Chapter 1 Exercises

Chapter overview.

Chapter 1 presents an introduction to the study of programming languages. The authors spend much of the chapter discussing the software development process and the role of programming languages in it and finish with of the programming language concepts that they intend to present in the remainder of the book.

Problems, solutions and discussions.

3. Write a program that shows why the optimization mentioned in Section 1.5.3 cannot be done in general for C.

Solution

(See the example from Section 1.5.3 in the discussion below.)

Assume that \( z \) is an integer global variable and that \( x \) and \( y \) are integer variables (although they could be any numeric type). Further, assume that \( \text{fun}(y) \) is

\[
\text{int fun (int y) } \{ \\
\quad z = y + z; \ \\
\quad \text{return } z; \ \\
\}
\]

Then, \( x = \text{fun}(y) + z + \text{fun}(y) \); would set \( x = (y + z) + (y + z) + (y + (y + z)) = 4y + 3z \), while \( x = 2\times\text{fun}(y) + z \); would set \( x = (2 \times (y + z)) + (y + z) = 3y + 3z \).

Discussion

The example in Section 1.5.3 says

… in general, a statement like

\( x = \text{fun}(y) + z + \text{fun}(y) \);

in C cannot be optimized as

\( x = 2\times\text{fun}(y) + z \);

which would presumably be more efficient since it would call the function \text{fun} just once. *The language feature that allows functions to modify global variables (like \( z \) in the example) disallows the optimization.* ([1, p. 13], emphasis added.)

As stated, by allowing functions to modify global variables as a side effect of their execution these two statements might produce very different results. Similar mismatched results could be obtained when the function manipulates pointers. In general this optimization cannot be made because of the potential side effects resulting from function calls and correct answers will demonstrate this understanding.
4. **Provide a succinct characterization of imperative vs. nonconventional (functional and logic) languages.**

**Solution**

Programs written in imperative (conventional) languages tell the computer how to do a computation, while programs written in nonconventional languages tell the computer what computation should be done without specifying how to do it.

**Discussion**

This question imposes one classification on programming languages (see [1, p. 7] for the authors’ list of paradigms), but there are other equally valid classifications. For example Scott [2, p. 5] divides languages into two top level categories (declarative and imperative), each with subcategories as shown here:

<table>
<thead>
<tr>
<th>Declarative Languages</th>
<th>Imperative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>von Neumann</td>
</tr>
<tr>
<td>Lisp/Scheme, ML, Haskell</td>
<td>Fortran, Pascal, Basic, C, …</td>
</tr>
<tr>
<td>Dataflow</td>
<td>Object-Oriented</td>
</tr>
<tr>
<td>Id, Val</td>
<td>Smalltalk, Eiffel, C++, Java</td>
</tr>
<tr>
<td>Logic, Constraint based</td>
<td></td>
</tr>
<tr>
<td>Prolog, VisiCalc</td>
<td></td>
</tr>
</tbody>
</table>

Both classifications are based on similar principles. Specifically, imperative programming languages present the programmer with an abstraction of the underlying von Neumann computational model. Programs consist of an algorithmic (step-by-step) sequence of instructions whose execution changes the state of the computation by modifying a repository of values.

Non-imperative programming languages (the authors’ consider declarative languages a paradigm alongside functional and logic languages) are based on mathematical foundations rather than on the technology of the underlying hardware: the theory of recursive functions and mathematical logic [1, p. 9].

5. **Take one or two languages you have used and discuss the types of expressions you can write in these languages.**

**Solution**

Based on the authors’ description of expressions [1, pp. 27-28], correct answers to this problem should address some of the following:
• What are the semantic and syntactic rules specified by the languages for building expressions?
• Do the languages provide the ability for an expression to produce values of different types at different times (overloading)?
• What operators are provided by the languages and on what types do they operate?
• Do the languages allow assignment to be a constituent of other expressions?
• How do the languages deal with order of operation and operator precedence and associativity?

Discussion

What are the semantic and syntactic rules specified by the languages for building expressions?

This covers a lot of ground. One thing worth mentioning is the kind of notation used in evaluating expressions. Infix notation \((a * (b + c))\) is used in procedural and object-oriented languages like C, C++, Java, Pascal, and FORTRAN. Prefix notation \((*a + b c)\) is used in functional languages like Lisp and Scheme. Postfix notation \((a b c + *)\) is used in stack based languages like Postscript and FORTH. Students might also note that some languages mix notations as in C where although infix is primarily used, unary postfix \((i++)\) and prefix \((++i)\) are also common.

