replaced the use of `goto` with `if`, `switch`, `while`, `for`, `do`, `break`, `continue`, `return`, etc.

Nonlocal gotos

- Nonlocal `goto` statements are trickier. What happens when you jump out of a block?

  `pub/ch6/goto1.c`
- What about into a block?

  `pub/ch6/goto2.c`
- Exceptions provide a structured alternative.

Statement sequencing

- Semicolon/newline: execute statements in top-to-bottom order.
- This isn't too surprising. Again, it's like assembly-language instructions.
- The sequencing of statement execution is only important when they have side effects.

Selection (1 of 3)

- Selection (e.g., `if`) allows skipping the execution of some statements, based on the value of an expression.
- This is like an assembly-language conditional jump, which is based on a bit in a status register, which is set by a previous arithmetic instruction (e.g., `add` setting a carry bit).
- Most PLs have an `if/then` and an `if/then/else` construct. Many have an `elsif` abbreviation:

  pub/ch6/elif
- Many PLs suffer from the "dangling else" problem.
- Many PLs have a `switch` or `case` construct, with restricted test expressions, which allow efficient (e.g., table-based) code generation.
- Lisp and Scheme have `cond`, which is more like a `switch`, but with arbitrary expressions.

Selection (2 of 3)

- Fortran has a cute one, based on a machine instruction from the IBM 704 (tubes, core memory, but floating point). NASA, 1957:

  `pub/ch6/ibm704.pdf`

  `pub/ch6/computedgoto.f`
- Short-circuit evaluation is part of expressions, not selection statements, but they work together.
- Examples 6.49 and 6.50 (pages 254–255) show long-circuit (e.g., Pascal) and short-circuit (e.g., C) generated code:

  `pub/ch6/ifwoss.txt`

  `pub/ch6/ifwss.txt`

Selection (3 of 3)

- Some PLs have conditionals that return values:

  `pub/ch6/retcond.scm`

- Some PLs have logicals that return values:

  `pub/ch6/retand.scm`

Iteration (1 of 4)

- There are many kinds of loop. A good classification is based on whether the number of iterations can be determined as the loop begins.
- *enumeration* controlled: There is one iteration for each value in a finite set. The number of iterations is sometimes known before iteration begins. These are more interesting, because they include iterators and generators.
- *logic* controlled: The terminating expression eventually becomes true, but you don't know how many iterations that will take, perhaps because of input data.

Iteration (2 of 4)

- Some PLs try to prevent the execution of the loop body from changing the precomputed number of iterations.
- Pascal tries hard:
  - The compiler doesn't let you change the loop variable. You can't even pass it by reference to a function/procedure.
  - Sometimes, changing other variables' values doesn't do what you'd expect, either:

    `pub/ch6/badloop.p`

Iteration (3 of 4)

- Many PLs have loops similar to Java's `for`, `while`, and `do` loops. A `do` loop, like old Fortran loops, perform the test at the bottom.

  `pub/ch6/do.f`

Iteration (4 of 4)

- Some PLs have very nice enumeration-style `for` loops.
- In Awk, "indices" don't even need to be integers:

  `pub/sum/awk/sum3`
- In Python, you don't even see the indices:

  `pub/sum/python/sum`
- Likewise, in Java:

  `pub/sum/java/Sum.java`
- In Bash, you have to explicitly switch from strings to numbers:

  `pub/sum/bash/sum`

Iterators

- Index-free loops are sometimes called *iterators*.
- You can simulate iterators in a PL that doesn't have them, like C++:

  `pub/ch6/iter.cc`
- There are external and internal iterators. External iterators are more flexible, but internal iterators are more convenient.
- With an external iterator, the client performs an action on each element, as in the C++ example, above.
- With an internal iterator, the client tells the iterator what to do with each element. For example, in Scheme and Perl (and many other PLs):

  `pub/ch6/iit.scm`

  `pub/ch6/iit.pl`

Generators

- Icon's generators are a generalization of external iterators. A function can:
  - `return` a value (as usual)
  - generate and `suspend` (i.e., produce) the next value of a sequence
  - `fail`
- Many of Icon's operators work with built-in or user-defined generators to provide *goal-directed evaluation*:

  `pub/ch6/generators.icn`
- Python has generators, stolen from Icon, but not goal-directed evaluation:

  `pub/ch6/generators.py`

Recursion

- Although most PLs provide both, non-imperative programmers tend to prefer recursion over iteration.
- Theoretically, recursion and iteration are equally powerful, in terms of the set of problems that can be solved.
- If a stack frame is used for each call, recursion can be less efficient.
- An optimization is to replace recursion with iteration. In some cases, this transformation can be automated.

Tail recursion

- The easiest kind of recursion to replace with iteration is *tail recursion*, where the recursion is the last thing that happens.
- We saw tail recursion in:

  `pub/ch1/gcd/gcd.scm`

  As opposed to:

  `pub/sum/scheme/sum.scm`

  where the addition happens last.
- Tail recursion can often be achieved by passing partial results as arguments. For example:

  `pub/sum/scheme/sumtail.scm`

Argument evaluation order (revisited)

- Up to now, we've been assuming that a function's actual parameters are evaluated before the call. This is called *applicative-order* evaluation (aka, eager or strict).
- *Normal-order* evaluation delays evaluation until (maybe) later (aka, lazy or nonstrict).
- Purely functional PLs can use normal-order evaluation, transparently:
  - Lisp and Scheme support normal-order evaluation with macros.
  - Haskell uses normal-order evaluation.

Nondeterminancy

- This is when sequencing is deliberately unspecified.
- For example, in some PLs, actual parameters may be evaluated in any order.
- Purely functional PLs offer more opportunities for safe nondeterminancy.

Data Types

- Many PLs require each variable to have a type.
- A variable's type constrains its assignable values.
- Almost all PLs have typed values.
- Curiously, hardware memory is only very rarely typed.
- Does assembly language have types?

What is a type?

- Is it a set of values (e.g., `int`)?
- Recursion alarm: "set" is a type!
- Is it the representation of a value (e.g., a `String` is a zero-terminated contiguous sequence of characters)?
- Is it an equivalence class of objects (e.g., `Students`)?
- Is it a set of operations that can be applied to values of that type?
- Types seem similar to an (abstract) algebra, from mathematics: an algebra comprises sets of elements (e.g., integers) and operations between them (e.g., +).
- A value (i.e., constant) can be thought of as an operation without parameters.

Why do we want types?

- Types provide implicit context, so an appropriate operation can be chosen (e.g., which +). This allows the convenience of overloading.
- Types limit the set of operations that may be applied, allowing errors to be caught statically, rather than during execution. This allows the safety of type checking.

Type system (1 of 2)

- A *type system* has two parts:
  - a mechanism to define types and associate them with certain program constructs (e.g., a subexpression)
  - a set of rules for type equivalence, type compatibility, and type inference
- *Type checking* is the process of determining whether a program obeys a PL's type-compatibility rules.
- A PL is *strongly typed* if it allows a translator to ensure that an unintended operation is not applied.
- A PL is *statically typed* if it is strongly typed and the translator performs the checks at compile time.

