Introduction

Why Study Programming Languages?

Major Differences in Programming Languages

- Amount of Abstraction
- Problem Domains
- Concept of a Program

Implementation of Programming Languages

- Von Neumann Architecture
- Machine Language
- Assembly Languages
- Compiled Languages
- Interpreted Languages

Acknowledgements

A large amount of the content of the slides used for this course came from:

- notes shared with me by Dr. George Pothering
- the course textbook: Louden, Kenneth C. and Kenneth A. Lambert, *Programming Languages: Principles and Practice*, Third ed., Course Technology, 2012.
- Felleisen, Matthias. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, 2013.

Why Study Programming Languages?

- Language determines thought.
- When we lack vocabulary or logical structure, our minds are limited in what they can express or understand.
- When we speak or write, we iron out our thoughts, making them clearer to ourselves and others.
- When we program, we do the same.
- If we limit our knowledge of programming languages to a single paradigm, we severely limit our imagination and means of expression; we limit what we believe is possible.

Example: Quicksort

- In 1960, C.A.R. Hoare invented Quicksort, but had a lot of trouble explaining the algorithm to his boss, (although he finally succeeded and won his bet).
- It was only when he learned about recursive procedures at a course on ALGOL 60 that he was able to code it.

  > *Due credit must be paid to the genius of the designers of ALGOL 60 who included recursion in their language and enabled me to describe my invention so elegantly to the world. I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed.*

  1980 Turing Award Lecture, C.A.R. Hoare
  `https://www.cs.fsu.edu/~engelen/courses/COP4610/hoare.pdf`

Why Study Programming Language Concepts?

1. To improve your use of your currently known programming languages.
2. To improve your vocabulary of useful programming concepts.
3. To allow a better choice of programming languages.
4. To make it easier to learn a new language.
5. To learn language design principles so that, if it be you who designs the next language, it will be elegant, thus facilitating future programmers in the design of beautiful programs.

Improving Use of Currently Known Programming Languages

"Think of some of the languages you currently 'know.' Do you know how recursive function calls are implemented? What are the advantages/disadvantages of having programs declare variables before using them versus not using declarations? What differences, if any, are there in how languages treat such structures as strings, arrays, lists, dictionaries, objects, etc.?
By understanding how features in the languages you know are implemented you can better understand how to use them effectively, or how to choose one over another."
— Pothering

Improving your Vocabulary of Useful Programming Concepts

"Familiarity with just a small number of programming languages tends to have a constraining effect on how one goes about solving problems using those languages. In looking for data representations or program operations appropriate for solving a problem, a programmer will rely first on the constructs available in the language in which he or she will be working. If there is nothing in that language that provides direct support then the programmer must devise an implementation for the representation or operation from what is available. Familiarity with how other languages have handled data representations and operations can be helpful."
— Pothering

Choosing the Right Languages for the Right Application

"As we will see shortly, some languages may be better suited for applications in certain areas than others (say artificial intelligence vs web-programming, scientific programming vs database applications, real-time systems vs report-generation). Knowing the strengths and weaknesses of a language vis-a-vis these areas can give a programmer better alternatives."
— Pothering

Ease of Learning a New Language

"Just as a person with a competency in two or more natural languages often can learn a new foreign language more readily than a person who knows only one language, so too can a broad knowledge of a variety of programming language constructs and implementation techniques allow the programmer to learn a new language when the need arises (and if you stay in the profession long enough, the need WILL arise)."
— Pothering

Learning Language Design Principles

- As Tony Hoare noted, "the highest goal of programming language design is to enable good ideas to be elegantly expressed."
- Example of Answer-Set Prolog (designed to express notions of commonsense reasoning for AI). The language was defined to allow a computer to reason according to the following rules:
  1. Believe in the head of a rule if you believe in the body.
  2. Do not believe in contradictions.
  3. Believe nothing you are not forced to believe.

  Can you represent such knowledge and reasoning with the languages you know?
- If you were to design a new language, you should know the consequences of design choices for previous languages.

Another Example from Hoare

"...my suggestion was to pass on a request of our customers to relax the ALGOL60 rule of compulsory declaration of variable names and adopt some reasonable default convention such as that of FORTRAN. I was astonished by the polite but firm rejection of this seemingly innocent suggestion: It was pointed out that the redundancy of ALGOL60 was the best protection against programming and coding errors which could be extremely expensive to detect in a running program and even more expensive not to. The story of the Mariner space rocket to Venus, lost because of the lack of compulsory declarations in FORTRAN, was not to be published until later. I was eventually persuaded of the need to design programming notations so as to maximize the number of errors which cannot be made, or if made, can be reliably detected at compile time."

What We Think We Want vs. What We Want

- In our experience with Answer-Set Prolog, we came to realize that adding sorts (similar to type declarations) has great advantages.
- However, once people get used to not having to do something, they are loathe to have to think about it.
- Programmers have a tendency to want to get something on paper first, so that we can feel like we have accomplished something, and think about it later.
- This comes with the price of constant revisions and even having to retro-fit a program to our specification even though we are writing it for the first time.
- The problem is universal: instant gratification vs. the good

Major Differenced in Programming Languages I

1. Amount of abstraction inherent in the language: high-level languages hide the details of implementation from the user, while low-level languages are closer to the hardware. The more a language forces a programmer to consider the machine itself, the lower-level it is.
   1.1 Examples:
   Machine language $<$ assembler $<$ C $<$ Java $<$ Prolog,
   where $<$ is read a lower-level than.
   1.2 Assembler abstracts bits necessary for representing opcodes, but you still must take things out of memory and place them in registers manually.

Major Differenced in Programming Languages II

1.3 C has higher-level abstractions for control flow, memory use, arithmetic operations, etc., while still giving direct access to specific addresses via pointers. Very popular for systems programming because of this *lack* of abstraction. For example, if you wish to zero-out the timer, you would need access to the address at which the system timer resides. (If you're not a system's programmer, or not a very good one, you can inadvertently overwrite system memory and completely destroy your computer too.)

1.4 Java abstracts out memory allocation by using garbage collection.

1.5 Prolog abstracts out the "how" altogether, and leaves programmers to define the "what". Programmers think in terms of what is true, vs. how to make things true.

Major Differenced in Programming Languages III

2. The types of problems/domains for which a language is best suited
   - 2.1 Scientific languages: designed primarily for the manipulation of numeric data, especially floating point values (think scientific notation).
   - 2.2 Commercial or business languages: designed primarily for the manipulation of files, production of reports, interfacing with databases, etc.
   - 2.3 Artificial intelligence languages facilitate symbolic computation.
   - 2.4 Real-time languages: activate procedures in response to external signals as required.
     Being able to get as close to the underlying architecture as possible in order to ensure a rapid and efficient response is critical.

Major Differenced in Programming Languages IV

2.5 Web-oriented languages:
Markup languages are used to control the presentation of data, for example, "represent these user names as a bullet list or as a table. (Technically these do not fall under the category of "programming" languages.)

Other languages provide dynamic structure to a Web page, and allow programmers to access data from or send data to Web servers by embedding their code in the markup code.

Major Differenced in Programming Languages V

3. How a program is regarded in the language. It is commonly considered that there are four paradigms.
   - 3.1 Procedural languages:
     - A program is a sequences of commands to the computer.
     - Reflect the underlying von Neumann architecture of the machine.
     - Examples: C, Perl, JavaScript, Visual BASIC, .NET
   - 3.2 Functional languages:
     - A program is a succession of (possibly recursive) function calls.
     - Examples: Lisp, Scheme, Haskell, FP
   - 3.3 Logical languages:
     - Programs specify a set of pre-conditions (or known "facts") and express desired post-conditions.
     - Programming in a declarative language amounts to specifying the objects of our domain and the relations between them.
     - Examples: Prolog, ASP, SQL

Major Differenced in Programming Languages VI

3.4 Object-oriented languages:

- A program is sequence of interactions among objects.
- An object is an encapsulation of data and the operations that may be performed on this data.
- The objects may be pre-existing objects or ones that the programmer specifies (often derived from existing objects)
- Object-oriented languages usually have a strongly imperative structure for specifying object operations.
- Inheritance is a major feature of OOs and contributes to code re-use.
- Examples: Smalltalk, C++, Java

Developments in Programming Languages

- Machine Language
- Assembler
- Fortran
- Algol 60 (and family: Pascal, C, Ada, Basic)
- Lisp
- Prolog
- Java

Von Neumann Architecture

To understand the evolution of programming languages, we need to know a little bit about basic computer architecture. The original, von Neumann architecture has the following features:

- Data and programs are stored in memory.
- Memory is separate from the CPU.
- The CPU contains a small set of its own memory cells known as **registers**. These are organized into what is known as the **datapath** of the processor. The datapath also incorporates the arithmetic logic unit, and the flow of data within the datapath is governed by the control unit. The CPU also uses a clock to coordinate all of the data transfers.
- A special-purpose register — the **program counter** — keeps track of where the next instruction resides in memory.
- Instructions and data are transferred between memory and the CPU and are normally placed first in registers.

Fetch-Execute Cycle

A (machine language) program on a von Neumann architecture computer is executed based on the fetch-execute-cycle:

1. Initialize the program counter to the address of the first instruction of the program.
2. Repeat until program termination:
   - fetch the instruction pointed by the counter;
   - increment the program counter;
   - decode the instruction;
   - execute the instruction [may involve getting operand values from memory or transferring results of operations back to memory];

   end repeat

Imperative Languages and the von Neumann Architecture

Imperative languages reflect the structure and operation of von Neumann architectures:

- Variables model memory cells.
- Assignment statements model memory transfers.
- Operations of the imperative language model what might be found in the instruction set for the CPU.

The von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed at which an instruction can be fetched.
- The technology used in the CPU is usually faster (but more expensive) than that used in memory, so program instructions are generally executed much faster than the speed of the connection.
- This difference in speeds results in a bottleneck. Known as the **von Neumann bottleneck**, it is the primary limiting factor in the speed of computers.

Reducing the Bottleneck

- Placing smaller, but faster, units of memory, known as **cache memory**, between the CPU and memory is one approach towards lessening the effects of the von Neumann bottleneck.
- Another is creating a **multi-core architecture**, with special processors for graphics and for floating-point computations. Multi-core architectures allow for parallelization.
- Note that as computer architectures became more and more complicated, applications programming would have become more and more complex if all programmers had been forced to control which parts of a program were farmed out to which processor, manage the back-and-forth of the cache, control parallelization, etc.

Machine Language

- Completely bound to a specific machine and its architecture.
- Example Command: 0010001000000100
- First four bits is an opcode (indicates type of operation such as load, add, increment)
- Remaining twelve specify operands; specifically, they refer to address locations such as registers or memory locations.

Assembly Language

- Uses mnemonic symbols for instruction codes and memory locations.
- Examples:
  LD R1, FIRST ; Copy the number in memory location FIRST to register R1
  FIRST .FILL #5 ; Location FIRST contains decimal 5
- Much easier to read, but still completely tied to the architecture of a specific computer.
- Programmers must tell the computer how to do everything, including moving things in and out of registers.

Assembler

- An **assembler** was needed to translate assembly language into machine language.
- It was essentially one-for-one translation.

Fortran: Formula Translation

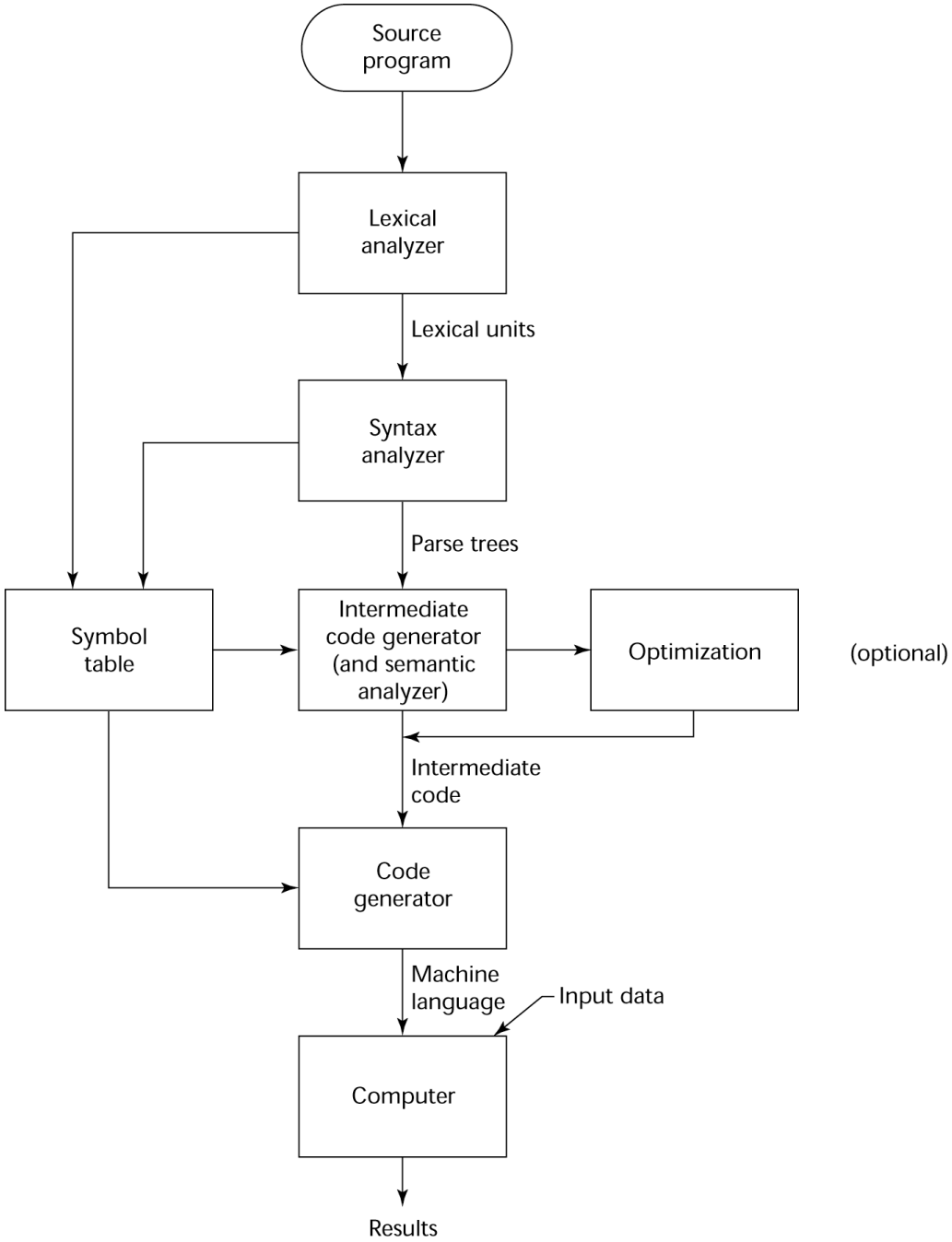
- In the early 1950s, John Backus developed FORTRAN for a particular type of IBM computer.
- Algebraic notation and floating point numbers!
- Programmers could concentrate on much higher-level math instead of implementing every operation themselves.
- Example: One no longer had to implement the square root operation from scratch and move operands from memory to registers and back.
- (Fortran today incorporates much more high-level features.)

Compilers

- Wouldn't it be nice to write code which would work on different machines?
- Enter compilation; a **compiler** translates code into **object code** for a particular machine type.
- Object code can be machine language or some other type of low-level language that is machine-specific. (Note that to get machine code, a compiler calls an assembler. We often don't think of them separately.)

Phases of Compilation

1. **preprocessing**: converts macros such as #define or #include into one or more lines of code in a high-level language.
2. **lexical analysis**: converts characters in the source program into lexical units.
3. **syntax analysis**: transforms lexical units into parse trees which represent the syntactic structure of program.
4. **semantics analysis**: generated intermediate code, checking that typing information is correct in the context.
5. **optimization**: modern compilers make use of code optimization techniques to make it run faster
6. **code generation**: generate object code


Source program
Lexical analyzer
Lexical units
Syntax analyzer
Parse trees
Symbol table
Intermediate code generator (and semantic analyzer)
Optimization
(optional)
Intermediate code
Code generator
Machine language
Input data
Computer
Results

ALGOL-60: ALGOrithmic Language

- Designed to be a compiled language so that code could be independent of a particular architecture, and it was up to the compiler writers to adapt it to different hardware.
- Introduced many structured abstractions:
  - structured control statements: begin-end
  - loops (the for loop)
  - selection (if and if-else statements)
  - notation to distinguish between different numeric types (integer and float)
  - Support for procedures, including recursive procedures.
- ALGOL-60 had a formal specification, so that there was no ambiguity as to what its structures were supposed to do.