One might also talk about order of operation in the languages and show how that can affect the result of an expression. For example, consider the binary numerical operators * and +. in C, * has precedence over + so the expression \(1 + 2 * 3\) evaluates to 7. In Smalltalk, binary operators use a left-to-right ordering so \(1 + 2 * 3\) evaluates to 9.

Do the languages provide the ability for an expression to produce values of different types at different times (overloading)?

There are two kinds of overloading commonly found in programming languages. User definable overloading as provided by C++ provides a method for the programmer to modify the behavior of the language on a case by case basis. So if it makes sense to overload the + operator for string concatenation, the programmer can do so. The second form of overloading is much more common and, in fact, is found in a large majority of languages. It is this form that allows the * math operator to provide both integer multiplication as well as floating point multiplication. Most programmers don’t even think about how this is implemented (if indeed they even know that overloading is taking place) but it standardizes the use of operators across a variety of types and frees the programmer from worrying about it.

A non-overloading approach to this question would be to consider the \(? :\) form of the if-then expression in C. This expression is capable of producing values of different types. Assume \(z\) and \(a\) are integers. Then \(z = (a > 0) ? 5 : \text{‘b’}\) would assign either an int or a char to \(z\) depending on the value of \(a\).
What operators are provided by the languages and on what types do they operate?

A look through the language standards or reference books should provide information not only about mathematical operators but also about operations for all of the types provided by a language. This includes not only unary and binary mathematical operators, but boolean comparison and associative operators, character and string operators, bitwise logical operators, and operations on other types such as assignment between structs in C or set inclusion in Pascal. Students could talk here about the types which operators accept as operands and the types which they produce which are often different. Comparisons of the syntax of operators would also be appropriate (for example the “not equal” operator is <> in Pascal, != in C, and /= in Ada).

Do the languages allow assignment to be a constituent of other expressions?

Assignment statements in C and C++ produce a value. This side effect means that they can be used as part of other expressions. For example, in C a = b = c + d will assign the sum of c and d to b and then assign the same value to a. In Pascal this would require two statements (b := c + d; a:= b). Pascal draws a sharp distinction between assignment statements and expressions [1, p.177].

How do the languages deal with order of operation and operator precedence and associativity?

As mentioned previously, Smalltalk relies on a strict left-to-right order of operation while languages like Ada evaluate by operator precedence. In C most operators associate left-to-right, but =, +=, and -= associate right-to-left. Parenthesizing can be used to modify operator precedence (parentheses being operators themselves in some languages).
Chapter 2 Exercises

Chapter overview.

Chapter 2 covers syntax and semantics. The topics which the authors stress are language definitions (2.1), language implementation and binding (2.2), variables (2.3), routines (2.4), and operational semantics (2.6 and 2.7).

Problems, solutions and discussions.

7. Can the l-value of a variable be accessed only when its name is visible (i.e., within scope)? Why or why not?

Solution

No. Languages like Pascal and C allow the creation of unnamed variables that are manipulated through either references or pointers to their l-values.

Discussion

By allowing the r-value of one variable to refer to the l-value of another, languages create a method of indirect reference. The authors state [1, p. 59] that variables that are accessed by pointers may even be unnamed. Two Pascal examples of this are provided in the text [1, p. 60].

9. In general, it is not possible to check statically that the r-value of an uninitialized variable will not be used during program execution. Why?

Solution

If the r-value of an uninitialized variable is used in a branch of a conditional statement and the branching cannot be determined at compile time, then it is impossible for the compiler to know if that value will be used.

Discussion

Consider the following C code:
int numargs (argc, argv) {
    int x, y;

    if (argc > 3) {
        y = argc;
    }
    else {
        y = argc + x;
    }
    return y
}

Since the compiler has no way to know how many arguments will be entered on the command line when this code is executed, it has no way of determining which branch will be taken during execution. If there are 4 or more arguments then the fact that x remains uninitialized is not a problem. If there are less than 4 arguments, it becomes a problem. Scott [2, p. 260] offers a brief discussion and gives some examples where this can be checked by the compiler. Some languages (Java for example) will signal an error if there is any way that a variable (only local variables for non-object types in Java) can be accessed if it has not been initialized. The above code would not compile if written in Java.