Type system (2 of 2)

- Fortran is strongly and statically typed.
- Lisp and Scheme are strongly, but not statically, typed.
- Pascal is strongly and almost statically typed.
- Ada is strongly and statically typed.
- Java is strongly typed. Much checking is static, but some is dynamic.
- C and C++ are "less" strongly typed than Java.

Built-in types

- *Discrete* types have a countable number of values (e.g., integer, boolean, character, enumeration, or subrange).
- *Scalar* types are one-dimensional (e.g., a discrete type or real).
- *Composite* types are composed of other types (e.g., records/structures, unions, arrays, sets, lists, tables, or files).
- What about references and pointers?

User-defined types

- Early PLs only had built-in types (e.g., scalars and arrays). There were no user-defined types.
- Later PLs added type "constructors" (e.g., arrays and records).
- Even later, user-defined types allowed a name to be bound to a type (e.g., `typedef` and `class`).

Orthogonality (revisited)

- We saw orthogonality last chapter.
- An orthogonal PL is easier to understand and has fewer exceptions to its rules.
- A counter-example, in C, is how initializers and assignments are treated differently:

  `pub/ch7/agg.c`
- Orthogonality is important for type systems, too.
- For example, C allows any member of a `struct` to be a `union`, but Pascal requires that a `union` be the last member.
- As another example, many PLs have arrays and/or tables. Some require the index/key to be an integer. Some require it to start at zero or one. Some require the index and/or base type to be scalar, others have no such restriction.

Type checking

- A semantic analyzer enforces a type system:
  - *type equivalence*: Do two values have the same type?
  - *type compatibility*: Can a value of one type be used in a context expecting another type?
  - *type inference*: What is an expression's type, based on the types of its subexpressions?
- Compatibility might require conversion (e.g., integer to real).
- These rules might be enforced statically or dynamically.

Is type equivalence obvious?

- Of course, we are done scanning and parsing. We are analyzing a parse tree or syntax tree.
- Should these four types be equivalent?

```
struct { int a, b; }
struct {
    int a, b;
}
struct {
    int a;
    int b;
}
struct { int x, y; }
```

Type equivalence

- There are two main styles, and PLs typically enforce something in between:
  - *Name equivalence* is stricter and safer. It says that two values have the same type if and only if the type names are identical (i.e., spelled the same).
  - *Structural equivalence* is more relaxed and risky. It says that two values have the same type if and only if the memory representations are the same, after type names are replaced by their definitions, perhaps recursively.
- Name equivalence is more popular, these days:

  `pub/ch7/temp.c`

  `pub/ch7/temp.cc`

Type-equivalence compromise

- PLs balance safety and convenience, and ease of translation, by enforcing something in between.
- Consider this C example:

  `pub/ch7/equiv.c`
- We can use this information to "fix" our C reactor-control program:

  `pub/ch7/temp2.c`

Type conversion and coercion

- When a PL construct expects a value of one type (e.g., integer), but the programmer provides a value of another type (e.g., real), something must be done:
  - The PL can perform an implicit *coercion*.
  - The programmer can perform an explicit *conversion* (aka, *cast*).
- A coercion is automatic.
- A coercion or cast may require run-time computation and/or checking.

Coercions (1 of 2)

- Coercion isn't a new idea, but it is becoming less popular. They can be tricky.
- Even early Fortran has many coercions. By the way, Fortran has implicit variable typing, based on the first letter of a variable's name. I–N means integer, while others are real.
- C has many coercions, too:
  - Any sort of `float` becomes `double`.
  - Any sort of `char`/`short` becomes `int`.
  - Precision might be lost assigning to a variable.
- Modula-2 and Ada don't have coercions.
- C++ has many coercions.

Coercions (2 of 2)

- Some PLs have types that are tricky to coerce. For example, Pascal and Ada have subranges:

  `pub/ch7/range.p`

  Can variables whose types are overlapping subranges be assigned? What about contained subranges?
- As mentioned earlier, sometimes overloading looks like coercion.

Casts (1 of 2)

- There are three kinds of cast:
  - The types are structurally equivalent (e.g., they resolve to pointers). The translator is being told to ignore the rules. No run-time action is needed.
  - The types share a set of values (e.g., C's `int` and `unsigned int`, or Pascal's subranges). At run-time, the value must be checked to be in the shared set.
  - The types have different low-level representations (e.g., C's `int` and `double`). At run-time, the value's bits must actually be converted.
- Watch out for pointers, though! Some CPUs (e.g., ARM) trap unaligned accesses:

  `pub/ch7/pointercast.c`

Casts (2 of 2)

- Prior to Java's generics, containers held objects of class `Object`, which needed to be "down-cast" to the right type:

  `pub/ch7/Cast.java`

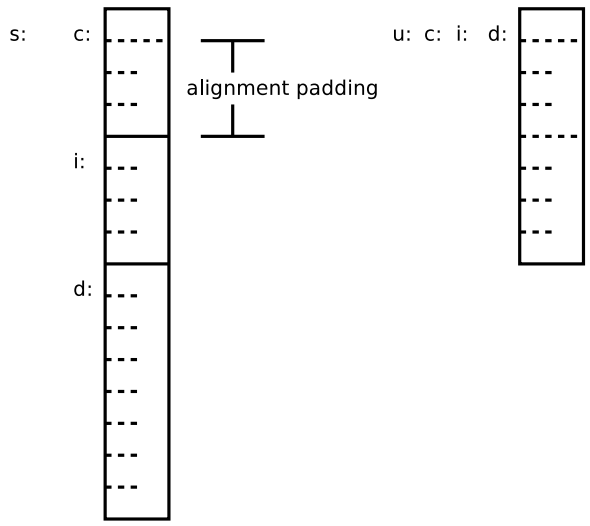
  This requires run-time checking.
- C++ even lets you define your own cast operators.

  `pub/ch7/temp2.cc`
  `pub/ch7/Celsius.h`
  `pub/ch7/Celsius.cc`
  `pub/ch7/Fahrenheit.h`
  `pub/ch7/Fahrenheit.cc`

Records (Structures) and Variants (Unions) (1 of 2)

- A *record* is a heterogeneous data structure: a record value contains *fields* of different types. Intuitively, fields are contiguous and in order, but, to be portable, a program should not assume that.
- A *variant record* is similar, but fields overlap, all beginning at the same address.
- For example, in C:

  pub/ch7/struct-union.c



Records (Structures) and Variants (Unions) (2 of 2)

- In contrast, arrays are typically homogeneous: all elements of a given array have the same (i.e., base) type.
- A `struct` containing function pointers is essentially a C++ class.
- Early PLs didn't have records (e.g., Fortran and Lisp). They became popular in "business" PLs (e.g., Cobol and PL/I), to represent records in disk files.
- Even today, many "scripty" PLs don't have records, except as classes (e.g., Bash, Awk, Perl, and Ruby). Python added records late, as "named tuples." As usual, Python stole the idea from Icon:

  `pub/ch7/record.icn`

Alignment

- The alignment of fields within a record can be tricky:

  `pub/ch7/struct-align.c`

- A translator might rearrange fields, to fill "holes" and save space. There is usually an option to tell it not to.