The Advent of Interpreters

- Using an interpreter is like having someone between you and the machine to help you communicate, except that "someone" is software.
- The interpreter often functions not just as an "interpreter," but also as a "reasoner."
- Example: a Prolog interpreter answers users' questions based on knowledge given to it by the program. This knowledge can include rules by which the interpreter adds facts to its knowledge base. New facts can also be added by the user, and the interpreter adjusts the knowledge base accordingly.
- LISP and PROLOG developers use interpreters in this way.
- Interpreted programs are slower than compiled ones, but computation speed is not always king. Using an interpreter can often help speed up development because effects of incremental changes can be ascertained as part of the process.

Getting Away from the Machine: LISP

- In the 1930's, Alonzo Church developed **lambda calculus**, a model of computation based on *recursive functions*.
- In the 1950's, John McCarthy used this model to create LISP, a model of programming independent of ties to the von Neumann architecture.
- The focus was on the problem, rather than on the machine.
- McCarthy started by thinking about a language for list processing. Lists could be lists of programs, so functions were in.
- Many people created extensions of Lisp, so it is more of a family of languages than a single language.
- Lisp $\rightarrow$ Scheme (added assignment statements and jumps in control flow) $\rightarrow$ Racket (added a whole bunch of libraries, classes, exceptions, etc.)

Getting Away from the Machine: PROLOG

- In the 1970's, Alain Colmerauer, Philippe Roussel, and Robert Kowalski created Prolog, a language based on *formal logic*.
- Programs are collections of facts and rules and can be queried to ascertain the truth of a given statement.
- The reasoning component of the language is implemented using a method called **resolution**. Programmers need only understand the principles of resolution, not its implementation, to be able to program in the language.
- The Prolog interpreter performs resolution; thus, the programmer need not implement the reasoning mechanism of the machine, and can focus on teaching it what is true in the world.
- Has been useful for automatic-theorem proving, natural language processing, expert systems, planning, etc.

The Virtual Machine

- We have compilers and interpreters, but what if I want to transfer an executable from machine to machine?
- The **Java virtual machine** allowed "compiled" code to be machine-independent because it was compiled into **byte code**, not a specific assembler/machine language.
- In order for byte code to run on a machine, it would have to have a java virtual machine (JVM) which would take byte code as input, and create the necessary translation to a language the machine would understand.
- Note that having an intermediate level of software will slow things down.

Evaluating the Quality of a Language

Why Do We Need So Many?

Popularity

Developing Taste

Expressiveness

Why Do We Need So Many?

- "Uncle Bob" Martin talks about "The Future of Programming" `https://www.youtube.com/watch?v=ecIWPzGEbFc&feature=youtu.be`
- Asks "Why do we need so many programming languages?"
- If you are choosing a language, making the case for it is on you.

Quality Cannot Be Judged by Popularity

Popularity can be influenced by:

- availability
- price
- quality of translators
- politics
- geography
- timing
- markets

Examples

- C was a success in part because of UNIX success, which supported its use.
- COBOL is still used in industry because of a large number of legacy apps.
- Ada was required for certain U.S. Defense Department projects.
- Java and Python — development environments were freely distributed.

Designers Disagree

- People disagree on what constitutes good design.
- However, just because people have different ideas about what is best, doesn't mean there isn't a right way and a wrong way for a particular goal.
- Develop your taste by looking at a variety of good code in different languages.
- Ask experts in a language what they like/dislike about it. Do they have a compelling case? How would you know?
- Don't forget to consider what a language is used for. Sometimes, the language may not be that great, but it is the best out of what is available given the application constraints. Perhaps it has libraries suited to your needs. Perhaps your priority is efficiency over anything else and you just can't sacrifice it for readability.

Syntax and Semantics

We will talk about syntax and semantics of languages at length later. However, you will need to know definitions now.

- **syntax** — the set of rules that defines the combinations of symbols that are considered valid in a given language.
- **semantics** — the meaning of the constructs in a given language.
- Syntax answers "What is a valid statement?"
- Semantics answers "What does it mean?"
- I like to think of them as *form* and *meaning*.

How Can We Evaluate the Quality of a Language?

- readability — the quality of a language that enables a programmer to read and understand a program easily;
- writability — the quality of a language that enables a programmer to use it to express computation clearly, correctly, concisely, and quickly;
- reliability — the ability of a program written in the language to perform to its specifications;
- efficiency — how quickly a program in the language can be executed.

Factors to Consider

1. Expressiveness
2. Regularity
   - Generality
   - Orthogonality
   - Uniformity
3. Reliability
4. Extensibility

Expressiveness I

The basic constructs available in the language for representing data and carrying out operations should be sufficient for the intended purpose of the language, but not excessively so:

- Assuming a programmer is using the language for a task close to the intended purpose of the language, is the basic data support and control structure support adequate for the programmer to map a design into code in a way that is straightforward to write and easy to read?
- On the other hand, does the language provide the ability for the programmer to define and use complex structures or operations beyond those supported in the language in ways that allow details to be ignored?

Expressiveness II

- Are there adequate predefined data types that are suitable for the intended purpose of the language?
  Early versions of C had no boolean type, forcing programmers to use an int to represent true/false (0 is false, everything else is true, so flag = 1; would be used to set flag to true.)
  Consider this statement fragment (here = represents assignment, not comparison):

```
if (k = 5) { ...  } else { ...  }
```

  Some languages allow users to specify their own named data types in order to design code that better reflects the purpose of a program being written in that language.

Expressiveness III

- Form and meaning: Does the language use meaningful keywords, self-descriptive constructs, and employ other aids to make the code easier to understand?
  As one example, consider that different languages delimit compound statements in various contexts:

Compound Statement Comparison: Python

```
def binarySearch(val,numList):
  low = 0
  high = len(numList) - 1
  while(low <= high):
    mid = (low + high) // 2
    item = numList[mid]
    if (val == item):
      return mid
    elif (val < item):
      high = mid - 1
    else:
      low = mid + 1
  return -1
```

Compound Statement Comparison: C

```
int binarySearch(int numList[], int val, int listSize) {
  int low, high, mid, item;
  low = 0
  high = listSize - 1;
  while(low <= high) {
    mid = (low + high) / 2
    item = numList[mid]
    if (val == item)
      return mid;
    else if (val < item)
      high = mid - 1
    else
      low = mid + 1
  }
  return -1
}
```

Compound Statement Comparison: C it your way

```
int binarySearch(int numList[], int val, int listSize) {
int low, high, mid, item;
 low = 0
   high = listSize - 1;
while(low <= high)
{
mid = (low + high) / 2
item = numList[mid]
 if (val == item)
return mid;
else if (val < item)
high = mid - 1
else
low = mid + 1}
 return -1}
```

I can C it, but I can't read it.

What Are You Willing to Force Programmers to Do and Why?

- Enforcing an indentation style while getting rid of requirement to type braces and shortening key words seems a reasonable tradeoff, especially given that you won't have to read someone else's sloppy code.
- However, enforcing typing keywords in all-caps, as in Modula-2, was probably too much sacrifice of writability for readability for too little gain, especially considering that compound statements were delimited by BEGIN...END pairs. (Much of this sort of thing has now been solved with formatting editors.)
- Ideally, programmers would constrain themselves to disciplined, professional code which is easy to read and maintain. Does experience show this?

The Tao of Programming Language Design

There must be a balance between order and chaos. We do not like constraints, but we do not like crashing satellites either.

Evaluating the Quality of a Language: Part 2

Regularity

- Generality
- Orthogonality
- Uniformity

Reliability

Extensibility

Regularity

Regularity — how well the features of a language are integrated.
Subdivided into generality, orthogonality, and uniformity.
Greater regularity implies:

- Fewer restrictions on the use of particular constructs
- Fewer strange interactions between constructs
- Fewer surprises in general in the way the language features behave

Generality

Avoid special cases whenever possible.

- Does the language avoid special cases wherever possible, while minimizing feature multiplicity (having multiple way of accomplishing the same task)?
- Examples:
  - Pascal allows nesting of functions and procedures and passing of functions and procedures as parameters to other functions and procedures, but does not allow them to be assigned to variables or stored in data structures.
  - In C, one cannot directly compare two structures or arrays with ==; thus, this operator lacks generality. Smalltalk, Ada, Python and others do not have this restriction. Their equality operator is more general. Haskell allows operator overloading and the creation of new operators.

Too Much of a Good Thing?

- Generality is a good thing when it makes things less confusing.
- It is debatable, for example, whether operator overloading is a good thing; if you are using the same symbol to mean different things, it is not necessarily clear to your readers what you mean.

Orthogonality I

- Can constructs can be combined in any meaningful way, with no unexpected restrictions or behavior?
- Does the language avoid situations in which some language constructs would behave differently in different contexts?

Orthogonality II

- A relatively small set of primitive constructs can be combined in a relatively small number of ways.
- Ideally, every possible combination is legal and the meaning of each such construct can be discerned from the primitive constructs involved.
- The level of orthogonality is diminished if particular combinations are forbidden (as exceptional cases) or if the meaning of a particular combination is not evident from the meanings of its component parts, each one considered without regard to context.
- Orthogonality was a major design goal of Algol68, which is still the best example of a language in which constructs can be combined in all meaningful ways, but it could make programs difficult to understand. (more later)

Examples of Lack of Orthogonality I

- Function return types:
  - Pascal allows only scalar or pointer types as return values
  - C and C++ allow values of all data types except array types
  - Ada and Python allow all data types
- Placement of variable declarations:
  - C requires that local variables be defined only at the beginning of a block
  - C++ allows variable definitions at any point inside a block prior to use
- Primitive and reference types
  - In Java, primitive types use value semantics (values are copied during assignment), while object types (or reference types) use reference semantics (assignment produces two references to the same object).

Examples of Non-Orthogonality in C

- A function can return a value of any type, except for an array type or a function type.
- Parameters to functions are passed "by value", except for arrays, which are, in effect, passed "by reference". (Is this a valid criticism? After all, one should (?) understand a variable of an array type to have a value that is actually a pointer to an array. So passing an array to a function is really passing a pointer "by value." This is exactly how Java works when passing objects to methods. What is being passed is really a reference (i.e., pointer) to an object, not the object itself.)
- In the expression `a + b`, the meaning of `b` depends on whether or not `a` is of a pointer type. (This is an example of context dependence.)

Example from Assembly Languages

- In VAX assembler, the instruction for 32-bit integer addition is of the form
  `ADDL op1 op2`
  where each of the op_is can refer to either a register or a memory location. This is nicely orthogonal.
- In contrast, in the assembly languages for IBM mainframes, there are two separate analogous ADD instructions, one of which requires op1 to refer to a register and op2 to refer to a memory location, the other of which requires both to refer to registers. This is lacking in orthogonality.

Too Much Orthogonality?

B.T. Denvir wrote (see page 18 in "On Orthogonality in Programming Languages", ACM SIGPLAN Notices, July 1979, accessible via the ACM Digital Library):

> *Intuition leads one to ascribe certain advantages to orthogonality: the reduction in the number of special rules or exceptions to rules should make a language easier "to describe, to learn, and to implement" — in the words of the Algol 68 report. On the other hand, strict application of the orthogonality principle may lead to constructs which are conceptually obscure when a rule is applied to a context in an unusual combination. Likewise the application of orthogonality may extend the power and generality of a language beyond that required for its purpose, and thus may require increased conceptual ability on the part of those who need to learn and use it.*

Example: Too Much Orthogonality

- As with almost everything, one can go too far. Algol 68 was designed to be very orthogonal, and turned out to be too much so, perhaps.
- For example, it allows the left hand side of an assignment statement to be any expression that evaluates to an address.

Uniformity

- Do constructs that look similar behave in a similar manner?
- Do dissimilar constructs behave similarly when they should not?
- The language should have a "manageable" set of basic constructs.

Examples

- Back to operator overloading: It makes sense for + to be used for both int and float. What about using * for multiplication and pointer dereferencing as in C and C++?
- C++ requires a semicolon after a class definition but forbids its use after a function definition (because designers wanted to keep C compatibility)
- Pascal uses the function name in an assignment statement to return the function's value; this looks confusingly like a standard assignment statement. Many languages use a `return` statement instead.

Reliability

- Refers to the degree to which a program performs to its specifications.
- Reliability can be compromised if a language does not support "natural" ways of expressing an algorithm — requiring the use of "unnatural" approaches.
- Reliability can also be affected if restrictions are not imposed on certain features.

Features which Promote Reliability

- **Type checking:** Testing for type errors can be done either by the compiler, or at run time. The earlier this is done, however, the more quickly errors are found and corrected.
- **Exception handling:** the ability of a program to intercept run-time errors (for example division by 0) and take corrective measures.
- **Restricting aliasing:** Aliasing occurs when there are two or more distinct referencing methods for the same memory location.

Semantic Safety

A language is considered **semantically safe** if it prevents programmers from compiling or executing any statements or expressions that violate the definition of the lanuguage.

- Python, Lisp and Java are considered semantically safe.
- C and C++ are not.
- Array index out of bounds causes a compile-time or a run-time error in semantically safe languages. Can you see why it might be catastrophic if it is not detected early?
- Garbage collection is also used to help make languages semantically safe. It is used to to prevents memory leaks. Forgetting to deallocate memory in C or C++ can lead to serious run-time errors which are very difficult to detect.

Style Guidelines

- If a language does not enforce good practice, style guidelines can be used to help alleviate some of that, but they cannot be enforced.
- In a recent interview, Bjarne Stroustrup, the creator of C++, stated: "My aim here [with the C++ Core Guidelines] first is to list [define] a style of C++ that is completely type safe and completely resource safe. That will solve half the number of bugs right there. Doing that without narrowing the application areas and without slowing down code is tricky, but I think we can do it."[1]

[1] Stroustrup, Bjarne. "Episode 5: An Interview with Bjarne Strustrup, Creator of C++." Interview by Pramod HS. Audio blog post. Mapping the Journey. N.p., 29 July 2017. Web. 18 Apr. 2018. `https://www.mappingthejourney.com/single-post/2017/07/29/interview-with-bjarne-stroustrup-creator-of-c/`.

Extensibility

Refers to a language's ability to allow the user to add features to it. Most common are the ability to define new data types and new operations (functions or procedures).

- C++, C#, and F# allow user-defined overloaded operators
- When sensibly used, such operators can be an aid to readability (avoid method calls, expressions appear natural)
- Potential problems: Users can define nonsense operations. Readability may suffer, even when the operators make sense.
- Very few languages allow additions to the syntax and semantics, although LISP allows something like this through its use of macros.

Functional Programming

Introduction to Functional Programming

Referential Transparency

DrRacket

Scheme (a dialect of LISP)

- Expressions
- Evaluation of Expressions
- List Evaluation
- Primitive Functions
- Testing for Equivalence
- Primitive List Functions

Reading

Read Sections 3.0–3.2, pp. 46–64.

Based on the Idea of Mathematical Functions

- In 1932 Alonzo Church, in order to understand more fully the what tasks can be described by algorithms, developed a theory of computation based on the concept of mathematical functions.
- Called theory "lambda calculus" — solid theoretical basis.
- Became the basis for LISP, the first functional language, developed by John McCarthy in the 1950s.
- The interpreter for LISP was written in LISP, so it was easy to modify and tinker with. Thus, many people played with it and we have many LISPs. (More like a family of languages than just one language.)
- Scheme is a variant of LISP which became pretty standard.

What's Good about Functional Programming?

- Simple syntax and semantics.
- Programs are treated as functions.
- Functions are treated as data.
- Book says: "A functional language is a particularly good choice in situations where development time is short and you need to use clear, concise code with predictable behaviour." (Is there ever a case when this is not true?)

Efficiency

- At first, LISP was much less efficient because it was interpreted rather than compiled.
- Advances in compilation techniques (and interpretation techniques), allowed functional languages to compete nicely and...
- It turned out that the lack of side effects in functional languages allows for easy parallelization, which makes optimal use of modern, multi-core architectures.
- Modern libraries and interfaces to other languages (such as C) are also available.