It should be noted that if the cost of dynamic checking can be tolerated, it is possible to determine if an uninitialized r-value will be used.

13. What is the difference between macros and routines? Explain the difference in terms of the concept of binding.

Solution

In C and C++ macros are bound by the pre-processor at compile time while routines are bound at runtime.

Discussion

A binding is an association between two things, such as a name and the thing it names. Binding time is the time at which a binding is created or, more generally, the time at which any implementation decision is made [2, p. 106].

With macros this means that they every instance of a macro call is replaced by the unevaluated right hand side of the macro definition with parameters substituted as necessary. This code is only evaluated when it is needed, a process called normal-order-evaluation. Routines on the other hand must be evaluated before they can be used. This is called applicative-order evaluation.

In C, any line which begins with #define is a macro definition, even if it only associates a name and a value. Macros are often found in real-time systems where their “evaluate as necessary” behavior can save execution time. They avoid the overhead of the subroutine call mechanism
including register saves and restores), and the code they generate can be included in any code optimizations that the compiler is able to make in the code associated with the call [2, p. 302]. Also note that inline functions are bound at compile time.

17. A routine is called history sensitive if it can produce different results when activated twice with the same values as parameters and visible nonlocal variables. Explain why a language with static memory allocation allows writing history-sensitive routines.

Solution

Variables which can be declared static are initialized only once and retain their value from one invocation to the next. In cases where a static local variable is modified within a routine, it will still have that value the next time the routine is called.

Discussion

Consider the following C code which demonstrates this property:

```c
void stat(); /* function prototype */

main() {
    int i;
    for (i = 0; i < 5; ++i) stat();
}

stat() {
    int auto_var = 0;
    static int static_var = 0;
    printf("auto = %d, static = %d \n", auto_var, static_var);
    ++auto_var;
    ++static_var;
}
```

The output from this program is:

```
auto = 0, static = 0
auto = 0, static = 1
auto = 0, static = 2
auto = 0, static = 3
auto = 0, static = 4
```

The program does not modify any variables outside the local scope of `stat()` and the value of `static_var` is different each time the function is called.
22. For the following C3 program fragment, describe each stage in the life of the run-time stack until routine beta is called (recursively), by alpha. In particular, show the dynamic and static links before each routine call.

```c
int i = 1, j = 2, k = 3;
beta();
alpha() {
    int i = 4, l = 5;
    ...
    i += k + l;
beta();
    ...
};
beta() {
    int k = 6;
    ...
    i = j + k;
alpaha();
    ...
};
main() {
    ...
    beta();
    ...
}
```

Solution

1. Global variables i, j, and k are allocated space on the stack and initialized with 1, 2, and 3 respectively.
2. main() is allocated space on the stack and a static link from main() to the global environment is created.
3. main() calls beta().
4. beta() is allocated space on the stack, a static link from beta() to the global environment is created, a dynamic link from beta() to main() is created, and local variable k is created and initialized with 6.
5. The sum of the values of the global variable j (2) and the local variable k (6) is assigned to global variable i, resulting in the new value of 8.
6. beta() calls alpha().
7. alpha() is allocated space on the stack, a static link is created from alpha() to the global environment, a dynamic link is created from alpha() to beta(), local variables i and l are created and initialized with 4 and 5 respectively.
8. The sum of the values of the global variable k (3), the local variable l (5), and the local variable i (4) is assigned to the local variable i, resulting in the new value of 12.
9. alpha() calls beta() recursively.

Discussion

Each stack frame contains a reference to the frame of the lexically surrounding subroutine which is called the static link [1, p. 88][2, p.120]. Analogously, the saved value of the frame pointer which will be restored on subroutine return is called the dynamic link and is a reference to the frame of the caller [1, p.79][2, p.429]. This example has a dynamic chain from alpha() to beta() to main(), but all of the static links reference the global environment. Static chains exist when routines are declared inside other routines [1, pp. 86-88] and describe the path that must be followed to access non-local variables.
Chapter 3 Exercises

Chapter overview.

Chapter 3 covers types. The topics which the authors stress are built-in types (3.1), extending built-in types (3.2), type systems (3.4 and 3.5), and implementation models (3.6).