With statements

- Pascal provides a very nice `with` statement.
- In C, you have to do something like this:

  `pub/ch7/with.c`
- In Pascal, you do this:

  `pub/ch7/with.p`

  which can be faster, too.

Discriminated and free unions

- Since union fields overlap, you need to keep track of which one is "active." A *discriminant* (aka, tag) field, which is typically an enumeration, can help.
- Thus, there are two kinds of union: *discriminated* and *free*. Some PLs (e.g., Pascal) have both kinds. Others (e.g., C) have only free unions, but you can simulate discriminated unions.
- A Pascal variant record, without a discriminant:

  `pub/ch7/union-free.p`
- A Pascal variant record, with a discriminant:

  `pub/ch7/union-disc.p`
- A C union without a discriminant:

  `pub/ch7/union-free.c`
- A C union with a discriminant:

  `pub/ch7/union-disc.c`

Unions and type-system subversion

- Unions have several purposes:
  - share space (ala, Fortran COMMON)
  - a kind of polymorphism, especially for pointers
  - break types (e.g., int to bits)
- Discriminated or free unions can be used to subvert type checking. For example:

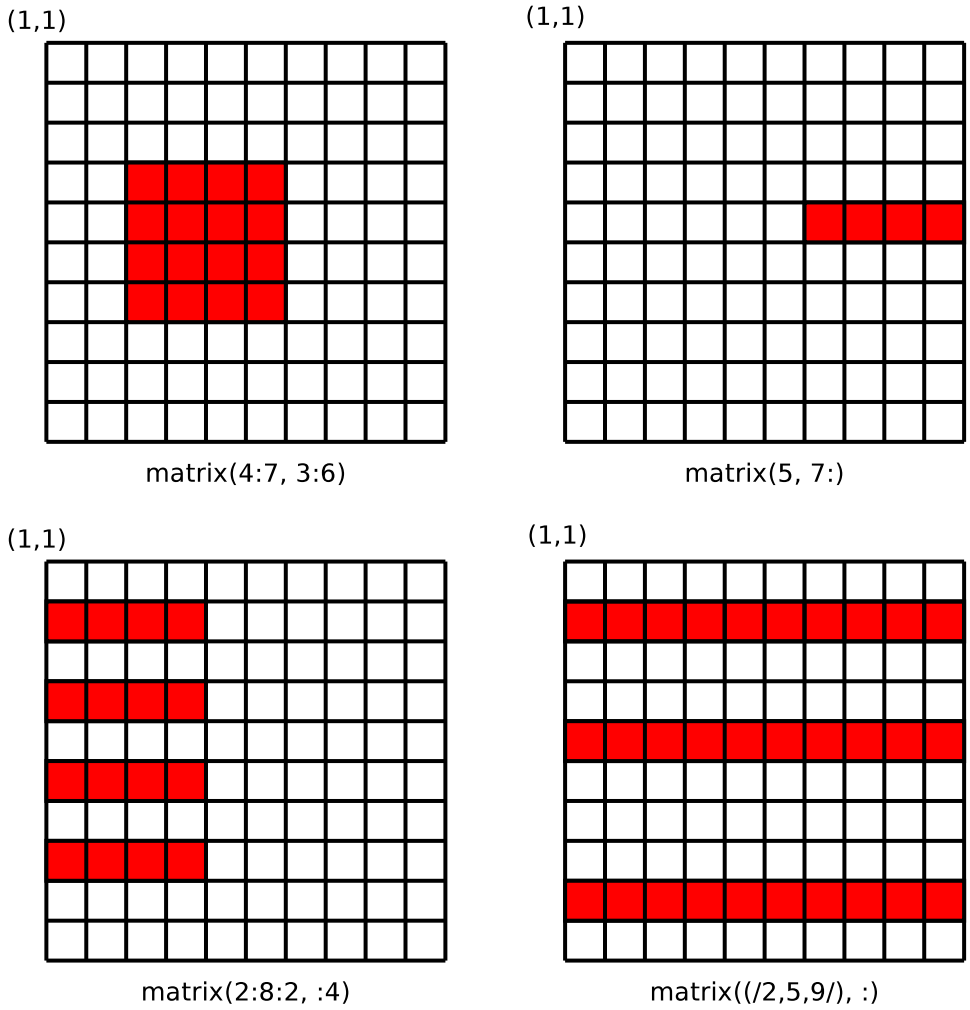
  `pub/ch7/union-break.c`
- Suppose we want to prevent that:
  - The run-time system can "uninitialize" fields when the discriminant is changed.
  - The PL can require assignment of the entire variant, including the discriminant, as in Ada.

Arrays (1 of 2)

- Almost all PLs have arrays. They are the most important aggregate type.
- As mentioned earlier, arrays are typically homogeneous: all elements of a given array have the same (i.e., base) type.
- Square brackets or parentheses are typically the array-index operator. They are also typically used to declare/define an array.
- In compiled PLs (e.g., Fortran, Pascal, and C), an index is typically some kind of an integer, starting at zero or one, and an array's size/shape is often bound at allocation time.
- In many interpreted PLs, an array maps an *index type* to an *element type*, and an array can "grow" during execution.

Arrays (2 of 2)

- Some PLs support array *slices*, which are a rectangular part of an array.
- Fortran uses one-origin *column-major* layout. Every other PL uses *row-major* layout. Here are some Fortran slices:



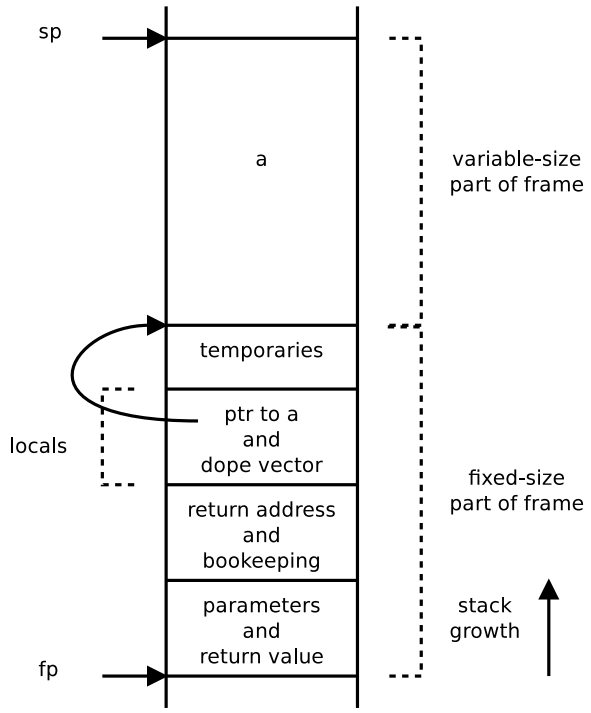
Array allocation (1 of 3)

- If an array's size/shape are static (i.e., bound at compile time):
  - If its lifetime is global, its space can be allocated in static memory.
  - If its lifetime is local, its space can be allocated on the stack.
  - Otherwise, it goes in the heap.
- If an array's size/shape is dynamic (i.e., bound at allocation time):
  - Size/shape information is made available, at run time, in a *dope vector*.
  - If its lifetime is local and it is allocated as the function is called, its space can be allocated on the stack (see next slide).
  - Otherwise, it goes in the heap.