Mathematical Functions and Function Notation

- A function is a rule that associates to each x from some set X a unique y from a set Y.
- In mathematical notation, the function is typically written as
  $$f : X \to Y$$
  or as
  $$y = f(x).$$
- The set X is called the **domain** of f.
- The set Y is called the **range** of f.
- Scheme (like LISP) uses prefix notation
  - `y = (f x)`
  - Multivariable: `z = (f x y)`

Formal vs. Actual Parameters

- The parameters used in function definitions are known as **formal parameters**, while the values in place of the formal parameters when a function is invoked are called **actual parameters**.

Pure Function Programming

- Has no assignment statements.
- Has no loops.
- Turns out, they're not needed for computational completeness.

Assignment

- In an imperative language variables refer to memory locations that store values; operations are done and the results are stored in variables for later use.
- In a functional programming language, variables are bound to values, not memory locations.
- `i = i + 1` has no meaning because the two sides are not equivalent; once a variable is bound to a value, that variable's value cannot change.
- In languages based on mathematical functions, since a value is permanently bound to a variable during a call to that function, there is no need for the concept of a memory. This removes the need for assignment as an operation.

Loops

- A key underpinning of looping in imperative programming languages is the use of a control variable whose value changes as the loop executes.
- Lack of assignment in a functional language makes loops impossible in this language. Instead recursion is used to accomplish repeated execution of code.

C Code for Greatest Common Divisor

```
void gcd(int u,
         int v,
         int* x){
  int y, t, z;
  z = u; y = v;
  while(y != 0){
    t = y;
    y = z % y;
    z = t;
  }
  *x = t;
}
```

```
int gcd(int u, int v){
  if(v == 0)
    return u;
  else
    return gcd(v, u % v);
}
```

Side Effects

- One consequence of the use of variables in an imperative language is that a sequence of operations can return different values for the same input because of side effects involving stored values.
- Example: `0example_referential_transparency.docx`
- No side-effects in a *purely* functional language.

Referential Transparency

- **referential transparency** — the property of a programming language that allows an expression to be replaced with one that is equivalent in meaning without changing a program's behavior.
- Pure functional programming languages strive for it, whereas imperative languages, where there could be "side-effects," do not. (However, people programming in those languages can strive to have referentially transparent functions, etc.)
- Referential transparency can help:
  - make code easier to understand and modify
  - optimization through parallelization and other means.

Side Effects in Functions

- A side-effect occurs in a function call if the function modifies the state of something outside its own scope.
- A function is referentially transparent if its value depends only on the values of its arguments (and nonlocal variables, which in effect are constants).
- For example, the `gcd` function is referentially transparent.
- A function `rand`, which returns a pseudo random value, cannot be referentially transparent, because it depends on the state of the machine (and previous calls to itself).

Dealing with Side Effects

- Any programming language that intends to allow programs to interact with the "outside world" — for example with a user interface, file system, outside instruments, other computers — must work with side effects.
- With functional programming languages there are two overall approaches to side effects:
  1. Don't be a pure functional language. Do not restrict side effects, but restrict the constructs where they can arise. Scheme is one language that takes this approach.
  2. Incorporate into the language a means for modeling side effects. For example the functional language Haskell addresses side effects such as I/O and other outside world interacts by incorporating them into its type system.

An Environment for Running Scheme

- Scheme is a popular extension of LISP.
- You can download a nice interface called DrRacket for working with Scheme programs from `https://racket-lang.org`. Choose your platform.
- (Racket is another extension of LISP and is lots of fun. We're using Scheme because it's in your book, but many of the programs will run in both Scheme and Racket.)
- DrRacket allows you to play with many, many different Lisp-like languages.

DrRacket Quickstart

- When you open the app, you'll see two large panels. The top is called the **definition panel** and the bottom is called the **interactions panel**.
- Choose your language by typing `#lang scheme` into the definition panel and clicking Run in the top right corner.
- Now you can play around with the interpreter in the interactions panel by typing something in and hitting return, or write a program in the definitions panel and clicking Run.
- Programs can be saved, etc., in an intuitive way. If you've opened a previously-saved program and don't see an interactions panel, click Run.
- There is good documentation on the website, so when in doubt, look it up.

Scheme Syntax

Using a meta-language known as extended Backus-Naur Form (EBNF), we define a Scheme expression as follows:

`expression` → `atom` | ‘(’ {`expression`} ‘)’

`atom` → `number` | `string` | `symbol` | `char` | `boolean`

In EBNF, the symbols are defined as follows:

→	is defined as
|	or
‘thing in quotes’	literally what’s in the quotes
{token}	there can be zero or more such tokens

Parenthesized Expressions

- Depending on context, parenthesized expressions can represent computations to be performed, or they can represent data.
- When they represent data, they are called lists.
- If they represent a computation to be performed, in most cases the element immediately after the leftmost parentheses is an atom that is either a string, symbol, or character representing the computation to be performed.
- A parenthesized expression with no expression inside is a valid parenthesized expression.
- Some symbols (for example `define`, `if`, `cond`, `car`, `cdr`, `+`, `*`, `/`, etc.) are keywords in Scheme and have special meanings when encountered as the first component of a parenthesized expression.
  An expression that starts with a keyword is called a **special form.**

Examples of Scheme Expression

(from Table 3.2 on page 51 of textbook)

Expression	What it represents
`42`	atom – an integer
`"hello"`	atom – a string
`#t`	atom – the boolean value true
`#f`	atom – the boolean value false
`#\a`	atom – the character 'a'
`(2.1, 2.2, 3.1)`	expression with three atoms as elements
`a`	atom – a symbol
`hello`	atom – a symbol
`(+ 2 3)`	expression – the first component is the symbol +
`(* (+2 3) (/ 6 2))`	expression with one atom (a symbol) and two expressions as components

Evaluation of Expressions in Scheme

- Atomic literals evaluate to themselves.
- Symbols other than keywords are treated as identifiers or variables. Their values are looked up in a symbol table known as the current environment, and each symbol is replaced by the appropriate value found there.
- A parenthesized expression or list is evaluated in one of two ways:
  1. If the first item is a keyword, a special rule is applied to evaluate the rest of the expression.
  2. Otherwise, the parenthesized expression is treated as a function application.

List Evaluation I

- Each expression within the parentheses is evaluated recursively, starting with the deepest nesting. Examples from 1Scheme-ArithmeticExpressions.docx.
- The first expression must evaluate to a function, which is then applied to remaining values (its arguments).

List Evaluation II

- When a list of data is to be represented directly in a program, say a list of numbers such as (2.1 2.2 3.1), to prevent the first element from being misinterpreted as a function name, a special form beginning with the keyword `quote` is used:

  ```
  (quote (2.1 2.2 3.1))
  ```

  This function call will return the list of elements being quoted as its value.

  `(quote (2.1 2.2 3.1))` evaluates to the list (2.1 2.2 3.1)

  Because this special form is used frequently, a special shorthand for it can be used — the single quote mark before the left parenthesis.

  `'(2.1 2.2 3.1)` is the same as `(quote (2.1 2.2 3.1))`

Primitive Functions in Scheme

- Arithmetic:

```
+, -, *, /, modulo, abs
expt, sqrt, remainder, min, max
```

- Numeric-based predicate functions (evaluate as #t or #f):

```
=, >, <, >=, <=,
even?, odd?,
zero?, negative?, positive?
```

- Logic-based predicate functions:

```
and, or, not
```

Testing for Equivalence

- You can compare strings with `string=?`, numbers with `=`, booleans with `boolean=?`
- You can create a struct, say `student`, and compare structs of type student with `student=?`
- Use `eqv?` to compare primitives.
- Use `equal?` as the universal equality predicate; it compares anything, including data within composite data.
- Consider why anyone would use anything but `equal?`.

Primitive List Functions

- `car`: takes a list as a parameter; returns the first element.
- `cdr`: takes a list as a parameter; returns the given list, but with the first element removed.
- `cons`: takes two parameters, the second of which is a list; it returns a new list in which the first element is prepended to the second element.
- `list`: takes one or more parameter and makes a list with the values of these parameters as elements.
- `list?`: takes one parameter; returns #t if the parameter is a list; otherwise returns #f
- `null?`: takes one parameter; returns #t if the parameter is the empty list '(); otherwise returns #f

Functional Programming

Control Flow
Lambda Special Form
Recursion

Control Flow

- **if** — the if statement
- **cond** — the conditional (like a switch or a case statement)
- function calls

Special Form: **if**

- ‘(’ if predicate
  then_expression
  [else_expression] ‘)’
- Example:

```
(if (not (= count 0))
    (/ sum count)
    "divide by zero")
```

(Here, `count` and `sum` are symbols that presumably have values that can be obtained by looking them up in the current environment.)

- Example:

```
(if (not (= count 0))
    (/ sum count))
```

Special Form: **cond**

- ```
  (cond
     (predicate1 expression1)
     (predicate2 expression2)
           ...
     (predicateN expressionN)
     (else else_expression))
  ```
- Always checked top to bottom, so the first success defines the value `cond` returns.
  ```

Example 1: `if` recast as `cond`

The if statement:

```
(if (not (= count 0))
    (/ sum count)
    "divide by zero")
```

The cond statment:

```
(cond
    (not (= count 0)) (/ sum count)
    (else "divide by 0"))
```

Example 2: Implementation of the conditions of the signum function sgn

```
Mathematical definition:
            1     if x > 0
sgn(x) =    0     if x = 0
           -1     if x < 0
```

```
Scheme definition:
(cond
     ((positive? x) 1)
     ((zero? x) 0)
     (else -1))
```

Example 3: Leap Year

The atom `year` is assumed to be numeric.
Note, the order of the predicate conditions is important here.

```
(cond
    ((zero? (modulo year 400)) #t)
    ((zero? (modulo year 100)) #f)
    (else (zero? (modulo year 4))))
```

Function Definition: **lambda** special form

- "lambda" special form and lambda functions named in honor of Alonzo Church who invented lambda calculus in his 1932 work in computation theory.
- (**lambda** parameters_list body_expression)
- Specify a lambda expression for calculating the area of a circle given its radius r.

```
(lambda (r)
   (* 3.14 (* r r)))
```

Invoking the Function

- To invoke a function one constructs a list with the lambda expression as the first element and actual parameter values as the remaining list elements.
- To calculate the area of a circle with radius 3.67 one would use the expression

```
((lambda (r)
    (* 3.14 (* r r)))
    3.67)
```

(Note: `r` is a parameter, not the function name.)

Relax — There's a Shortcut!

- You'll probably want to name your function so that you can use it more than once.
- The long way is:

```
(define circleArea
    (lambda (r)
        (* 3.14 (* r r))))
```

- The short way is:

```
(define (circleArea r)
    (* 3.14 (* r r)))
```

- Either way, you can call the function with:

```
(circleArea 3.67)
```

More Generally

```
(define (functionName parameters) expression)
```

is an abbreviated form of

```
(define functionName (lambda parameters (expression)))
```

Example Function: Defining not=

We can define our own `not=` operator.

```
(define (not= x y)
    (if (= x y)
        #f
        #t))
```

Note: You can also name it <> or `!=`. Also note that these names are not quite as self-descriptive as `not=` because they bring with them associations from other languages. For example, would you expect `!=` to work on things other than numbers?

More Examples

Implementing the signum computation shown earlier as a function

```
(define (sgn x)
   (cond
     ((positive? x) 1)
     ((zero? x) 0)
     (else -1)))
```

Iteration

- Iteration in Scheme is done with recursion.
- Recursion Review:
  - Define your base cases. Example questions you might ask: What should the function do if it is called with an empty list? What does it do with a single element?
  - Have the function call itself to process the rest of the elements.
  - For example, when processing a list, you can specify what your function will do with an empty list (and/or with a single element), and have it call itself to process the rest of the list.

```
;; Computes the number of elements in a list.
;; add1 is a built-in function
(define (my-length list)
  (if (empty? list)
    0
    (add1 (my-length (cdr list)))))
```

More Example of Recursion and List Processing

- `member?` takes an atom and a simple list; returns #t if the atom is in the list; #f otherwise
- prepend takes two lists as parameters; returns a new list consisting of the elements of the first list followed by the elements of the second list.
- reverse returns the reverse of its simple list parameter
- `apply_to_all` applies the given function (which accepts a single parameter) to all elements of the given list, giving the results in a list.
- (CSCI320-S17-Scheme Function Definitions.docx: member, prepend, reverse, apply_to_all)

Outline

Negation as Failure
Built-in Operators
Lists

Negation as Failure

- Prolog allows rules which contain the operator \+ which represents negation.
- Negation in Prolog does not mean that something is known to be false, but only that it cannot be proved by the rules of the program.
- It is implemented by having the interpreter try to prove the negated atom; if it fails, then the negation of the atom is true. Thus the term, **negation as failure**.

Example: Negation as Failure

```
p(a).
p(b).
q(a).
r(X) :- p(X), \+ q(X).
```

Here, `r(X)` is true only when `X=b`, because Prolog can prove `q(a)`, but cannot prove `q(b)`.

Floundering

- Warning: When using negation as failure, you may get into trouble if you are working with negated atoms which still contain variables at the time the interpreter tries to resolve them.
- This is called **floundering**.
- In our example, `p(X)` ensures that `q(X)` is instantiated with `X=a`, and then with `X=b`, so Prolog doesn't have to worry about variables.
- It is best to consult how your particular interpreter handles this.

Arithmetic Operators

Infix or prefix notation can be used:

```
X + Y    or    +(X,Y)
X - Y    or    -(X,Y)
X * Y    or    *(X,Y)
X / Y    or    /(X,Y)
X mod Y or mod(X,Y)
```

Comparison Operators

```
X = Y    or  =(X,Y)
X \= Y   or  \=(X,Y)
X < Y    or  <(X,Y)
X > Y    or  >(X,Y)
X =< Y   or  =<(X,Y)
X >= Y   or  >=(X,Y)
```

Calculations, Comparisons and the **is** Operator

- The programmer signals his or her intention that an arithmetic expression be evaluated by using the special term **is**.
- Especially when appearing in the body of a rule, is will take an arithmetic expression as a right operand and a variable as left operand.

```
A is B / 17 + C  /* Evaluate and use value for A */
```

- Note, this is not the same as an assignment statement! The variable on the left merely represents an instantiation of that variable. The following is illegal:

```
Sum is Sum + Number.
```

Example of a Use of **is**

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :- speed(X,Speed), time(X,Time),
                 Y is Speed * Time.
```

The query:

```
?-distance(chevy, Chevy_Distance).
```

would be satisfied with the instantiation

```
Chevy_Distance = 2205
```

List Structures

- Another basic data structure (besides numbers and atomic terms) is the list.
- The simplest way to write a list is to display its elements, comma-separated, inside square brackets.
- The elements are terms (including other lists and variables).
- The empty list is represented as []
- Examples:

```
[apple, prune, grape, kumquat]
[1, 2, [a, b, c]]
```

Splitting Up Lists

- A common operation with lists is to split them up into initial elements and a trailing list using the special notation

  ```
  [initial elements | trailing list]
  ```

- While the trailing list part should remind you of cdr, the initial elements part is more general than car since more than one element can be presented there.
  Example: The list `[a, b, c]` can also be written

  ```
  [a, b, c | []]
  [a, b | [c]]
  [a | [b, c]]
  ```

- However, the use of head and tail to split up a list is the most common, so it is a lot like car and cdr.

Lists: Example 1

```
p([1,2,3]).
p([the, cat, sat, [on, the, mat]]).

?- p([X|Y]).
X = 1 Y = [2,3]
X = the Y = [cat, sat, [on, the, mat]]
```

Lists: Example 2: member

- Define **member(X, List)** which holds if **X** is a member of a **List**.

```
member(X, [X | _]).
member(X, [_ | Y]) :- member(X,Y).
```

- Here the underscore represents an **anonymous variable**, and it stands for "anything." It saves us from having to dream up different variable names when they will not be used anywhere else in a clause.
- "X is a member of any list that has X as its head."
  "X is a member of a list with any head if it is a member of that list's tail."

Lists: Example 3: append

```
/* append(List1, List2, ResultingList) */

append([], List, List).
append([Head | List1], List2, [Head | List3]) :-
                append (List1, List2, List3).
```

- The result of appending a list `List` to the empty list is the list `List`
- The result of appending `List2` to `[Head | List1]` is `[Head | List3]` if the result of appending `List2` to `List1` is `List3`.