Problems, solutions and discussions.

6. Consider the C++ class in Figure 3.4. What happens if the size of the array used to store the stack is exceeded by the push operation?

Solution

Figure 3.4 shows a template for a generic Stack class. Because the class does not contain either an assertion mechanism to prevent the stack from growing beyond the bounds of the array, or an exception handling mechanism to deal with that occurrence, the program can write data to memory that it shouldn’t. The end result of this may be a runtime segmentation fault or improper semantic behavior when stack data has overwritten other data.

Discussion

Exceeding array bounds is a very common and easily made error. C++ provides the programmer with two tools to deal with these kinds of errors. Assertions could be used to keep the program from exceeding the array bounds by comparing the length of the stack with the size of the array. If a call to push would exceed the array bounds, the test condition in the assert statement would return 0 (false) and call abort which would exit the program with an illegal instruction fault. If the programmer wished for program execution to continue in such a case, she could use the C++ error handling routines (throw, catch) to provide a mechanism for recovering from this error. Languages like Java and Pascal provide run-time array bounds checking to deal with this situation.
7. Add assertions to the Eiffel program in Figure 3.7. Discuss what happens when the length of the array is exceeded by a call to push, and what happens when pop is called for an empty stack.

Solution

class STACK[T] export
    push, pop, length
creation
    make_stack
feature
    store: ARRAY[T];
    length, size: INTEGER;

make_stack(n: INTEGER) is
    do store.make(1, n);
       -- this operation allocates an array with bounds 1, n
    length := 0
    size := n
end; -- make stack

push(x: T) is
    require
        length < size
    do length := length + 1;
        put(x, length)
        -- element x is stored at index length of the array
    ensure
        length <= size
end; -- push

pop: T is
    require
        length > 0
    do Result := store@(length);
       -- the result in the array whose index is length is copied in the
       -- language predefined object Result, which always contains the
       -- value returned by the function
        length := length - 1
    ensure
        length >= 0
end; -- pop
invariant
    length > 0 and length < size
end; -- class STACK
The preconditions keep the program from either pushing more elements onto the stack than it can hold or from popping more elements off than exist. If any attempt is made to do either of these, an exception is thrown.

**Discussion**

In this example, the push precondition \( \text{length} < n \) keeps the program from exceeding the size of the array by requiring at least one free array element for a push operation. The push postcondition \( \text{length} \leq n \) ensures that the array will not have exceeded its bounds when the routine exits. Similarly, the precondition for pop requires that the array have at least one item and the postcondition ensures that the stack size is not negative after the item has been removed.

The purpose of assertions is to guarantee correct program execution. Eiffel is one of a handful of languages (C++, Euclid, and Algol W are others) which allows the programmer to associate a class with an invariant property [1, p. 134]. In Eiffel the invariant property specifies a precondition which must exist before a routine can execute (**require**) and a postcondition that must exist when the routine terminates (**ensure**). If both the precondition and postcondition are true, the routine executes normally. If either condition is false, an exception is thrown. In C++, only the precondition is checked. The **assert** utility (defined in **assert.h**) tests the value of the expression. If the value is 0 (false) then **assert** prints an error message and calls function **abort** (defined in **stdlib.h**) to terminate program execution. More information on why assertions are useful can be found in [12].

**13. Is a static type system strong? Conversely, is a strong type system static?**

**Solution**

In theory, static type systems are strong, but not all strong type systems are static.

**Discussion**

The goal of a type system is to prevent the writing of type-unsafe programs as much as possible. A type system is said to be strong if it guarantees type safety, that is, if programs written by following the restrictions of the type system are guaranteed not to generate type errors [1, p. 138]. Scott [2, p.321] says that if a language is strongly typed and type checking can be performed at compile time, then the language is statically typed. Therefore, a static type system must be strong. (The authors define static typing as the process of binding variables to their type at compile time [1, p. 54]. Binding is obviously not the same thing as checking and by this definition, C (and other languages) are strongly typed, which is clearly not the case.)

Conversely, there exist some dynamically typed languages which are also strong (SML for example), so a strongly typed language does not have to be a statically typed language.
16. Define monomorphic and polymorphic type systems.

Solution

In a monomorphic type system each constant, variable, parameter, and function result has a unique type. In a polymorphic type system function operands can have more than one type.