Array allocation (2 of 3)

- Ada and newer C allow a local array's size to be bound, and its space to be allocated, as the function is called, if its size is known at that time:

  `pub/ch7/dynarr.c`



Array allocation (3 of 3)

- If its size/shape is even more dynamic, perhaps varying during execution, its space must be allocated in the heap.

Array layout

- Within an array, elements are typically stored contiguously. Higher-indexed elements are typically at higher addresses, even with a down-growing stack.
- However, with two-dimensional (or more) arrays, there are two methods:
  - *Row major:* elements in a row are contiguous.
  - *Column major:* elements in a column are contiguous (only in Fortran).
- Row major order makes:

```
int a[ROWS,COLS];
```

  the same as:

```
int a[ROWS][COLS];
```

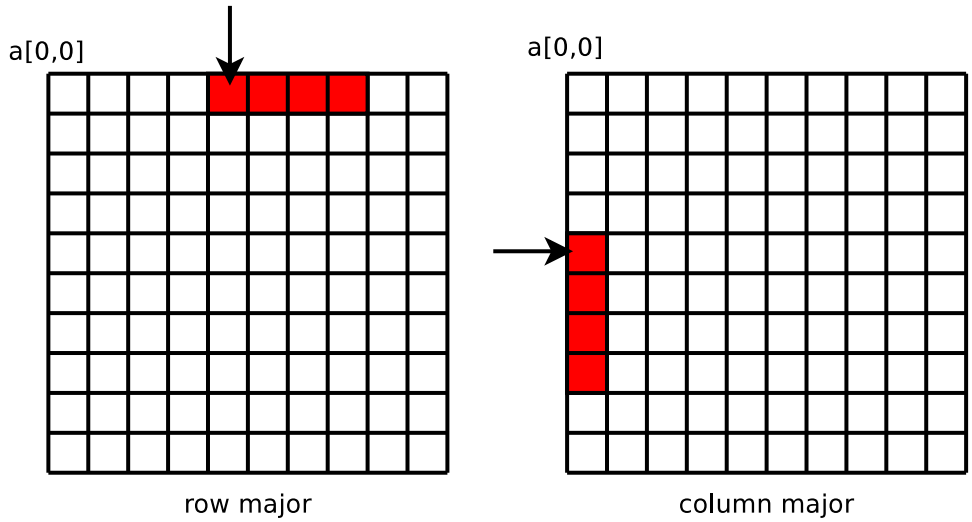
- Apparently, Fortran chose column major due to the IBM 704's architecture.

Array traversal and caches

- Modern processor data caches require a high-performance program to match element traversal with memory layout. For example, with row major, nest column traversal within row traversal:

```
for (int row=0; row<rows; row++)
    for (int col=0; col<cols; col++)
        ···a[row][col]··· // or a[row,col]
```

- Assuming 64-bit elements, 32-byte cache lines, and typical alignment, the second cache line is shaded:



Other layout strategies

- When the size/shape of an array is bound at compile time, it must be large enough to hold the largest expected amount of data. This wastes space.
- The actual amount of data is typically recorded in another variable or indicated by a termination value in the array. C *rectangular* array:

  `pub/ch7/days1.c`

  `pub/ch7/days1.ps`

  C *ragged* array:

  `pub/ch7/days2.c`

  `pub/ch7/days2.ps`

Element-address computation

- Since programs that use arrays tend to be computation intensive, and have stringent performance requirements, efficient element-address computation is important.
- The goal is to do as much at compile time as possible.
- Consider this Pascal 3-d array program, where we want to compute the offset of `a[i,j,k]` from the start of the array:
  `pub/ch7/array3d.p`
- We can perform the whole computation at run time, or use compile-time constants to reduce the amount of arithmetic.
- The dope vector is $< s_1, s_2, s_3 >$.

Character strings

- The importance of strings is often overlooked.
- Early Fortran did not have strings, but, eventually, characters could be converted to integers, via Hollerith constants.
- In many PLs, a string is just an array of characters. However, there are several representations:
  - fixed length, padded with spaces
  - zero-byte terminated
  - zero-bit terminated
  - length with character sequence
  - discontiguous substrings
- Multibyte characters (e.g., Unicode) complicate representation.

String-oriented PLs (1 of 2)

- Icon, like Snobol, has extremely powerful string operations:

  `pub/ch7/scan.icn`
- The program reads itself as input, writing whitespace-separated "tokens" on separate lines.
- The "augmented string scanning" loop assigns successive suffixes of `s` to itself.
- Icon uses one-origin indexing. An index refers to positions *between* characters. Position zero is just after the end, position -1 is just before the last character, and so on. Python uses this scheme, too.

String-oriented PLs (2 of 2)

- Icon's string-scanning operations are an alternative to regular expressions.
- Many PLs, especially scripting PLs, support regular-expression matching.
- Most PLs use single or double quotes to delimit string literals, which may contain *escape sequences*.
- String concatenation operations vary remarkably:

  `+` in Java, C#, Algol
  `||` in Icon
  `,` in Smalltalk
  `.` in Perl
  `&` in Ada
  `++` in Haskell
  `//` in newer Fortran
  juxtaposition in Snobol, Awk, Bash
  (and C/C++ for literals)

Sets

- A *set* is a collection:
  - no duplicates
  - unordered
  - homogeneous?
  - finite size?
- Some PLs only have *bitsets*, allowing operations like intersection, union, and membership to be implemented efficiently with bitwise logical instructions. Such sets typically have a (lame) size limit.
- A *bag* (aka, *multiset*) allows duplicates.
- A *sequence* (aka, *list*) provides order.

Pascal sets

- Our text says Pascal introduced sets to PLs, in 1970, but Setl had much better sets in 1969:

  `pub/sum/setl/sum.setl`
- Pascal sets are limited in size (e.g., 255 elements). Thus, some operations can be implemented with a single machine instruction.
- Pascal forgot to have a way to iterate through a set:

  `pub/ch7/sets.p`

Sets in other PLs

- Programmers often use lists to simulate sets, but you don't get set operators (e.g., union and intersection).
- Icon and Python also have (real) sets.
- Icon's sets are character sets, called *csets*, which are mainly used in string scanning:

  `pub/ch7/scan.icn`

  `pub/ch7/sets.icn`
- Python sets are more flexible. They can contain (some) nonscalar values:

  `pub/ch7/sets.py`

  However, a set cannot contain a set.