Outline

Lists: Example 4: reverse

```
reverse([], []).
reverse([Head | Tail], List) :-
        reverse (Tail, Result),
        append (Result, [Head], List).
```

More Examples

The next three examples show the application of Prolog to some common programming problems.

- Computing the greatest common divisor
- Using a generate and test strategy
- Quicksort

Euclid's Algorithm for Finding GCD in Prolog

```
/* gcd(A,B,C): "the gcd of A and B is C" */
gcd(A,0,A).
gcd(A,B,D) :- (A > B), (B > 0),
              R is A mod B,
              gcd(B, R, D).
gcd(A,B,D) :- (A < B), gcd(B, A, D).
```

Division Using Generate-and-Test

- The generate-and-test approach to a problem: a rule or goal has subgoals that generate potential solutions to a problem which are then checked to see if they are indeed solutions by later "test" subgoals.
- The purpose of this program is to perform integer division using addition and multiplication. [Note, since Prolog supports division, this program only serves as an example of a generate and test problem; you would never actually use it.]

Division Using Generate-and-Test

```
/* is_integer(X) generates successive integers.
 */
is_integer(0).
is_integer(X) :- is_integer(Y), X is Y + 1.

/* divide(N1,N2,Result)
   N1,N2, and Result are integers
   Notice that divide(N1,N2,Result) holds only when
     Result is the integer quotient of N1 and N2;
     thus, it will not hold when Result is incorrect.
 */
divide(N1,N2,Result ) :-
    is_integer(Result),
    Product1 is Result * N2,
    Product2 is (Result + 1) * N2,
    Product1 =< N1, Product2 > N1.
```

Quicksort Review

Direct quote from Wikipedia
https://en.wikipedia.org/wiki/Quicksort

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base cases of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

Quicksort: Definition

Quicksort — we specify a sorted relationship according to quicksort. We assume we have the append relationship we defined earlier. Splitting is defined next.

```
quicksort([],[]).
quicksort([Head|[]], [Head]).
quicksort([Pivot|Unsorted], AllSorted) :-
   split(Pivot, Unsorted, Small, Large),
   quicksort(Small, SmSorted),
   quicksort(Large, LgSorted),
   append(SmSorted, [Pivot | LgSorted], AllSorted).
```

Quicksort: Partitioning

Now let's specify the partitioning/splitting relationship. The pivot value used will be the first one in the list (if there is one).

```
/* split(Pivot, Unsorted, Small, Large */

split(_,[],[],[]).
split(Pivot, [Head|Tail], [Head|Sm], Lg) :-
        (Head < Pivot),split(Pivot, Tail, Sm, Lg).
split(Pivot, [Head|Tail], Sm, [Head|Lg]) :-
        (Pivot =< Head), split(Pivot, Tail, Sm, Lg).
```

The Cut Operator

The "cut", abbreviated !, allows you to tell Prolog which previous choices it need not consider again when it backtracks through a chain of satisfied goals.

Why do this:

- The program may operate faster because it will not waste time attempting to satisfy goals that one can tell beforehand will never contribute to a solution.
- The program may occupy less of the computer's memory because more economical use of memory can be made if backtracking points do not have to be recorded for later examination. In some cases this could be an infinite number of backtracking points.

How to Use Cut

- Syntactically the cut symbol ! appears as a (sub)goal for a predicate.
- As a subgoal this succeeds immediately upon being encountered and cannot be re-satisfied.
- This has the effect of stopping backtracking at the cut location. Prolog will not attempt to re-satisfy any subgoal preceding the cut symbol.

Cut: Example

```
p(X,Y) :- A1, A2, A3,!,B1,B2
```

- Until the cut is encountered, processing goes as usual.
- Once ! is executed, however, no other alternatives to p will be considered and alternative solutions to A1, A2, A3 will not be sought.

Outline

Reading

Chapter 6: Syntax

- Read Chapter 6 through Section 6.7; i.e., pp. 204–236.

Syntax vs. Semantics

- **Syntax** refers to the form or structure of the expressions, statements, and program units found in a programming language.
- **Semantics** refers to the meaning of such expressions, statements, and program units.
- In this chapter we will be focusing on concepts related to the syntax of programming languages.

Tokens

The **lexical structure** of a programming language is the structure of its **tokens**:

- reserved words, such as `if` and `while`
- literals or constants, such as 42 or `"hello"`
- special symbols, such as “;”, “<=”, or “+”
- identifiers, such as `x24` or `monthly_balance`

Scanning vs. Parsing Phase

Determining the lexical structure of a language is usually done in two phases:

- **scanning phase** — when a translator collects sequences of characters from the input program and forms them into tokens; i.e., before this, a program is just a bunch of characters and white space.
- **parsing phase** — when a translator processes tokens to determine a program's syntactic structure; i.e., it determines whether the tokens, being where they are, follow the syntactic rules of the language.

Regular Expressions

Tokens can be formally described by **regular expressions**, which are descriptions of patterns of characters. There are three basic patterns of characters in regular expressions:

- **Concatenation:** done by sequencing the items.
  `xy` means an `x` followed by a `y`
- **Repetition:** indicated by an asterisk after the item to be repeated. This is also known as the Klene closure. Sometimes a + is used to represent a positive closure, meaning "one or more repetitions"
  `x*y` means a sequence of 0 or more `x`s followed by a `y`
  `x+y` means a sequence of 1 or more `x`s followed by a `y`
- **Choice** (or selection, or alternation): indicated by a vertical bar between items to be selected.
  `x | y` means an `x` or a `y`

Other Conventions for Ease of Expression

- `[ ]` enclosing two ordered characters separated by a hyphen indicates a range of characters.
  `[1-9]` means a digit between 1 and 9, inclusive
- `?` indicates an optional item.
  `x?y` means a `y` optionally preceded by an `x`
- A period indicates "any character"
- Parentheses are used for grouping and for clarity.
- A backslash preceding a character works to remove the special meaning of a symbol

Examples of Regular Expressions

- Token **signed_integer** representing constants of one or more digits, but with no leading 0s:
  `(\+ | \-)[1-9][0-9]*`
- Token **identifier** specifying that it must be a letter followed by zero or more letters or digits:
  `([A-Z]|[a-z])([A-Z]|[a-z]|[0-9])*`
- Unsigned floating-point literals consisting of one or more digits followed by an optional fractional part consisting of a decimal point, followed again by one or more digits:
  `[0-9]+(\.[0-9]+)?`
- Binary strings containing 01:
  `(0 | 1)*01(0 | 1)*`
- Binary strings representing numbers divisible by 2:
  `(0 | 1)*0`

Outline

Reading

Chapter 6: Syntax

- Read Chapter 6 through Section 6.7; i.e., pp. 204–236.

Context-Free Grammars

- In 1950 Noam Chomsky developed the idea of context-free grammars.
- Although originally meant to describe the syntax of natural languages, they are especially useful for representing the structure of computer languages.
- A formal grammar is a logical system for specifying the syntax of a language.

Context-Free vs. Context-Sensitive

- "The yak tied his shoe." is a grammatically correct sentence, even though the context is strange. If the verb "tied" was to be limited only to subjects of a sentence that could actually tie something, then the grammar would be context-sensitive.
- Context sensitive languages may model constraints which cannot be modeled in context-free grammars; for example "no identifier can be declared more than once in the declaration section of a program."
- It is often better to define a formal grammar which is context-free, and write out any other limitations in natural language.

Backus-Naur Forms

- In the late 1950s John Backus and Peter Naur developed a notational system for describing context-free grammars.
- This notational system is now called the Backus-Naur Form, or BNF.
- First used to describe the syntax of early Algol.
- Original BNF was later refined into variants: Extended BNF (EBNF) and syntax diagrams.
- Every modern computer scientist needs to know how to read, interpret, and apply BNF or a variant to descriptions of language syntax.
- Languages used to describe other languages are called **meta-languages**.

BNF

- Backus-Naur Forms are sets of productions (rules) for syntactically-correct statements of a given language.
- Every production has the following structure:

$$\textit{nonterminal} \rightarrow \textit{expansion}$$

  where the symbol $\rightarrow$ should be interpreted as "may expand into" or "may be replaced with."
  Sometimes a more convenient notation such as `:=` or `::=` is used in place of $\rightarrow$.
- Every nonterminal is surrounded by angle brackets, `< >`, whether it appears on the left- or right-hand side of the rule.

BNF, cont.

- An expansion is an expression containing terminal symbols and nonterminal, joined together by sequencing (so order matters) and choice.
  Example:
  `<stmt>` → `<var> = <expr>`
- The order of juxtaposing in expansion expressions indicates their sequencing.
  Example:
  `<stmt>` → `<var> = <expr>` is different from
  `<stmt>` → `<expr> = <var>`
- A vertical bar | indicates choice ("or").
  Example:
  `<stmts>` → `<stmt>` | `<stmt>;<stmts>`
  Read this as "statements are either a single statement, or a statement followed by a semi-colon, followed by statements"

Example: Simple Grammar and Derivation

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

Example derivation:

<program> ⟹ <stmts>

⟹ <stmt>

⟹ <var> = <expr>

⟹ a = <expr>

⟹ a = <term> + <term>

⟹ a = <var> + <term>

⟹ a = b + <term>

⟹ a = b + const

Thus, expression a = b + const is a sentence in this grammar.

BNFs, Derivations, and Parsing

- Determining whether a given string is a sentence with respect to a grammar is known as **parsing** the string. (Recall the *parsing phase* of a translator.)
- BNF production rules give us the capacity to build grammatically-correct strings,
- **Derivation** refers to both the process of building in a language by beginning with the start symbol and replacing left-hand sides by choices of right-hand sides in the rules, and it also refers to the sequence of replacements itself.
- Given a string and a grammar, one can attempt to parse the string by:
  - attempting a derivation from the goal to the string,
  - by trying to work backwards from the string to the goal,
  - or a combination of the two.

Left-Most vs. Right-Most Derivation

- A *left-most derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A *right-most derivation* is one in which the rightmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither left-most nor right-most but a mix of the two.

Parse Trees

A **parse tree** is a hierarchical representation of a derivation.

- The start symbol is the root of the tree.
- Nodes that have at least one child are labeled with nonterminals.
- Leaves (nodes with no children) are labeled with terminals.
- The structure of a parse tree is completely specified by the grammar rules of the language and a derivation of the sequence of terminals.
- All terminals and nonterminals in a derivation are included in the parse tree.

Example 1

Consider the following grammar

<G> → <E>

<E> → <F> | <E> + <F>

<F> → <id> | <F> * <id>

<id> → X | Y | Z

For the expression X + Y * Z give

- a left-most derivation,
- a right-most derivation, and
- a parse tree for each (left-most and right-most).
- circle the terminals in the parse tree (the leaves).

Example II

Consider the following grammar for simple assignment statements:

```
<stmt> → <id> = <expr>
<expr> → <id> + <expr> |
         <id> * <expr> |
         ( <expr> ) |
         <id>
<id> → A | B | C
```

- Derive the string A = B * (A + C) and draw a corresponding parse tree
- Do the same for A = B * A + C

Example III

Consider the following grammar for a small language

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt>; <stmt_list>
<stmt> → <var> = <expression>
<var> → X | Y | Z
<expression> → <var> + <var> | <var> - <var> | <var>
```

Derive the string below and draw a corresponding parse tree

```
begin X = Y + Z; Y = Z end
```

Ambiguous Grammars

- A grammar is **ambiguous** if and only if it generates a sentential form that has two or more distinct parse trees.
- Your book has the following example grammar (Figure 6.4):

```
<expr> → <expr> + <expr> |
         <expr> * <expr> |
         ( <expr> ) |
         <number>
<number> → <number> <digit> | <digit>
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Construct `3 + 4 * 5` in two different ways.
  (Start with `<expr> + <expr>` and expand the second `<expr>` or start with `<expr> * <expr>` and expand the first `<expr>`.)
- Then let's do the parse trees; they are not the same.

Associativity of Operators as Defined by Grammars

- Operator associativity can be indicated by a grammar.
- First consider the following ambiguous grammar
  `<expr>` $\rightarrow$ `<expr> + <expr>` | `const`
- To see the ambiguity, develop leftmost and rightmost derivations of the expression.
  `const + const + const`
  taking each remaining leftmost (resp. rightmost) term to `const` before expanding another `<expr>`. Then compare the resulting parse trees.

Associativity of Operators as Defined by Grammars II

- Now consider the grammar:
  `<expr>` $\rightarrow$ `<expr> + const` | `const`
- No matter how you derive the expression
  `const + const + const`
  you get the same parse tree. Work it out.
- Trace the parse tree to determine the associativity.
- Notice that this rule is left-recursive, which causes the grammar to left-associate.

Operator Precedence as Defined by Grammars I

- Operator precedence can also be defined by grammars.
- Consider the following ambiguous grammar:

  `<E>` $\rightarrow$ `<id>` | `<E> / <E>` | `<E> - <E>`
  `<id>` $\rightarrow$ `X` | `Y` | `Z`

- To see the ambiguity, construct two parse trees based on leftmost (or rightmost) derivations for the expression

  `X - Y / Z`

  Hint: The first step matters.

Operator Precedence as Defined by Grammars II

- Now consider the grammar

  <E> → <F> | <E> - <F>
  <F> → <id> | <F> / <id>
  <id> → X | Y | Z

  and the expression X - Y / Z

- Any parse tree based on either a left- or a right-derivation for this yield the same tree.
- Note that this tree has / at a lower level than -, indicating the / is performed first

Grammar for Selection — if-then-else I

- Consider the following grammar that models the if-statement construct from Java

  `<if_stmt>` → `if (<logic_expr>) <stmt>` |
  `if (<logic_expr>) <stmt> else <stmt>`

- This grammar is ambiguous, which can be shown by finding two derivations which have differing parse trees for the statement

  `if (a < b) if (c < d) x = y else y = x`

  Here we assume that
  - `a < b` and `c < d` both parse as `<logic_expr>`
  - both of the assignment statements `x := y` and `y := x` parse as `<stmt>`
  - `<stmt>` can also be an `<if_stmt>`
- Let's do it. Can we tell which `if` the `else` belongs to?

Grammar for Selection — if-then-else II

- Consider now the following grammar for if-then-else

```
<stmt> → <matched> | <unmatched>
<matched> → if (<logic_expr>) <stmt> |
            <non-if statement>
<unmatched> → if (<logic_expr>) <stmt> |
        if (<logic_expr>) <matched> else <unmatched>
```

- You should show that the expression

```
if (a < b) if (c < d) x = y else y = x
```

will yield the same association. The `else` belongs to the second `if`.

The Parsing Process

A parser is a program that accepts a stream of terminals (usually from a lexical analyzer) and either accepts, or rejects, the stream as a sentence in an associated grammar.

Parsers fall into two categories:

1. **Bottom-up parser:** given an input stream, constructs a parse tree (or a logical equivalent) from the leaves to the root.
2. **Top-down parser:** given an input stream, builds a parse tree (or logical equivalent) from the root down.

Bottom-up Parsing

One type of bottom-up parser is a **shift-reduce parser**.

- The basic idea is to shift tokens from the input stream onto a stack until the right side of a production appears.
- If appropriate, the right side is reduced to the left side of the production.
- Repeat.
- Note: we don't always want to reduce every time the right-hand side of a production is on the stack. Sometimes it is necessary to continue to shift. Designing parsers from a grammar so shift-reduce decisions are made properly is a fundamental problem of bottom-up parsing.

Top-Down Parsing

- **Recursive-descent parsing** is straightforward to describe and nicely illustrates the relationship between a formal description of a programming language via a grammar and the ability to generate executable code for programs in the language.
- Grammars written in BNF are often difficult to use for automated processing (see Section 6.6 for an explanation); however, if we add a few simple notational niceties to BNF, we get a notation which is not only nice to use for grammar descriptions, but for automation as well.
- This notation, called **EBNF (Extended Backus-Naur Form)**, allows us to write rules which work well for a recursive-descent parser.

Extended Backus-Naur Form

EBNF is based on BNF but has constructs that avoid some of the more unnatural or awkward ways to specify simple syntactic properties.

- Optional parts are placed in square brackets [ ]
  `<proc_call>` → `identifier [(<expr_list>)]`
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  `<term>` → `<term> (+|-) const`
- Repetitions (0 or more) are placed inside braces { } (some versions require a trailing asterisk * or +, but our authors do not)
  `<identifier>` → `letter {letter | digit}`

Example of BNF vs. EBNF

- BNF

  <expr> → <expr> + <term> |
  <term>
  <term> → <term> * <factor> |
  <factor>

- EBNF

  <expr> → <term> {+ <term>}
  <term> → <factor> {* <factor>}

Example: An EBNF Grammar for Simple Assignment Statements:

<assign_stmt> → <var> = <arith_expr>
<arith_expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <var> | const | (<arith_expr>)
<var> → id | id\[<subscript>\]
<subscript> → <arith_expr>

An EBNF for Simple Integer Arithmetic Expressions

(Figure 6.18 on page 222 of textbook)

```
<expr> → <term> { + <term> }
<term> → <factor> { * <factor> }
<factor> → ( <expr> ) | <number>
<number> → <digit> { <digit> }
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The Recursive-Descent Process: Intro

- For each nonterminal in the grammar, define a function that can parse sentences generated by that nonterminal.
- EBNF is ideally suited for being the basis for a recursive-descent parser, because it reduces the number of nonterminals used in the grammar.
- Assume we have a lexical analyzer called Lex, which puts the next token code into variable `nextToken`.

The Recursive-Descent Process: One RHS

The coding process when there is only one RHS:

1. For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error.
2. For each nonterminal symbol in the RHS, call its associated parsing subprogram.

The Recursive-Descent Process: Multiple RHSs

The coding process when a nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse:

1. The correct RHS is chosen on the basis of the next token of input (the lookahead).
2. The next token is compared with the first token that can be generated by each RHS until a match is found.
3. If no match is found, it is a syntax error.

Recursive Descent Parser for Arithmetic Expressions

Example from textbook, pp. 228–230.

- Written in C.
- Parses simple integer arithmetic expressions such as `15 + 3 * (5 + 10)`
- The function `isdigit(char c)`, which is used in this code but not defined, belongs to the library ctype, which is imported into the code.

Tools for Parsing

- Lex is a lexical analyzer which is very useful for getting tokens from your input if you provide their description in the form of regular expressions.
- YACC (Yet Another Compiler Compiler) allows you to specify your language grammar in EBNF and does the parsing magically for you.
- So, if you want to write a compiler, or experiment with the syntax of the language that you designed, you can get Lex to process the input and then feed it to YACC to analyze further.

Outline

Object-Oriented Languages

Reading

- pp. 142–146 (finish first paragraph)
- Section 5.2.2 through p. 153.
- Section 5.3.3.
- Sections 5.6.1–5.6.2.

You are already familiar with the object-oriented paradigm, so

- we will review basic concepts and
- focus on language design issues of OO languages

Object-Oriented Paradigm

- Built around the idea of an object with properties and actions (or, perhaps, reactions).
- Objects react to messages from other objects.
- A program is a set of interacting independent objects.

Software Reuse and Independence

- OO languages are good for creating software components which are
  - reusable
  - independent
  - encapsulated
- The goal is to be able to design independent parts which others can use, while maintaining the option of changing the internal implementation, provided the interface does not change.
- Note that a language does not have to be object-oriented for programs to have these features, but OO languages are designed specifically to facilitate these goals.

Goal 1: Component Reuse

Four basic ways a software component can be modified for reuse:

1. extension of the data or operations
2. redefinition of one or more of the operations
3. abstraction
4. polymorphism

Extension of Data or Operations

- Suppose there's a basic definition of a Queue object.
- You may want to extend it by allowing for elements to be removed from its tail, rather than its head, if these operations are not included.
- What feature of OOs does this remind you of?

Redefinition of Operations

- The basic operations may be similar, but their manifestation may be specific.
- Example: text window vs. general-purpose window. Both display stuff, but a text window needs to account for displaying text.
- *application framework* — a collection of related software resources, usually in object-oriented form, that is used by software developers through redefinition and reuse to provide specific services for applications. In other words, the services are similar, but still need to be customizable.

Abstraction

- **abstraction** — the collection of similar operations for different components into a new component.
- Example: circle and rectangle are both objects that have position and that can be translated and displayed. These properties can be combined into an abstract object called a figure.

Polymorphism

- **polymorphism** — the extension of the type of data that operations can apply to.
- Examples of polymorphism are overloading and parameterized types.
- Example: The `toString` function should be applicable to any object as long as it has a textual representation.

Goal 2: Component Independence

- Restricting access to internal details of software components can help make sure that their implementation can be modified without affecting their users.
- As long as the interface remains unchanged and the component fulfils its promised function, the internals should not matter.

Reuse vs. Independence

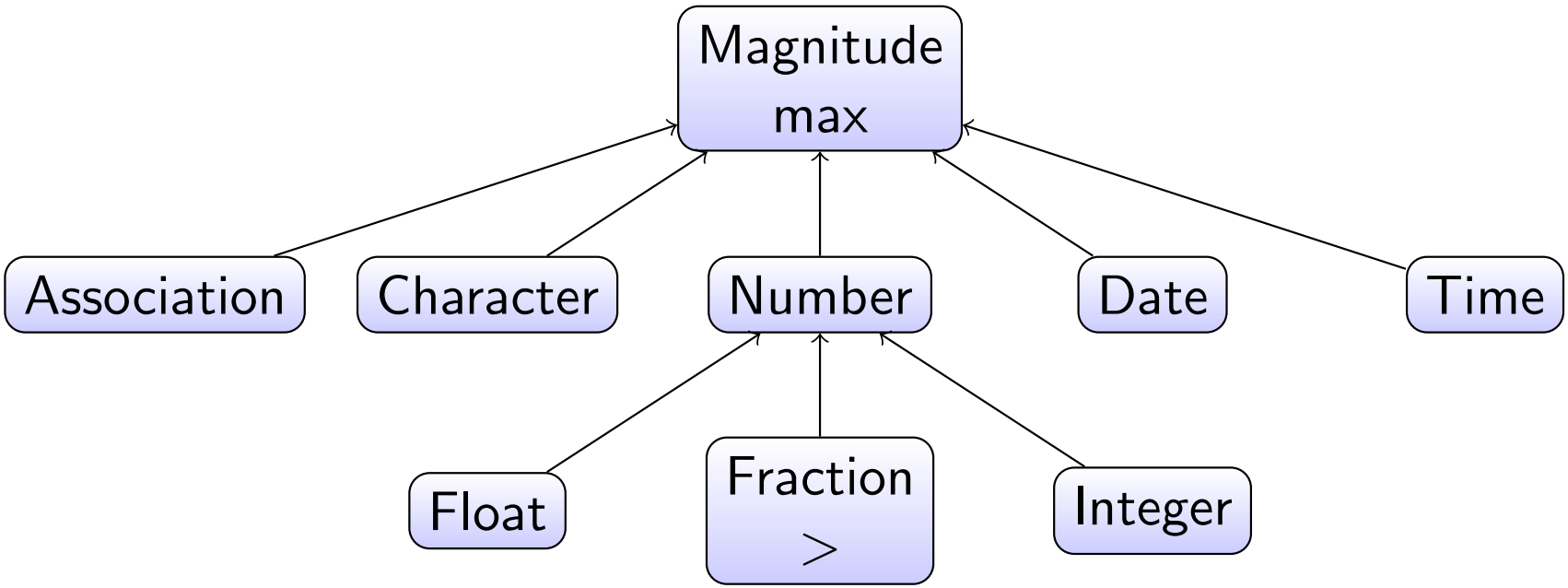
- Sometimes, the two goals are incompatible.
- Example: we mentioned adding operations to a Queue class. We would have to have access to the internals of the class to do this.
- Sometimes, the goals are complementary.
- When reuse and independence are explicit design goals, programmers are forced to consider their interface ahead of time, and abstract out the important properties of the class, rather than focusing on a task that is too specific.
- Chances are, if you think through your implementation, you will benefit from a more-abstract class yourself, and reuse your own code, instead of having to write yet another over-specific bit of code.

Basic OO Concepts

- message — a request for a service
- method — service performed in response to a message
- The sender of a message may also supply data with it (parameters).
- mutators — messages which result in a change of state in the receiver object.
- message passing — the process of sending and receiving messages
- interface — the set of messages that an object recognizes

Class Hierarchies

- Programming in OO languages often entails extending existing classes.
- The new subclasses inherit useful properties from the parent class.
- Below, subclasses of Magnitude inherit the `max` method; if invoked, it will rely on the $>$ method being defined in the subclass. (p. 153)



Dynamic Binding

(Section 5.3.3, pp. 175–176)

- When an object of a given class is instantiated and called upon to invoke a method, the proper method must be searched for.
- Most of the time this must happen at runtime because it is not always clear at compile time which method will be needed.
- Example:
  - Suppose we have a Collection class which allows for Lists, Stacks, etc., as subclasses.
  - This class has an `addAll` method which delegates the actual addition of individual elements to its respective subclasses by using `this.add(element)`.
  - It would not be clear at compile time what type of object `this` was since it's a generic method and the compiler would not know which `add` method was appropriate. (Do we add an element to a List, a Stack, ...?)
- This runtime binding is called **dynamic binding** as opposed to **static binding**, which happens at compile time.

Design Issues in Object-Oriented Languages

- OO languages (normally) dynamically bind methods to objects.
- Gain: extra flexibility.
- Loss: efficiency.
- Language design goal: introduce features which reduce the runtime penalty of dynamic binding.
- Languages have introduced the ability to label a method as `static` or `final` to allow static binding. (A `final` method cannot be overridden by a subclass; a `static method` is defined on a class, not an object.)
- Java uses a jump table instead of searching the class hierarchy to gain efficiency during dynamic binding.

Some General Notes on the Popularity of Java

- portability: virtual machine gave machine independence
- conventional syntax (C-like)
- improved code reliability
  - compile-time type checking
  - garbage collection and the use of references instead of explicit pointers (no more memory leaks because someone forgot to deallocate)
- encapsulation: support for packages

Outline

Reading

Chapter 7: Basic Semantics

- Read Chapter 7 through Section 7.7; i.e., pp. 256 – 309.

Overview

- In programming languages, a fundamental abstraction mechanism is the use of names (or identifiers) to denote language entities or constructs.
- You are familiar with the use of attaching names to variables as well as subprogram units such as procedures or functions.
- In the next chapter we will discuss procedures and functions as abstraction mechanisms for associating a name with a block of statements.
- In this chapter we shall examine issues related to the use of names with variables, and the closely-related concept of a named constant.

Variables as Abstractions of Memory Locations

- We noted in earlier chapters that imperative languages are abstractions of the von Neumann architecture, whose primary features are a central processing unit (CPU) processor and a memory that is separate from the CPU.
- The CPU, however, does contain its own small set of memory cells known as *registers* that assist the CPU in carrying out its tasks.
- In imperative programming languages variables are abstractions of memory cells and registers.

Variables: The Details

We can characterize a variable as a sextuple of attributes (N, T, L-value, R-value, LT, S), where:

1. N is the variable's name (the identifier);
2. T is the variable's type;
3. L-value is a reference to a location (address);
4. R-value is a value;
5. LT is the variable's lifetime — the time it is bound to a specific memory location.
6. S is the variable's scope — the range of statements over which the variable is accessible.

L-value and R-value come from the intended roles of the left and right side of an assignment statement:
`X := A` (`X` refers to an address, while `A` refers to a value.)

A Variable's Type

A variable's type determines it's

- range of values
- the set of operations that can work with it
- In the case of floating point, type also determines the precision.

Binding

- A **binding** is an association between an entity and an attribute, such as between a variable and its type or L-value or R-value, or between an operation and a symbol.
- Binding time is the time at which a binding takes place.
- **Static binding** — the binding is made before execution time and cannot be changed later.
- **Dynamic Binding** — binding is done during execution time and can be changed in accordance with language-specified rules.
- A **symbol table** can be thought of as a function which expresses the binding of attribues to names; in practice, it is some sort of data structure which helps us model this function to associate names with their meaning.

Binding Times: Language Differences

- Languages differ substantially in which attributes are bound statically or dynamically.
- Functional languages tend to have more dynamic binding than imperative languages.
- Static attributes can be bound during language design time, translation, linking, or loading of the program.
- Dynamic attributes can be bound at different times during execution, such as entry or exit from a procedure or from the program.

Static Binding of Variable Attributes: Examples

Possible attribute binding times for variable-related properties:

- **Language design time** — bind operator symbols to operations; values true/false bound to data type Boolean
- **Language implementation time** — bind floating point type to a representation
- **Compile time** — bind a variable to a type in C or Java
- **Load time** — bind a C or C++ static variable to a memory cell

Name binding — in compiled languages, name binding is generally done at translation time, either through an explicit declaration, or in some languages, implicitly upon first use.

Design Issues for Names

- Length of names, case sensitivity, use of special characters in name
- Are special words keywords or reserved words?
  - A **keyword** is a word that is special only in certain contexts. In many languages words such as if, begin, while, for, etc are keywords.
  - A **reserved word** is a special word — often a keyword — that cannot be used as a user-defined name.
  - Without reserved words, statements such as the following are valid in an language with an if-then-else conditional construct:
    `if if = then then then = else else else = if`

Static Type Binding

For static type binding, the type is typically bound at the same time the name is bound.

- Explicitly — by a variable declaration, e.g.
  ```
  int count;
  char initial;
  ```
- Implicitly — by the form of the identifier
  In FORTRAN 77 undeclared variable names beginning with I,J,K,L,M,N are integers, and real otherwise.

Dynamic Type Binding

For dynamic type binding, the type is usually implicitly determined by the current value of the variable.

Consider the following sequence of Python statements

```
value = 1.23    # value has real (float) type
value = 1       # value now has integer type
value = 1.0     # value now has real (float) type
value = "a "    # value now has char (or string) type
```

Binding of L-value (reference/storage allocation)

- In many languages the binding of an L-value is done at run-time, either upon explicit direction from the programmer (e.g. `malloc` in C) or automatically when execution enters the variable's scope.
- In some languages, however, the storage allocation may be static and remains bound to the same memory cell throughout execution, e.g., C and C++ static variables in functions.

Aliasing

Some languages allow the same reference to be bound to more than one variable — **aliasing**.

Example from Python