Pointers and recursive types

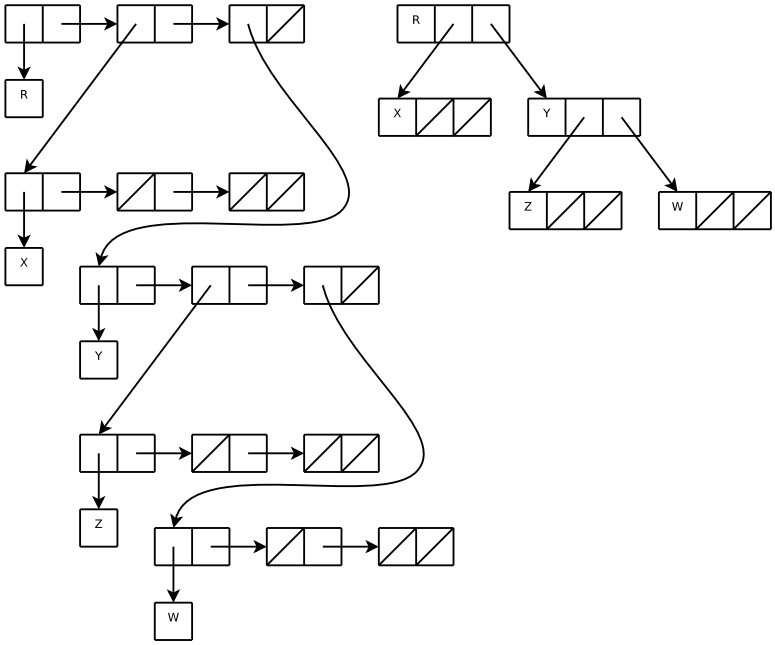
- A *pointer* is a variable whose value is a the address of, or reference to, another value. Untyped pointers were introduced in PL/I, from IBM, in 1964.
- A *recursive type* is one whose values contain pointers to values of the same type (e.g., a linked list).
- Operations on pointers include:
  - construction (e.g., `&` and `new`)
  - assignment (e.g., `=` and `:=`)
  - dereferencing (e.g., `*` and `^`)
- Some PLs (e.g., Pascal) restrict pointer access to the heap. Others (e.g., C) allow a pointer to be bound to any address.
- Some PLs (e.g., C) allow pointer arithmetic (e.g., `*p++`).
- Some PLs (e.g., Lisp, Scheme, and Java) pretend not to have pointers.

Pointers and trees

- In Lisp and Scheme, programs and data are lists, which can look more like trees. For example:

  `pub/ch7/list.scm`

- The value of `list` might be either of:



A slash represents null. Cells can be distinguished from atoms, perhaps by address.

C pointers and arrays

- In Java, an array is much like an OO object.
- In C/C++, arrays and pointers are very similar. Here are some crazy C examples:

  `pub/ch7/arrptr1.c`

  The cute macro prints sizes. Note that the parameter to `f3` is a pointer, but its local variable is an array. Wow!
- Here's some more:

  `pub/ch7/arrptr2.c`

  Note the last four lines of `f`. Wow! Wow!
- A definition with all sizes allocates an array; omitting the "last" size allocates a pointer.
- To compute element addresses, a declaration must have a size for every dimension but the last.

  `pub/ch7/arr.c`

Pointer problems

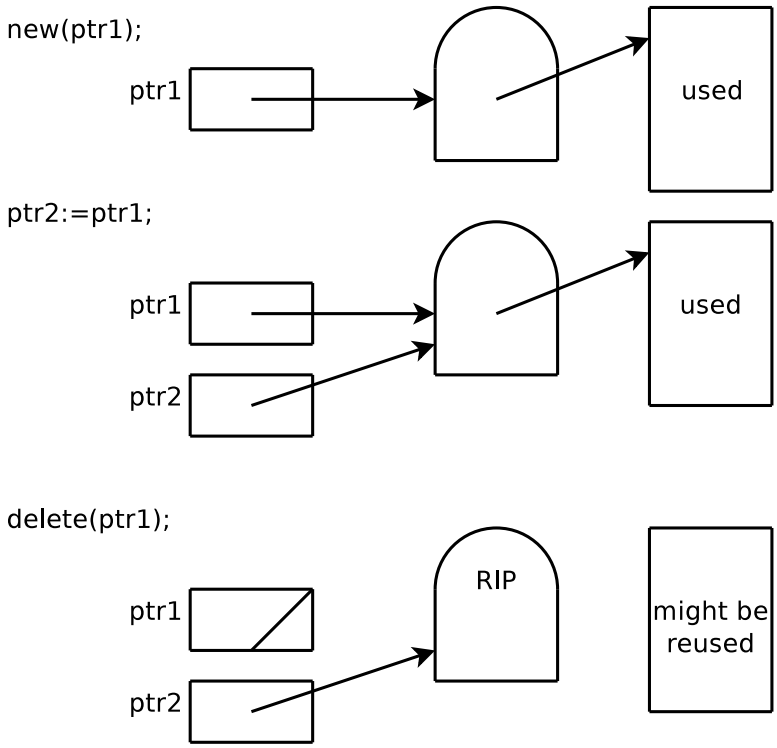
- A pointer that references a deallocated object is called *dangling*. Such deallocation can happen in two ways:
  - for heap objects, explicitly, with an operator or function like `dispose`, `delete`, or `free()`:

    pub/ch7/dangle-heap.c
  - for stack objects, implicitly, by leaving the allocating block:

    pub/ch7/dangle-stack.c
- This is a ubiquitous problem.

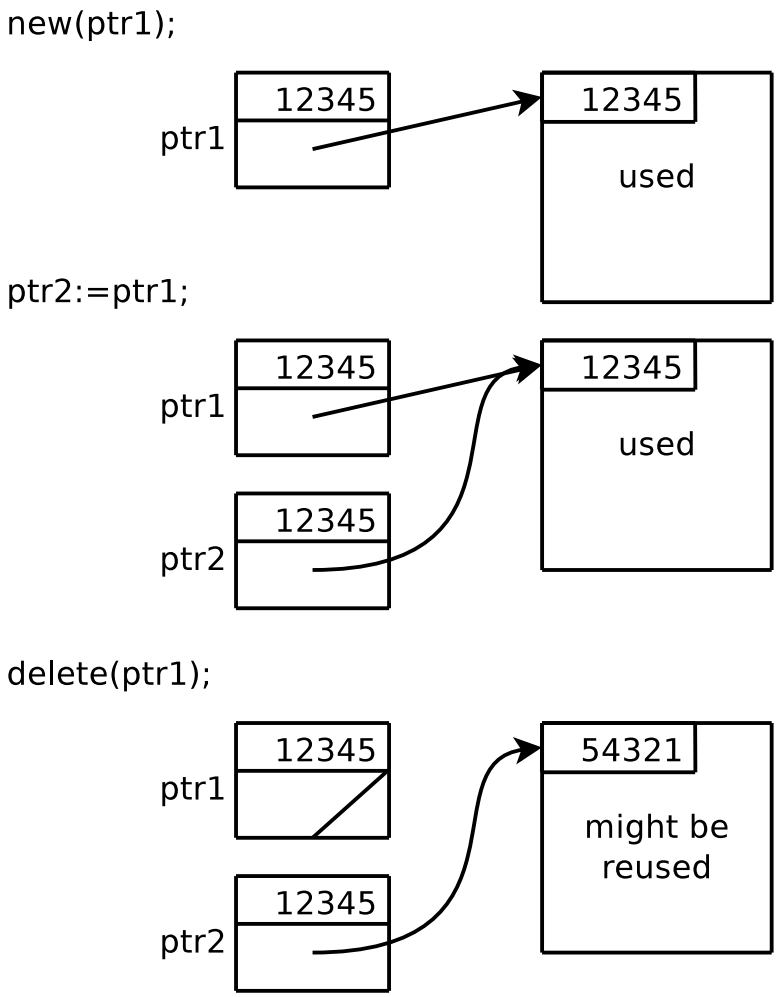
Run-time dangling-reference solutions (1 of 2)

- Tombstones:



Run-time dangling-reference solutions (2 of 2)

- Locks and Keys:

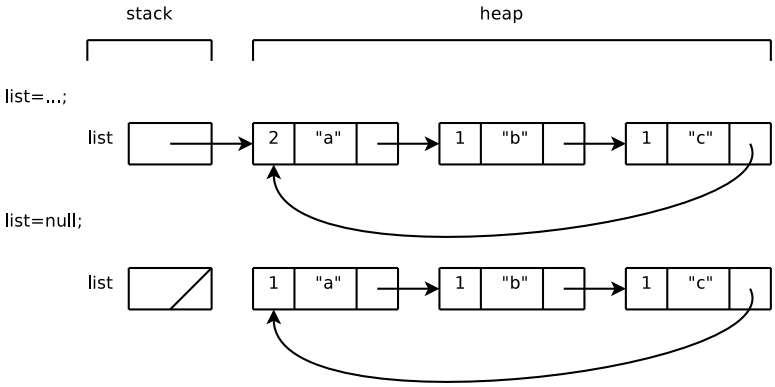


Garbage collection

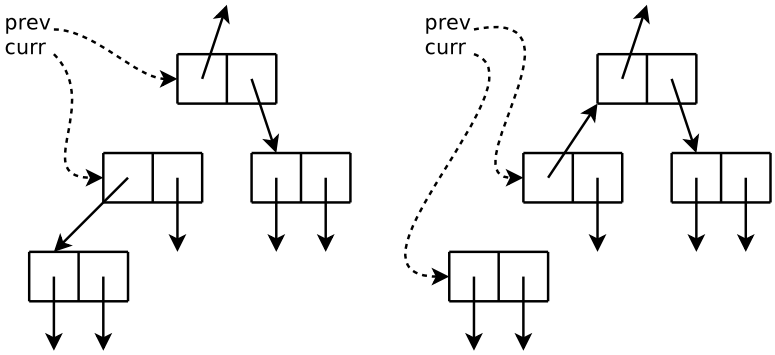
- Programmers aren't very good at explicitly deallocating heap objects. Automatic deallocation and reclamation of heap objects is called *garbage collection*.
- There are two main techniques:
  - *Reference counting* adds a field to each object, whose value is the number of references to that object. When it becomes zero, the object can be returned to the free pool.
  - *Tracing* traverses memory twice: First, each object referenced by any other object is "marked." Then, unmarked objects are "swept" into the free pool.

Garbage-collection pitfalls (1 of 2)

- Reference counting permits incremental collection, which avoids what appear as unpredictable "hangs" or "freezes," but cyclic structures cause problems:



- Tracing usually occurs when memory is exhausted, but a naive traversal requires memory. *Pointer reversal* is a clever trick to conserve memory:



Garbage-collection pitfalls (2 of 2)

- Garbage-collection algorithms are tricky and hard to test. A latent bug will be catastrophic and affect all applications. Indeed, such algorithms have been the subject of program-correctness proofs.

Alternative to explicit deallocation and garbage collection

- Both explicit deallocation and garbage collection have drawbacks.
- So, don't forget about `alloca()`, dynamically allocated stack memory in C and C++:

  `pub/ch7/alloca.c`

Lists

- Lists can be built with records and pointers.
- PLs that provide dynamically allocated data types, typically provides lists.
- A list is defined recursively:
  - an empty list (e.g., *nil*)
  - an *atom* (e.g., 123)
  - a *pair* of lists
- Lists are ideal for functional and logic PLs.
- In Lisp and Scheme, a program is a list, and can be constructed, translated, and executed at run time:

  `pub/ch7/eval.scm`
- Many imperative PLs also have lists.

Dotted pairs

- Most PLs prevent the construction of an *improper* list: one whose tail is not a list. Lisp and Scheme allow them, representing them as a *dotted pair*, with a dot between the head and tail.
- This Scheme example shows that the tail of a list need not be a list:

  `pub/ch7/pair.scm`

Comprehensions

- Some functional PLs provide list *comprehensions*, adapted from set theory. They are rather declarative ways of defining complex lists. For example, in Python:

  `pub/ch7/list.py`

Files and input/output

- Input/output (I/O) facilities allow a program to communicate with the outside world.
- Typically, a library or operating system handles I/O.
- However, some PLs (e.g., Pascal) directly support files as a data type:

  `pub/ch7/files.p`
- Some Pascal translators support external (disk) and internal (memory) files. Ours only has external files.

Equality testing and assignment (1 of 2)

- Scalar equality and assignment is straightforward. All the bits are either compared or moved.
- Aggregate values are trickier.
- Consider assigning an array or tree from one variable to another:
  - Is it just a pointer/reference assignment, like in Java?
  - Is it a shallow, one-level, copy?
  - Is it a deep, complete, copy? This is needed in a remote procedure call, for example.

Equality testing and assignment (2 of 2)

- Cobol has a `MOVE CORRESPONDING` statement, for assigning parts of a record to a similar record, by field name. Is this a good idea?
- Lisp and Scheme provide multiple predicates for comparing aggregates (e.g., `eq?`, `eqv?`, and `equal?`). These are like Java's `==` operator and `equals()` method.
- Even Java strings are weird:

  `pub/ch7/StrEq.java`
- Furthermore, do we really want to compare the *whole* structure? Consider a stack implemented with an array. Do we care about the elements above the top?

Subroutines and Control Abstraction

- Chapter 6 was about control flow.
- Chapter 7 was about data abstraction.
- This chapter is about control abstractions.
- Note that object-oriented classes allow abstraction of both control and data.
- We'll use the term *subroutine* for any encapsulation of computation: methods, operations, operators, procedures, functions, subroutines, iterators, generators, coroutines, etc.
- We'll restrict our discussion to single-process and single-thread execution.

Where are we?

- Control flow:
  - seqencing
  - selection
  - iteration
- Data abstraction:
  - types
  - variables
  - allocation and deallocation
- What else do we need, for subroutines?
  - actual parameters
  - call sequence
  - formal parameters
  - local variables
  - return values
  - return sequence

Parameter allocation: We saw $\cdots$

- that memory can be allocated statically, on the stack, and in the heap.
- that formal parameters and local variables can usually be allocated on the call/return stack.
- that an actual parameter provides an initial value for a formal parameter, which might be a reference or pointer.
- stack-frame organization [slide 3.14].
- that a static chain or display can implement static-scope rules [slide 3.25].
- We have not seen: who does what, and when.

Calling sequences

- There are four parts to executing a subroutine:
  - The caller prepares for the call (e.g., evaluates actual parameters and puts them onto the stack).
  - *Prolog*: The callee prepares to execute its body (e.g., allocates its stack frame).
  - *Epilog*: The callee prepares to return (e.g., deallocates its stack frame).
  - The caller prepares to resume its body (e.g., obtains return value(s)).
- Thus, the caller and callee must agree on stack-frame structure.