```
x = 5                          # x has its own memory address
y = x                          # y has its own memory address
list1 = [1, 2, [3, 4]]         # list1 references a location
                                 in ‘‘heap’’ storage that stores
                                 the list value shown
list2 = list1                  # list2 references the same heap
                                 storage location as list1
```

Binding of R-value (or simply value)

- The binding of an R-value is usually dynamic since by their nature variables are intended to have their values change during program execution via assignment statements.
- Some languages allow values to be frozen once they are made, creating a symbolic constant/named constant, manifest constant, etc.

Binding Constants

- Can be done at translation time — true static binding.
- Can be established at run-time, allowing the value to be determined by an expression involving other variables and constants.
  - C++ and Java: expressions of any kind, dynamically bound
  - C# has two kinds, **readonly** and **const**.
    - The values of **const** named constants are bound at compile time.
    - The values of **readonly** named constants are dynamically bound.

Initialization of Non-Constant Variables

There are various policies:

- Allow the programmer to initialize at declaration or run-time.
- Forbid accessing a variable's R-value until a meaningful value has been assigned by the programmer.
- Use a system-specified initial value.
- Accept what is in the L-value when it is bound.

Variable Lifetime

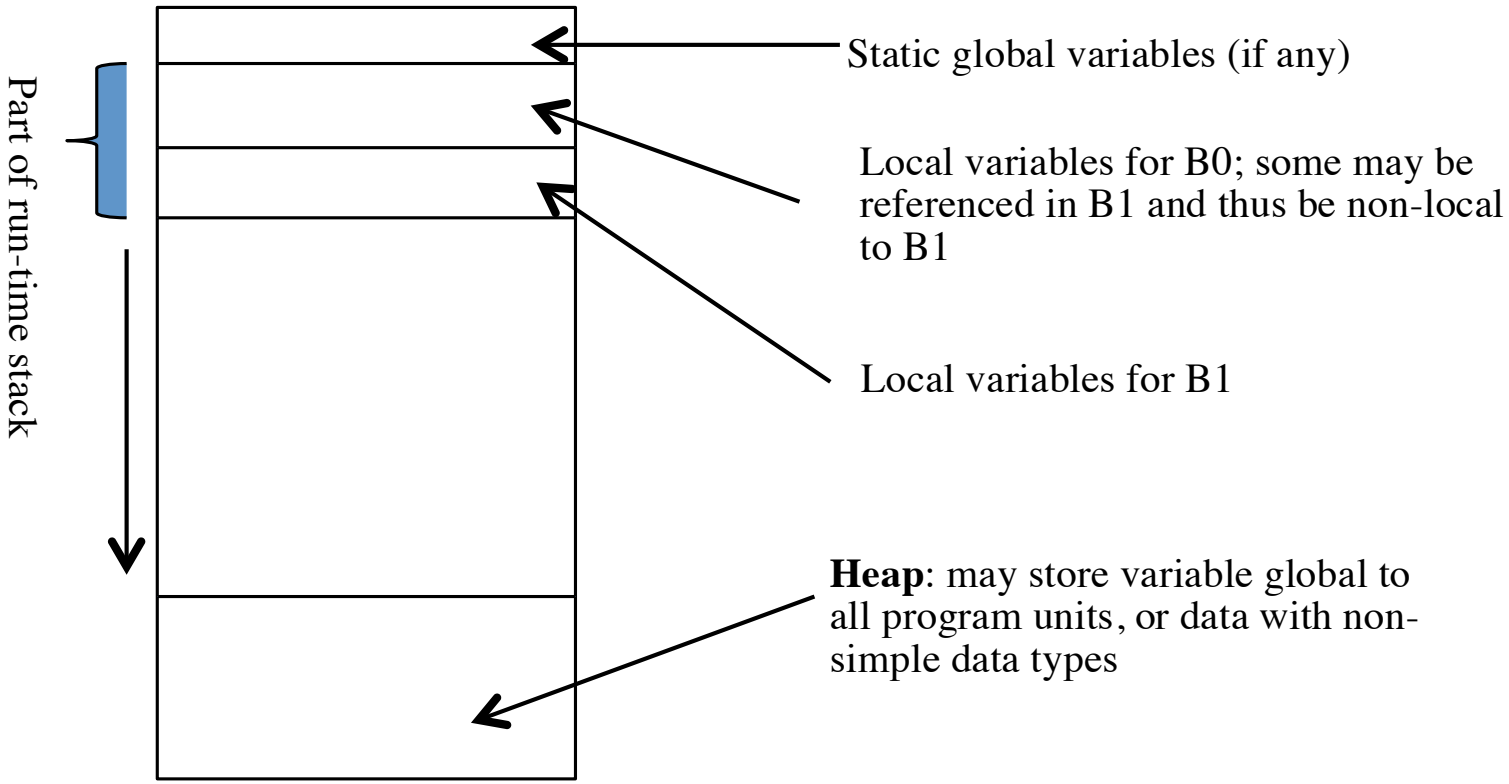
The **lifetime** of a variable is the time during which it is bound to a particular memory cell.

- **Allocation** — getting a cell from some pool of available cells
- **Deallocation** — putting a cell back into the pool

Heap and Stack Overview (more details later)

- When a program unit B1 begins execution, an area for its variables is created on a run-time stack. This area is pushed atop that of the program unit, say B0, that was running just before B1 was activated.
- When B1 completes its execution, the area for its variables will be popped from the stack and that for B0 will again be atop the stack.
- Global variables and allocations to pointer variables (and some local variables that have complex type values) will be allocated from another area of memory allocated to the program known as its "heap." Some implementations, however, may use a separate area for (static) global variables.

Memory Allocation



Categories of Variables by Lifetimes

- **Stack-dynamic** — Storage bindings are created for variables when their declaration statements are elaborated; a declaration is **elaborated** when the code associated with it is executed.
- **Explicit heap-dynamic** — Allocated and deallocated by explicit directives, specified by the programmer, which take effect during execution. A variable is referenced only through pointers or references; e.g.,
  - dynamic objects in C (via `malloc`)
  - dynamic objects via C++ (via `new` and `delete`),
  - objects in Java
- **Implicit heap-dynamic** — Allocation and deallocation caused by assignment statements. Examples:
  - Lists in Python;
  - all strings and arrays in Perl, JavaScript, and PHP;
  - pretty much all objects in any language that supports objects.

  Examples illustrating lifetimes (pages 289-291 of your textbook).

Scope

The scope of an variable is the range of statements over which it is "visible." Consider the code below (it resembles C, or C++, or Java, but is not intended to be any of these)

```
void big(){
   int x = 3;
   sub1();

   void sub1(){
      int x = 7;
      sub2();
   }
   void sub2(){
     int y;
     y = x;
    }
}
```

Which variable `x` is being referenced in `y = x` in `sub2`? The answer here depends on whether the program is using static or dynamic scoping, and on how "block structured" the language is.

Local Variables

The **local variables** of a program unit are those that are declared in that unit.

```
void sub() {
  int count;
  . . .
  while (some condition) {
    int count;  <-- this count is local to the while loop
    count++;
    . . .
  }
. . .
}
```

Non-Local Variables

- The **nonlocal variables** of a program unit are those that are visible in the unit but not declared there.
- **Global variables** are a special category of nonlocal variables.

Scope Rules

- The **scope rules** of a language determine how references to names are associated with variables.
- Based on our definitions of local, nonlocal, and global, we need to know what is meant by "program unit." For purposes of scope, the basic program unit is the **block**.
- A **block** is a section of code in which variables can be declared and whose scope is limited to those statements within the block.

C Scope: Example 1

Blocks in C are delimited by { }

```
(1)     if (list[i] < list[j]) {
(2)       int temp;
(3)       temp = list[i];
(4)       list[i] = list[j];
(5)       list[j] = temp;
(6)     }
(7)     for (int index = 0; index < 10; index++) {
(8)       printf(valArray[index]);
(9)     }
```

scope of **temp**

scope of **index**

C Scope: Example 2

What are the scopes of x, p, y, q, z, main, and w?

```
(1)   int x;

(2)   void p(){
(3)       char y;
(4)       ...
(5)   } /* p */

(6)   void q(){
(7)       double z;
(8)       ...
(9)   } /* q */

(10)  main(){
(11)      int w[10];
(12)      ...
(13)  }
```

Figure 7.4 Simple C program demonstrating scope

C Scope: Example 2 Elaborated

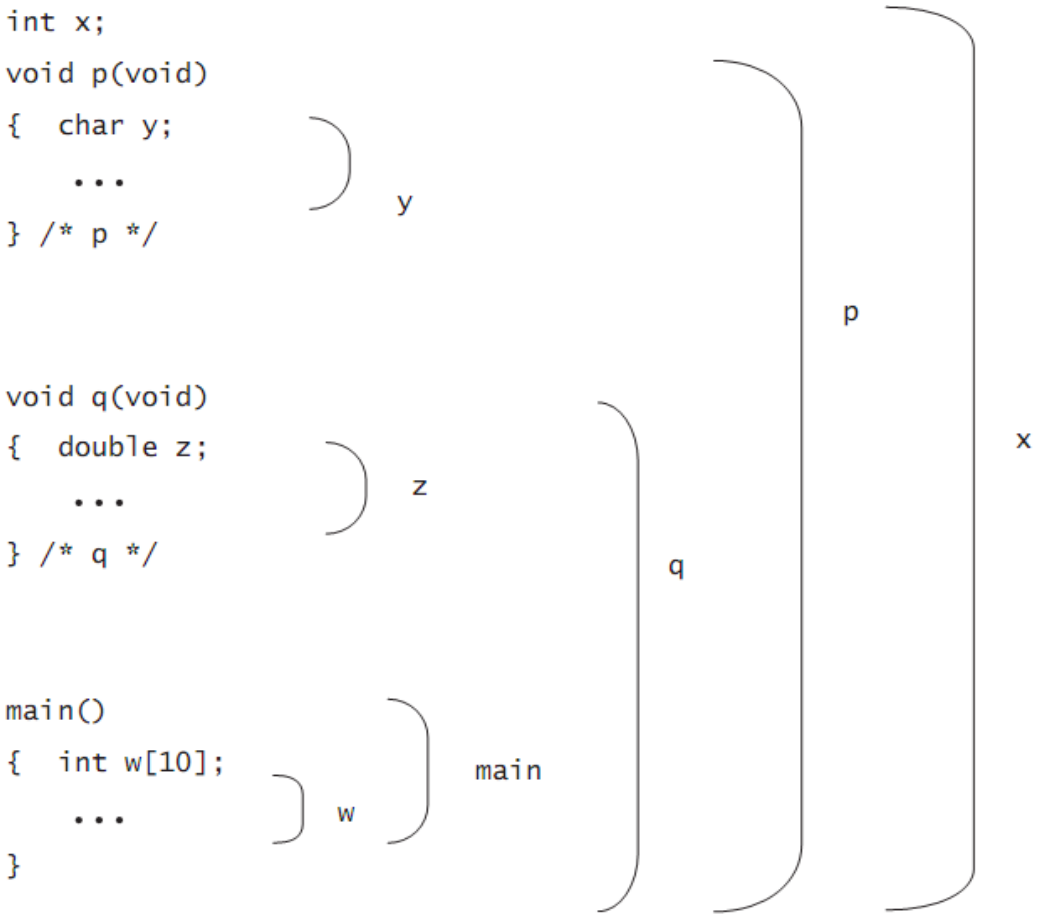


Figure 7.5 C Program from Figure 7.4 with brackets indicating scope

Additional Examples

- Additional examples of blocks of code in different languages can be found on pages 261-263 and pages 265-267 of your textbook.

Scope Holes

A global variable is said to have a **scope hole** in a block containing a local declaration with the same name.

- In C++ one can use the scope resolution operator :: to access the global variable.
- Other languages have similar ways of addressing scope holes. (see page 267 of the textbook).

Static Scope (Lexical scope)

Static scope, also known as **lexical scope**, requires an examination of the program code. To connect a name use to a variable, you (or the compiler) must find the appropriate declaration:

1. Look inside the immediate block of the statement first for a declaration with that name. If found, that is the applicable declaration; otherwise go to step 2.
2. Look inside the block enclosing the block for a declaration of the name used. If found, that is the applicable declaration; otherwise go to step 3.
3. Look inside ever-enclosing blocks until the first declaration of the name used is found. If none is found, then the name use is invalid.

Example of Static Scope

```
int main(){
  int a;
  int w;
  w = 5;
  a = f(w);
}

int f(int a){
  int w;
  w = 2*a
  return w;
}
```

Global Scope

Declarations that appear outside of all functions create variables that are global to all of the functions.

```
int w;              <-- w is global to both main and f.
void main(){
  int a;
  w = 5;
  a = f(w);
}
int f(int a){
  int b;
  b = w + 2*a
  return b;
}
```

Python Globals

In Python, a global variable can be referenced in functions, but can be assigned in a function only if it has been declared to be global in the function.

```
def f():
    global s
    print s
    s = "Now we can re-assign it."
    print s

s = "First, global assignment."
f()
print s
```

Output

```
First, global assignment.
Now we can re-assign it.
Now we can re-assign it.
```

Dynamic Scope

- Dynamic scoping is based on calling sequences of program units, not their textual layout (temporal versus spatial).
- References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point.

Tricky Effects of Dynamic Scoping

With dynamic scope rules:

- While a subprogram is executing, its variables are visible to all subprograms it calls.
- It is impossible to statically type check.
- It is not possible to statically determine the type of a variable.

Dynamic Scoping: Example

Consider the following subprogram structure

```
void big(){
   int x = 3;
   sub1();
   void sub1(){
      int x = 7;
      sub2();
   }
   void sub2(){
     int y;
     y = x;
    }
}
```

- With static scoping: the reference to `x` in `sub2` is to `big`'s `x`
- With dynamic scoping: the reference to `x` in `sub2` is to `sub1`'s `x`

Outline: Subroutines: Concepts and Implementation

Reading

Chapter 10: pp. 444–459 and pp. 462–465

Subroutines

- A subroutine (procedure, function, subprogram) is a mechanism in programming languages for abstracting a group of actions or computations.
- By giving a name to this abstraction, the actions or computations can be activated by simply using the name in a desired context.
- Such an activation is known as **invoking** the subroutine.
- The group of actions is known as the **body** of the subroutine.

Procedures vs. Functions

Many languages make strong syntactic distinctions between procedures and functions.

- Procedures are collection of statements that define parameterized computations. They return no values and each carries out its intended action by producing **side effects** in the environment from which it was invoked.
- Functions structurally resemble procedures but are semantically modeled on mathematical functions. They return a value back to the environment from which it was invoked.

Example: Ada — Procedures vs. Functions

```
-- Ada procedure
procedure swap ( x,y: in out integer) is
  t: integer;
begin
  if (x = y) then return;
  end if;
  t := x;
  x := y;
  y := t;
end swap;

-- Ada function
function max ( x,y: integer ) return integer is
begin
   if (x > y) then return x;
   else return y;
   end if;
end max;
```

Keeping Track of Subroutine Calls

- Semantically, a subroutine is a block whose declaration is separated from its execution.
- The **run-time environment** determines memory allocation and maintains the meaning of names during execution.
- Memory allocated for keeping track of local objects of blocks is called the **activation record** (or **stack frame**)
- The block is said to be **activated** as it executes.

Subroutine Definition

```
// C++ code
void intswap (int& x, int& y){ // specification
    int t = x;    // body
    x = y;        // body
    y = t;        // body
}
```

- A subroutine is defined by providing a **specification** (or **interface**) and a body
- **Specification (header)**: includes the subroutine's name, parameter profile (the number, order, and types of its parameters) and the return value type (if any).
- Aside: Discuss difference with C and design decision to use & to signal variable parameters.

Subroutine Calls

- You **call** (or **activate**) a subroutine by stating its name and providing arguments to the call which correspond to its formal parameters.
- A call to a subroutine transfers control to the beginning of the body of the called subroutine (the **callee**).
- When execution reaches the end of the body, control is returned to the **caller**.

Program Control Transfer

- Each subroutine has a single entry point.
- The calling program is suspended during execution of the called subroutine.
- Control always returns to the caller when the called subroutine's execution terminates.
- Control may be returned before the end of the body of the subroutine by using a `return` statement. What are the pros and cons of using a return statement?

Evaluate This Return

```
// C++ code
void intswap (int& x, int& y){
    if (x == y) return;
    int t = x;
    x = y;
    y = t;
}
```

Subroutine Communication with Caller

A subroutine communicates with the rest of the program through its parameters and also through **nonlocal references** (references to variables outside the procedure body).

- **A formal parameter** is a dummy variable listed in the subroutine header and used in the subroutine.
- An **actual parameter** represents a value or address used in the subroutine call statement.
- **Scope rules** that establish the meanings of nonlocal references were covered in Chapter 5.
- There may be reasons for altering variables other than ones explicit in the interface, but this is a measure of last resort. Normally, this leads to disasterous consequences and is only done if the language you are using does not provide the support you need, and you can't switch languages.

Actual/Formal Parameter Correspondence

1. **By Position**: The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth.
   — Used in most languages.
2. **By Keyword**: The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter.
   - *Advantage:* Parameters can appear in any order, thereby avoiding parameter correspondence errors. Useful when there is a large number of parameters. Very useful when most of them have default values and are optional.
   - *Disadvantage:* User must know the formal parameter's names.

Example

- Suppose summer is a subroutine that is defined with parameters `length`, `list`, and `sum` in this order.
- When invoking by keyword, the following is permitted

  ```
  summer(list=mylist, length=numberToSum,
  sum=total)
  ```

  whereas, when invoking by position one would use