Typical stack-frame structure

- Our textbook's Figure 9.2, on page 416, shows a frame containing, in decreasing-address order:
  - parameters
  - return address
  - saved fp
  - saved registers and static link (or display entry)
  - locals
  - temporaries
- Questions/complaints:
  - Which registers?
  - Where's the return value?
  - The caller must deallocate parameters?

GCC (4.8.2) x86_64 stack-frame structure

- Let's use this program to deduce a real stack-frame layout:

  `pub/ch8/frame.c`
- This is typical cleaned-up output:

  `pub/ch8/frame.txt`
- This is dense, tricky, code. I worked on it for hours. Let's discuss some details...

Registers

- A code generator tries to keep data in registers, rather than memory. Is that always okay? No, that's why C has the `volatile` keyword.
- Some registers are typically "reserved" for subroutine calls (e.g., a stack pointer, a frame pointer, the first few actual parameters, and the return value).

Parameter passing

- Almost all PLs allow parameters to be passed to a subroutine, and zero or more values to be returned. Basic does not.
- Different PLs pass parameters in different ways (aka, *modes*).
- Many PLs (e.g., Pascal and C++) provide multiple modes.
- There are four basic modes:
  - by-value (aka, copy-in)
  - by-reference (aka, by-sharing)
  - by-value/result (aka, copy-in/copy-out)
  - by-name (aka, by-need and by-closure)

Pass by-value

- At call time, the value of the actual is copied and assigned to the formal.
- At return time, the formal is simply deallocated.
- This is the only mode provided by C and Java, but people will argue.
- This mode is also provided by Pascal and C++, as demonstrated by formal `a`:

  `pub/ch8/modes.p`

  `pub/ch8/modes.cc`

Pass by-reference

- At call time, the address of the actual is assigned to the formal.
- During the call, references to the formal are automatically dereferenced, thereby accessing the actual.
- At return time, the formal is simply deallocated.
- This mode is provided by Pascal and C++, as demonstrated by formal `b`:

  `pub/ch8/modes.p`

  `pub/ch8/modes.cc`
- This mode can be simulated by pointers, as demonstrated above by formal `c`. Note that the pointer is passed by-value.

Pass by-value/result

- At call time, the value of the actual is copied and assigned to the formal.
- At return time, the value of the formal is copied and assigned back to the actual. Then, the formal is deallocated.
- This mode is provided by, and is the only mode provided by, traditional Fortran. It tries to simulate by-reference. Newer Fortran uses by-reference.
- This example produces a different result for by-value/result and by-reference:

  `pub/ch8/vr.f`

Pass by-name

- The actual is not evaluated until the formal is referenced.
- The actual is evaluated each time the formal is referenced.
- This mode is provided by Algol and Simula, and demonstrated by Scala:

  `pub/ch8/ByName.scala`
- It is similar to textual substitution in macro processors (e.g., CPP and M4), and passing a closure in Lisp and Scheme.
- This lazy evaluation lets you build your own control structures (e.g., loops).
- This mode is implemented by passing a reference to a function, a *thunk*, which can be called, zero or more times, to produce the parameter's value. The value may be *memoized*.
- Try to write a general-purpose swap function with by-name:

  `pub/ch8/swap.c`

A menagerie of parameterish features (1 of 7)

- C, C++, Java, and other PLs allow a parameter's mode to be modified with a keyword like `const` or `final`, making the parameter read-only.
- This might seem silly for by-value parameters, but:
  - it promotes self-documenting code
  - it allows the compiler to catch mistakes
  - for a pointer or reference, you can make the pointed-at object read-only
- Ada extends this idea to `in`, `out`, and `in out` modes. This allows a compiler to choose a good implementation, perhaps based on the size of the parameter.

A menagerie of parameterish features (2 of 7)

- Default/optional parameters can drastically simplify a subroutine call. In C++:

  `pub/ch8/default.cc`

- In contrast, a subroutine may take a variable number of parameters. C and C++ use the ... "keyword" and a kludgy macro-based protocol:

  `pub/ch8/args.c`

- Java and C# approximate this by using ... to bundle a homogeneous sequence of parameters into a list, in a type-safe way:

  `pub/ch8/Args.java`

A menagerie of parameterish features (3 of 7)

- Scheme is similar, but note the subtlety:

  `pub/ch8/args.scm`

  How would you do that in Java?
- Named parameters allow a user to ignore the order of parameters. This can prevent errors, and is especially useful with default/optional parameters. For example, in Python:

  `pub/ch8/named`

A menagerie of parameterish features (4 of 7)

- Conformant arrays allow a subroutine to process an array of any size, while still being strongly typed. For example, instead of:

  pub/sum/pascal/sum.p

  we can write:

```
    function sum(var seq:
        array [low..high] of integer
    ): integer;
```

  where `low` and `high` are essentially automatically assigned formal parameters to the function `sum`. Sadly, our `fpc` translator doesn't support the Pascal standard.

A menagerie of parameterish features (5 of 7)

- Various PLs also have various function-return-related features.
- Fortran and Pascal use an assignment to the "function name" to determine the return value. In Pascal:

  `pub/sum/pascal/sum.p`
- The function's name is a variable, which is like mathematics, but can be confusing in the presence of recursion.
- In Pascal, this is especially confusing, because you don't need parentheses when the function has no parameters.

A menagerie of parameterish features (6 of 7)

- In SR (Synchronizing Resources), a function's signature specifies a "return variable's" name, separate from the function's name:

  `pub/sum/sr/sum.sr`

- These complications lead to the `return` statement.

A menagerie of parameterish features (7 of 7)

- In some PLs, a function can return a value of any type, others constrain the type to be something simple. For example, a C function can return a scalar or a struct, period.
- Some PLs allow a function to return multiple values (e.g., Perl and Python). In Python:

  `pub/ch8/return`

Closures (1 of 3)

- If a function f_1 creates/defines a function f_2, and f_2 refers to a formal/local of f_1, but f_2 is called in a context where the formal/local is out of scope or has been deallocated, our stack-frame strategy is insufficient.
- We need a closure: a function paired with an environment.
- And, we have to worry about deep/shallow binding.

Closures (2 of 3)

- We saw Scheme closures, back in Chapter 3:

  `pub/ch3/closure1.scm`

  `pub/ch3/closure2.scm`
- What about "normal" PLs?
- Pascal:

  `pub/ch8/closure1.p`

  `pub/ch8/closure2.p`
- Java:

  `pub/ch8/Closure.java`
- Most statically scoped PLs use early/deep binding.

Closures (3 of 3)

- What about PLs with dynamic scope?
- Scheme uses static scope, but other Lisp dialects, including Emacs Lisp, use dynamic scope.
- Emacs Lisp uses late/shallow binding, but you can switch to static scope and early/deep binding, by setting a variable!
- Scheme allows switching, too:

  `pub/ch8/dynamic.scm`
- This is like the Bash program we saw earlier:

  `pub/ch3/dynamic1`
- Here's a Scheme closure, with dynamic scope:

  `pub/ch8/dynclosure.scm`
- We can do that in Bash, too:

  `pub/ch8/dynamic`
- Most dynamically scoped PLs use late/shallow binding.