  ```
  summer(numberToSum, mylist, total)
  ```

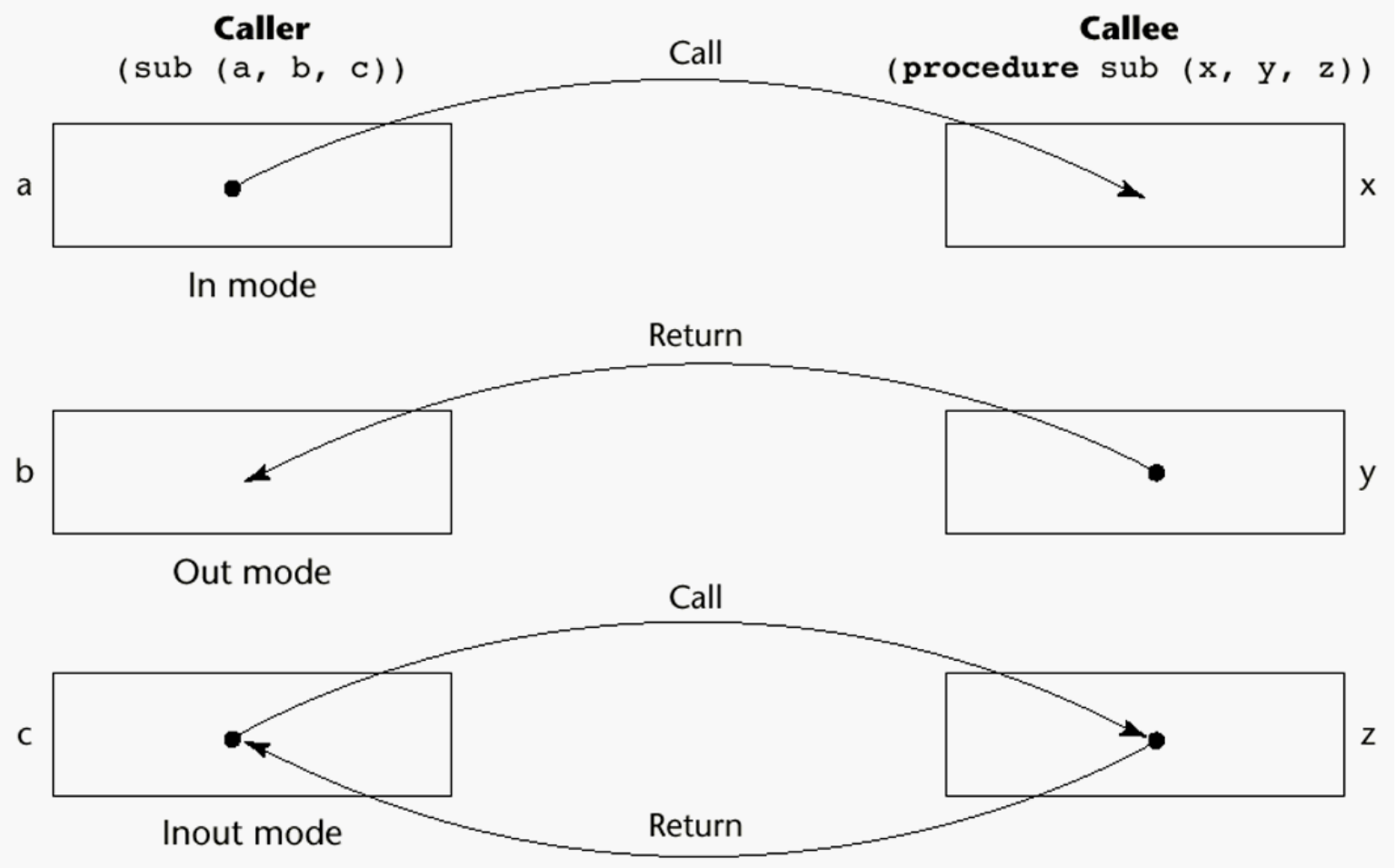
Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)
- In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)

Semantic Modes of Parameter Passing

- The nature of the bindings of arguments to parameters affects the semantics of subroutine calls.
- Languages differ significantly in the kinds of parameter-passing mechanisms available and the range of permissible implementation effects.
- Three broad categories of mechanisms for parameter-passing: **in mode**, **out mode**, and **in-out mode**.

Parameter Passing Modes



In mode, or Pass-by-Value

Actual parameters are evaluated at time of call, and their values become the initial values of the formal parameters.

- The most common mechanism for parameter passing
- In the simplest form of pass by value, value parameters behave as constant values during execution of the procedure.
- This mechanism is usually the only mechanism used in functional languages.
- Pass by value does not imply that changes outside the procedure cannot occur through the use of parameters. A pointer or reference type parameter contains an address as its value, and this can be used to change memory outside the procedure.

Out Mode, or Pass-by-Result

- No value is transmitted to the subroutine; the corresponding formal parameter acts as a local variable and its value is transmitted to the caller's actual parameter by copying when control is returned to the callee
- Requires extra storage locations and a copy operation for parameters with complex data types.

In-Out Mode: Three Types

There are three types of in-out mode:

1. Pass by reference
2. Pass by value-result
3. Pass by name

In-Out Mode: Pass by Reference

- This mechanism passes the location (reference) of the actual parameter variable as the value of the formal parameter, making the formal parameter an alias for the actual parameter.
- Any changes to the parameter occur to the argument as well.

In-Out Mode: Pass by Value-Result

- A combination of pass-by-value and pass-by-result.
- The value of the argument is copied and used in the procedure.
- Pass by value-result can only be distinguished from pass by reference when using aliasing.

In-Out Mode: Pass by Value-Result: Example

(Example uses C syntax only for convenience.)

```
void p(int x, int y) {
  x++;
  y++;
}

main(){
  int a = 1;
  p(a,a);
  /* a == 3 if passed by reference;
     a == 2 if passed by value-result
   */
  . . .
}
```

In-Out Mode: Pass by Name

- Values are passed from the caller to the callee by textual substitution.
- The argument is not evaluated until its actual use as a parameter in the called procedure.
- Changes inside the procedure can affect the argument.

In-Out Mode: Pass by Name: Example

```
int i;
int a[10];

void inc(int x){
  i++;
  x++;
}

main() {
  i = 1;
  a[1] = 1;
  a[2] = 2;
  inc(a[i]);
  return 0;
}
```

In-Out Mode: Pass by Name: History

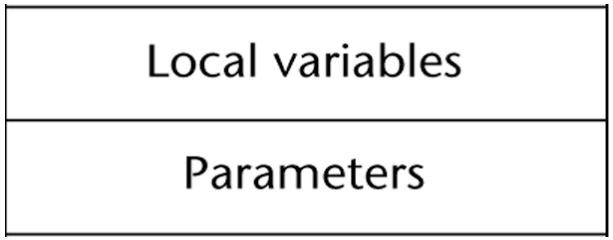
- Introduced in Algol 60 as a kind of inline process for procedures.
- Difficult to implement.
- Complex interractions with arrays and assignment.
- Removed from Algol60 descendants.
- Possibly of interest for languages without side-effects.

Implementing Subroutines

- Semantically, a subroutine is simply a block whose declaration/definition is separated from its execution.
- Normally the code and the data for subroutines are separate from each other.

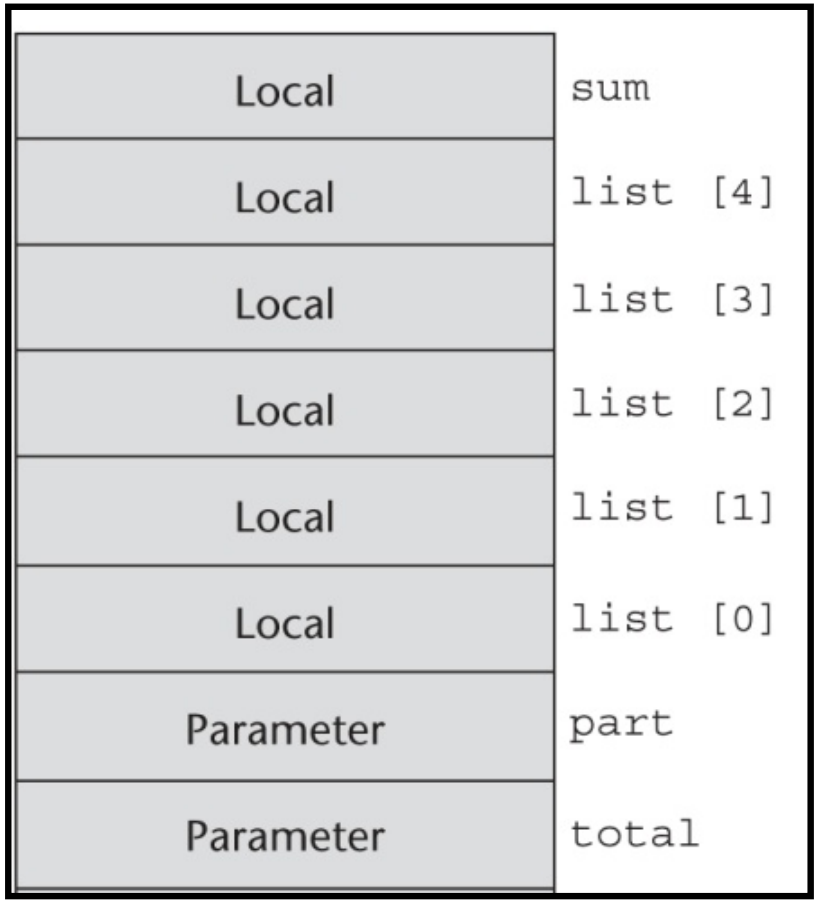
Memory Allocation for Subroutine Data

- The memory allocated for local objects of a block are collectively called the **activation record** (or **stack frame**).
- When applied to a subroutine that has parameters, the activation record must be augmented to accommodate the parameters of a subroutine so that they can be accessed while the subroutine is executing.
- Common organization for a subroutine activation record:

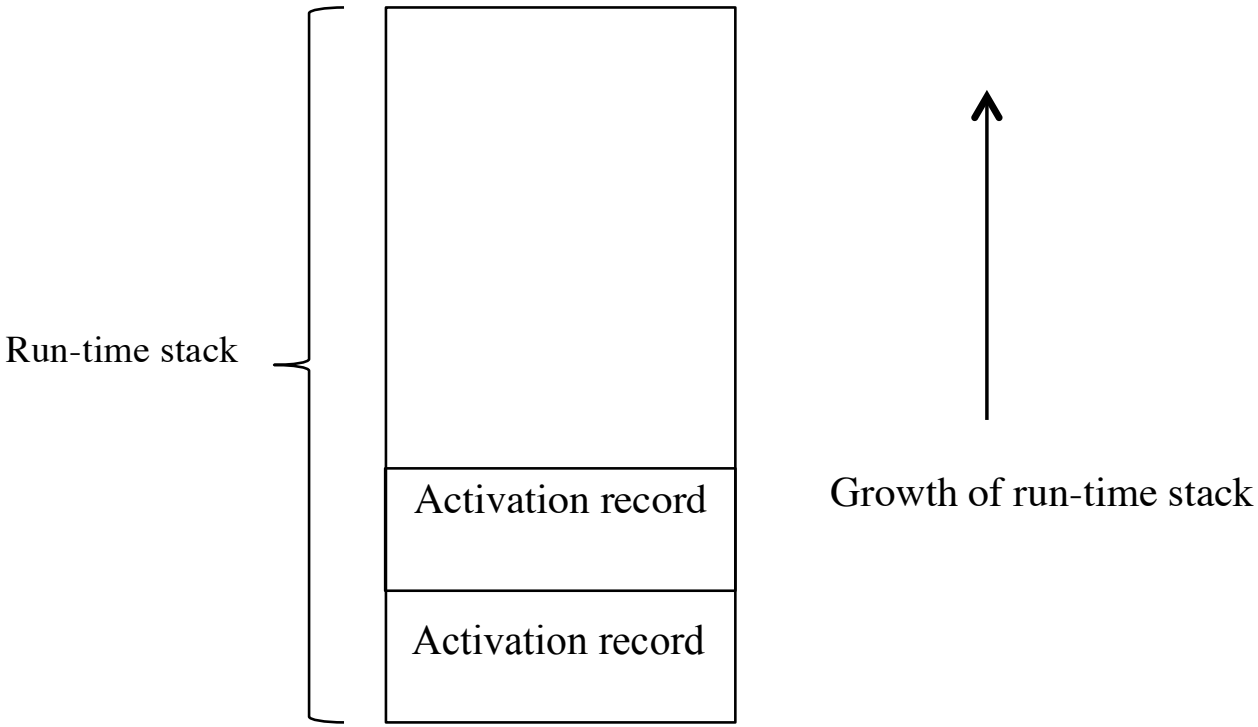


Subroutine Activation Record Example

```
void sub(float total,
         int part){
  int list[5];
  float sum;
  ...
}
```



The idea behind instantiating the data at run-time is that as subroutines are called and then complete their execution, the activation records for them are pushed and popped from a run-time stack.



Cleaning Up the Stack

Problem:

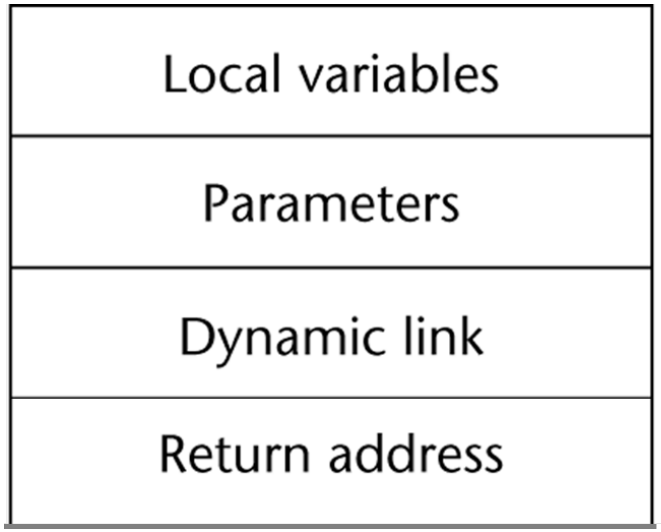
- Once a called subroutine completes and control of execution is returned to the callee, how do we know the base location of the called program's activation record?
- It is not a matter of moving down a pre-determined number of locations, since the size of the activation records of subroutines will vary.

Cleaning Up the Stack

Solution:

Augment the activation record with two additional fields:

- One containing the value of the **return address** of the calling function.
- One containing a link to the activation record of the calling function (the **dynamic link**).



The Environment Pointer

- During program execution the run-time system maintains a reference, known as the **environment pointer**, to the activation record instance of the currently executing program unit.
- It is this value that goes into the dynamic link field if the currently executing subroutine calls another.
- When the called program concludes, the value of the dynamic link also gets copied into the environment pointer.
- This is, in essence when the activation record of the called program gets popped from the stack.

Call/Return Actions

Before proceding to examples, we'll look at a summary of actions carried out by the run-time system which implement calls and returns to subroutines.

1. Caller actions
2. Initial actions of the called subroutine
3. Concluding actions of the called subroutine

Caller Actions

1. Create an activation record instance.
2. Save the execution status of the current program unit: register values, CPU status bits, environment pointer value.
3. Compute and pass the parameters.
4. Pass the return address to the called.
5. Transfer control to the called.

Initial Actions of the Called Subroutine

1. Save the old environment pointer in the stack as the dynamic link and create the new value.
2. Allocate local variables.

Concluding Actions of the Called Subroutine

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters.
2. If the subroutine is a function, its return value is moved to a temporary location accessible to the caller.
3. Restore the stack pointer by setting it to the value of the current EP-1 and set the EP to the old dynamic link.
4. Restore the execution status of the caller.
5. Transfer control back to the caller.

Example

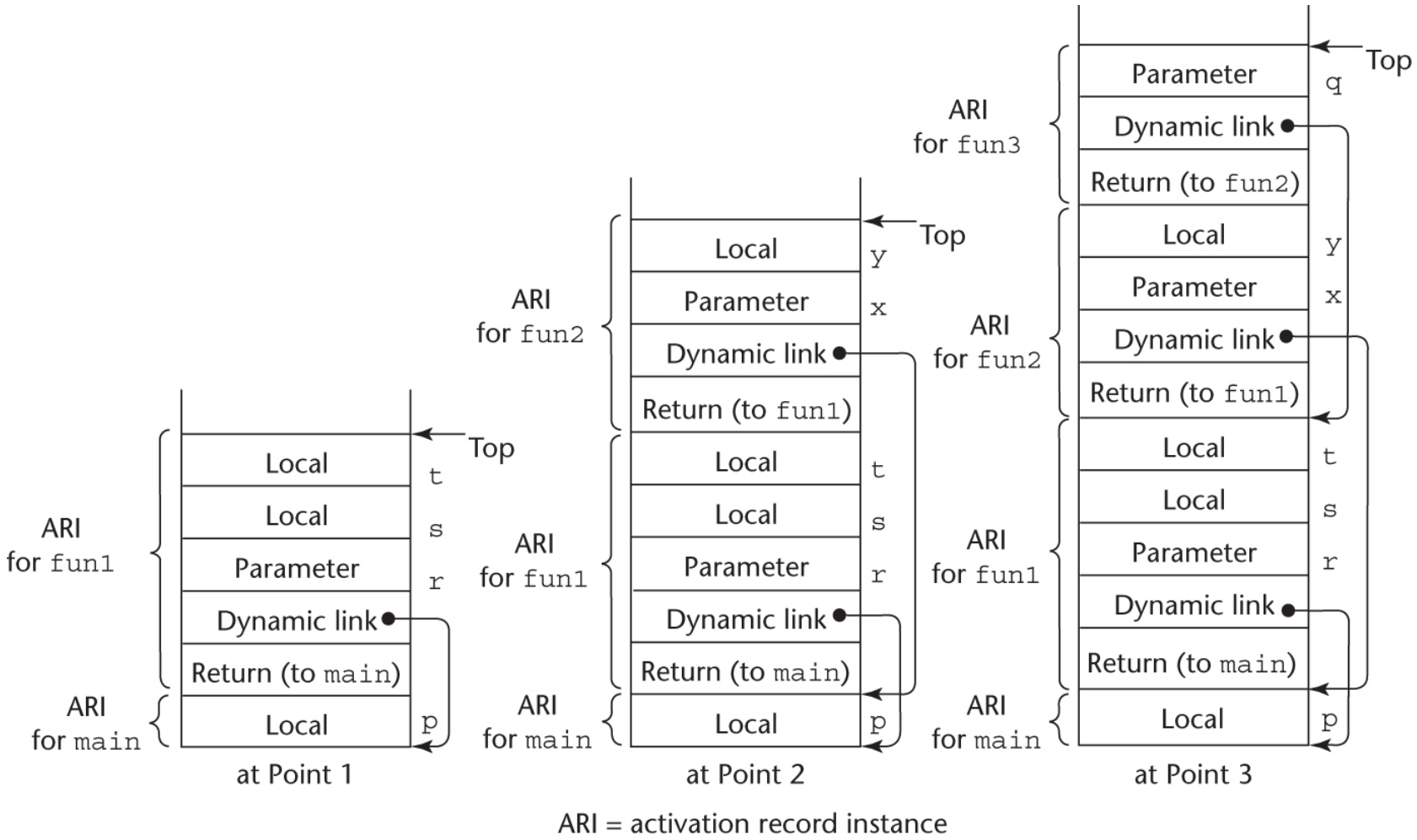
Consider the following set of subroutine definitions:

```
void fun1(float r) {
int s, t;
...
fun2(s);
...
}
void fun2(int x) {
int y;
...
fun3(y);
...
}
```

```
void fun3(int q) {
...
}
void main() {
float p;
...
fun1(p);
...
}
```

Here `main` will call `fun1`, which will call `fun2`, which will call `fun3`.

With an activation record organization as just described, the run-time stack would change as follows during program execution:

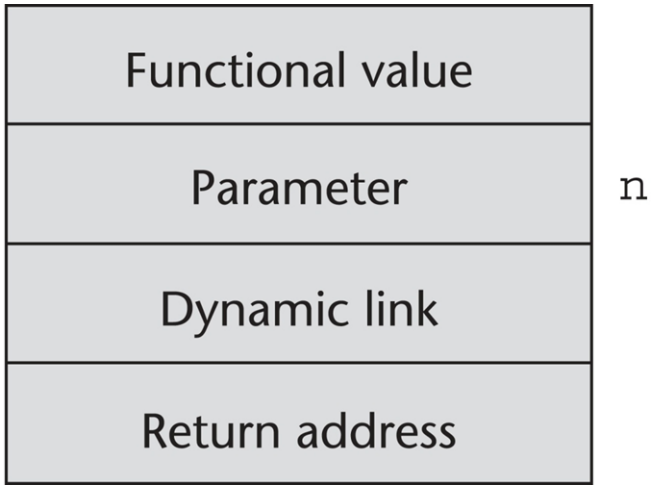


Recursion and the Run-Time Stack

The activation record structure used in the previous example readily supports recursion. Example:

```
int factorial (int n) {
    if (n <= 1) return 1;
    else return (n * factorial(n - 1));
}
```

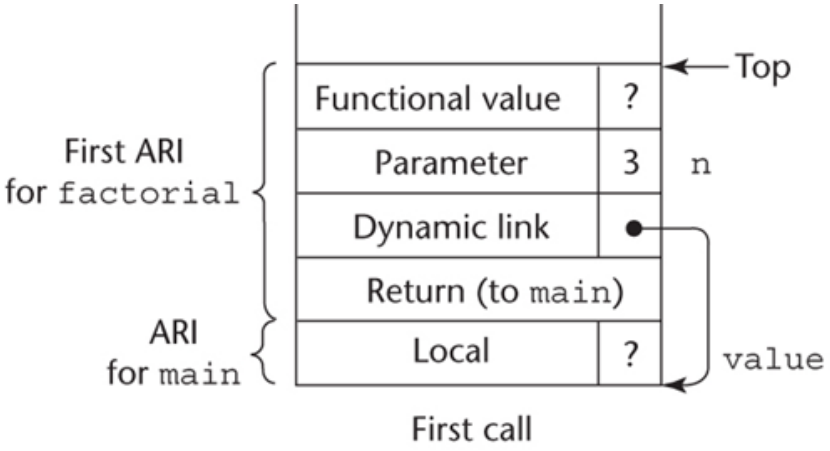
Activation record being augmented by a field representing the return value:



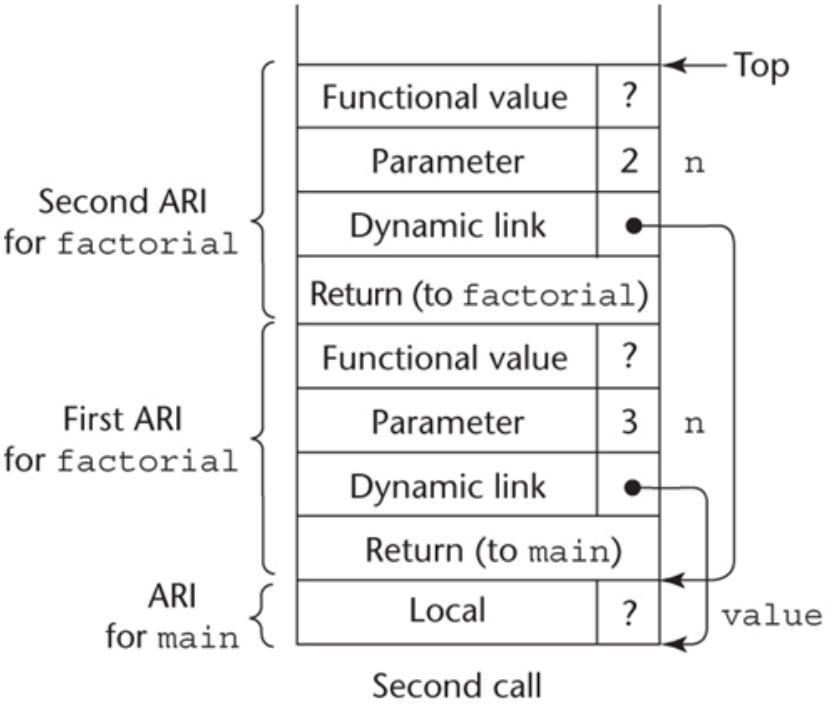
Now let us look at the change in the run-time stack when the call `factorial(3)` is made

```
int factorial (int n) {
            <-----------------------------1
    if (n <= 1) return 1;
    else return (n * factorial(n - 1));
            <-----------------------------2
}

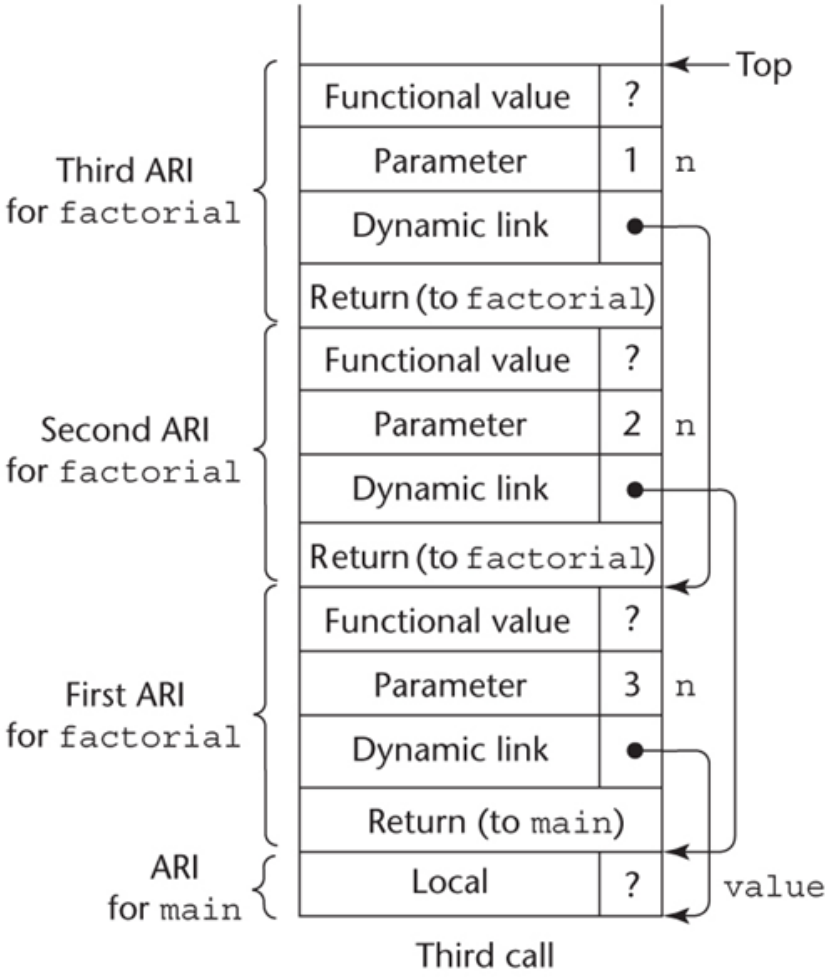
void main() {
    int value;
    value = factorial(3);
            <-----------------------------3
}
```



First call



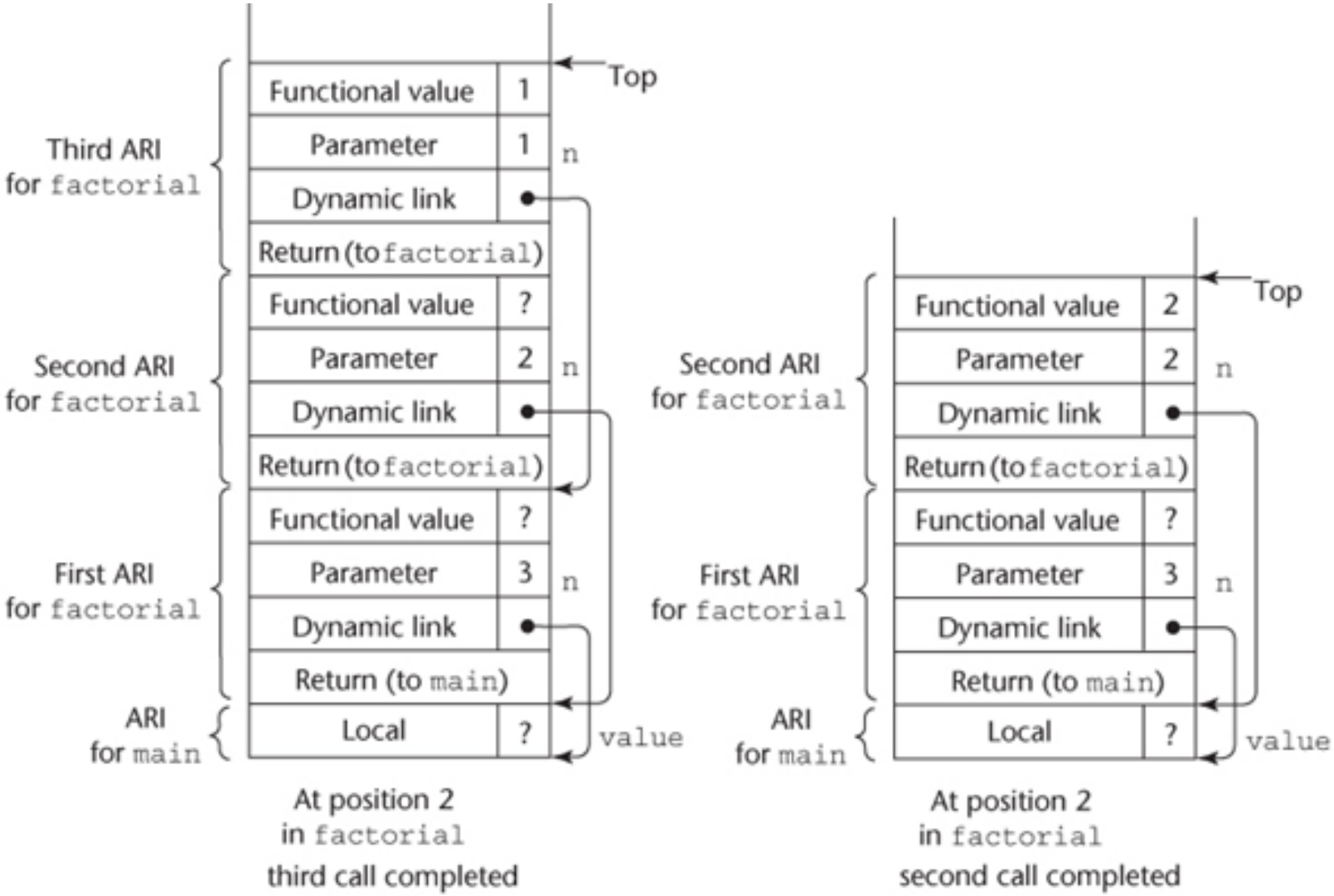
Second call



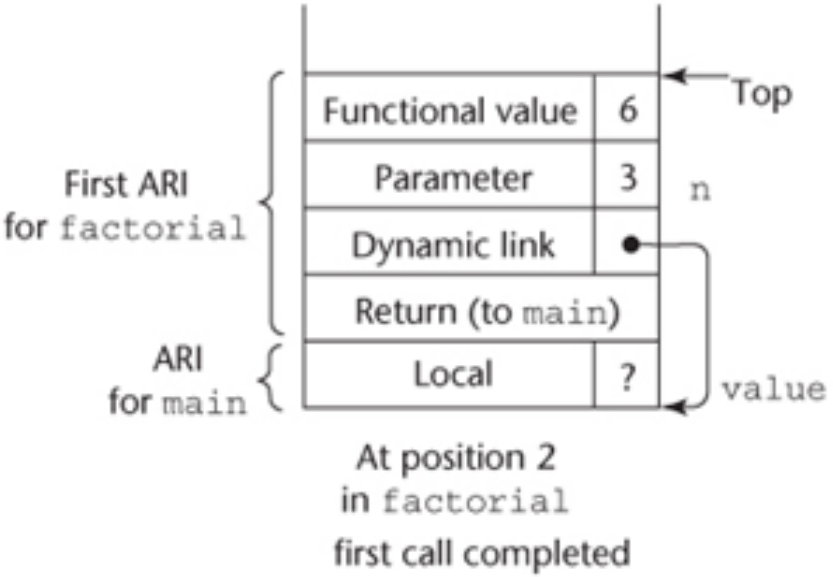
Third call

ARI = activation record instance

Cleaning the Stack: 1



Cleaning the Stack: 2



At position 2
in `factorial`
first call completed

In position 3
in `main`
final results

Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the **dynamic chain**, or **call chain**.
- A local variable can be accessed by its offset from the beginning of the activation record, whose address is in the environment pointer.
- This offset is called the **local offset**.
- The local offset of a local variable is normally determined by the compiler at compile time.

Locating a Non-Local Reference

First requires finding the correct activation record instance of the non-local reference:

- Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made.
- The approach for locating the correct activation record is different depending on whether dynamic or lexical scoping is used.
- Dynamic scoping can just follow the dynamic chain to find the right varible.
- Lexical scoping requires an extra pointer, called the **static link** to access the proper variables.