Generics (aka, Templates)

- Subroutines are a powerful procedural-abstraction tool. They allow you to:
  - encapsulate computation
  - name it, and refer to it by name
  - parameterize it, so it can be used with caller-specified *values*
- Generics are more powerful, also allowing parameterization by *type*.
- This is called *parametric polymorphism*. It is orthogonal to *inheritance polymorphism*, which is provided by object orientation.
- They do *not* have to be used together, or both be provided by a PL:
  - Ada has generics, but is not OO.
  - Older Java was OO, but did not have generics. They were added, later.
  - C++ is OO and has always templates.

Prehistoric Generics

- Is `type` a type?
- Some PLs have a "type" named `any` (e.g., TypeScript, the strongly type version on JavaScript).
- A PL/I pointer can point to any type of value.
- A C/C++ `void*` can do the same. You can cast one to a pointer to a "real" type:

  pub/ch8/generic.c
- Unlike C++, in Java, all classes are subclasses of class `Object`. Thus, to some degree, all objects can be treated uniformly.
- Real generics came from Barbara Liskov's (MIT) ground-breaking PL Clu [1974]. It is an object-based PL (clusters, but no inheritance), which also had iterators and exception handling.

Java and C++ Examples

- A Java generic class:

  `pub/ch8/Box.java`
- A C++ generic class:

  `pub/ch8/Box.h`

  `pub/ch8/Box.cc`

  Note nasty syntax.

Interesting Generic Variations (1 of 3)

- What can the parameter's "type" be?
  - class/type (C++ allows `class` or `typename` keyword)
  - scalar constant (e.g., container size)
- What can be parameterized?
  - classes (we've seen)
  - functions:

    `pub/ch8/GenericFunc.java`

    `pub/ch8/template-func.cc`
  - types (other than classes):

    `pub/ch8/template-type.cc`
  - variables (C++ 2014 standard):

    `pub/ch8/template-var.cc`

Interesting Generic Variations (2 of 3)

- Considering generic parameters as subroutine arguments opens a whole can of worms!
- With nested scopes: static or dynamic scope?
- With dynamic scope: early or late binding?
- Can a generic parameter have a default value (C++)?

```
    template <class Item=int>
        class Box {···}
    template <class Item=Student,
                int size=30>
        class Roster {···}
```

Interesting Generic Variations (3 of 3)

- Can a generic have a variable number of parameters (C++ 2011 standard (hideous syntax!))? For example, imagine a generic `Tuple` class, which can be instantiated with any number of class parameters.
- Can a programmer specialize a generic for certain parameter "values" (C++)? For example, imagine a generic `max` function that uses `<` for all types, but `||` for `bool`.
- Can a generic parameter be generic?
- Can a generic with multiple parameters be partially instantiated, with only some parameters specified, resulting in another generic?

Implementation (1 of 3)

- How would you do, manually, what generics provide?
  - Develop a class, function, etc. for a specific type T_1 in a file named something like `FooT1.java`
  - Copy `FooT1.java` to `FooT2.java`.
  - Use a text editor to perform a global search/replace of T_1 to T_2.
  - Compile, relink, and use.
- Or, you might try to use macros. As we did with:

  `pub/ch8/swap.c`

  Could you use a macro for a whole class?

Implementation (2 of 3)

- You might try to automate your copy/edit process, with a script.
- You might change your build process to scan an application's source code, to determine which "versions" to construct.
- It might maintain a "repository" of instantiated versions.
- You might put that logic in your PL translator!
- That's how C++ generics started: translator-generated versions of source code.

Implementation (3 of 3)

- A very different approach is to translate a generic to a single class, function, etc.
- At run time, it would need to work with values of different, perhaps any, types. It could use `void*` or `Object` arguments.
- At compile time, it would still perform careful type checking. Appropriate casts would automatically be added to the application.
- This is how Java generics work. It's called *type erasure*. The JVM knows nothing about generics. An old JVM can be used with bytecode produced from generic programs.

Constraints (aka, Bounds) (1 of 2)

- A generic often expects a parameter to have certain characteristics. It needs to specify requirements on such a parameter, so the translator can ensure conformance. It is said to *constrain* the parameter.
- Some constraints are syntactic. For example, a class might be required to provide a method with a particular signature (e.g., a < operator on the class's objects).
- Other constraints are semantic. For example, a required method might be required to define an equivalence relation (i.e., symmetric, transitive, and reflexive).
- Mainstream PLs only support syntactic constraints (e.g., a Java `interface`).

Constraints (aka, Bounds) (2 of 2)

- An early experimental OO PL, Obj3, allows a set of constraints to be encapsulated in a *theory*. The way an object satisfies a theory is encapsulated in a *view*. A view maps an object's names to the theory's names, so they don't need to be the same.
- Java example:

  `pub/ch8/Constraint.java`
  `pub/ch8/Value.java`
  `pub/ch8/BoxConstrained.java`
- C++ example:

  `pub/ch8/Constraint.h`
  `pub/ch8/Value.h`
  `pub/ch8/BoxConstrained.cc`

  How does the programmer know that an `Item` must provide a `foo` method?

Exception Handling

- An *exception* is an unexpected or unusual event that occurs during execution.
- They have evolved from hardware interrupts (i.e., fetch/execute polling) and trap instructions (e.g., Linux/Pentium `int 0x80`). However, these return to the interrupted code.
- Some exceptions are detected by the OS/PL run-time system (e.g., numeric overflow)
- Other exceptions are detected by the application (e.g., a parser detecting a syntax error).
- The threshold of unexpectedness is a design decision.

Prehistoric Exception Handling

- You can ignore errors (e.g., `printf` return values).
- You can return a "special" value to indicate an error (e.g., `null`, 0, or -1). The caller might pass the value to be used to indicate error.
- You can use a separate by-reference parameter, or separate/multiple return values, to indicate success or error.
- The caller can pass a closure (or function pointer), which you can jump to in the event of an error.
- You can use what the C standard library provides:

  `pub/ch8/jmp.c`

Exception Handling Semantics

- When an error occurs, we want to perform a non-local transfer of control, perhaps after some local cleanup (e.g., call destructors, deallocate memory, or close files).
- It is a jump, not a call, because we don't want to return to the event-detecting code.
- Thus, we'll have to "unwind" the call stack, as with a jump to a nonlocal label.
- The target of the jump is determined at run time, with dynamic scope.
- We want to transfer control to the dynamically nearest handler, for that kind of error.
- This suggests different *types* of handler.
- We may want to pass data to the handler: values of builtin and/or user-defined types.

Exceptional Examples

- Java:
  `pub/ch8/Excep.java`
- C++:
  `pub/ch8/Excep.cc`
- Lisp (Guile):
  `pub/ch8/excep.guile`
- Python:
  `pub/ch8/excep.py`
- Bash:
  `pub/ch8/excep.sh`
  `pub/ch8/tmpfiles.sh`