Discussion

In practice, even languages like Pascal are not strictly monomorphic. Pascal forces the programmer to specify the exact type of each formal parameter and function result. But Pascal allows operator overloading which would not be allowed in a strictly monomorphic type system. In fact, many monomorphic languages have some polymorphic features, such as overloaded mathematical operators or Pascal’s nil value which can be assigned to pointers of any type.

Polymorphism comes in a number of flavors. Ad hoc polymorphism occurs when an operator works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways on each type. This can happen in several ways: overloading, coercion, subranges, and value sharing. Overloading occurs when the same name is used to refer to more than one function and the proper function is determined through context. Coercion occurs when an operand is converted automatically to an entity of the right type either at compile-time or run-time. The proper conversion is determined by context. Subranges occur when a language (like Pascal) allow a new ordinal type to be defined as part of the range of another ordinal type. Value sharing occurs when one value is shared by a number of distinct types. The nil pointer in Pascal is an example, where any pointer variable can be assigned the value nil.

Universally polymorphic functions generally work on a number of types, all having a common structure. There are two types of universal polymorphism; parametric and inclusion. Parametric polymorphism is most common in functional programming languages. The same code is executed for arguments of any admissible type. Inclusion polymorphism occurs in languages that allow subtypes and inheritance. An instance of a subtype can be manipulated by the same functions that operate on instances of the supertype.

23. In C++, each class has an associated default assignment operator that does a memberwise copy of the source to the target. But when a new object of class T is declared this way,

\[ T \; x = y; \]

the copy constructor of x is used to construct x out of y. Why can the assignment operation not be used? What is the difference between assignment and copy construction?
Solution

The assignment operator cannot be used if, as in this example, the target object does not exist. Copy construction creates a new object identical to the original while assignment (by memberwise copying) assigns all the attributes of the source object to the target object.

Discussion

This behavior is not unexpected. In order to copy the r-value of a variable to the r-variable of another variable (assignment) the destination must already exist. The default assignment operator for a class performs a memberwise copy (based on the assignment for struct in ANSI-C) from object to another object of the same class.

33. The simplest possible reaction of the run-time system to a statement like dispose (in Pascal) is to ignore it. That is, storage is not freed for later reuse. How would you design an experiment to check what a language implementation actually does? Perform an experiment on an available implementation of Pascal.

Solution

Depending on the specific implementation of Pascal available, the following code should work:

```pascal
program test_dispose;
var a, b : int_pointer;
begin
  a := new(a);
  b := a;
  dispose(a);
  repeat
    a := new(a);
  until a = b;
  writeln ("Memory is being reused")
end.
```

Discussion

The method used here is to allocate two pieces of memory and see if they are the same. This code does this by first creating two integer pointers which both point to the same entity. One pointer is then deallocated with dispose. A loop is then entered in which a new integer entity is created and the two pointers are compared to each other. If the memory is not being reclaimed, the two pointers will not point to the same memory address, and the loop will repeat. If the memory is being reclaimed, the pointers will be equal and the loop will terminate with a message to the user.
Chapter 4 Exercises

Chapter overview.

Chapter 4 covers how computation is structured. The topics which the authors stress are expressions (4.1), statement-level control structures (4.2), routine call and return (4.3), exception handling (4.4), pattern matching (4.5), nondeterminism and backtracking (4.6), event-driven control structures (4.7), and control structures for concurrent programming (4.8).

Problems, solutions and discussions.

1. Study the case statement of Pascal and compare it to the C++ switch statement and the Ada case statement.

Solution

The Pascal case statement is of the form:

```pascal
  case expression of
      constant_list_1 : statement_1;
      constant_list_n : statement_n;
  end
```

Features:
- *Expression* is any ordinal type, that is it can be any simple type except real.
- Values can be either single constants or constant lists separated by commas.
- Statements can be single or compound, but multiple statements are not allowed.
- Only one segment can be executed.
- An unrepresented control expression value results in a run-time error (1984 ISO Standard), although many dialects have an otherwise clause.

The C++ switch statement is typically of the form:

```cpp
  switch (expression) {
      case constant_list_1 : statement_1;
      case constant_list_n : statement_n;
      [default: statement_n+1]
  }
```

Features:
- *Expression* must be an integer or integer compatible type (e.g., char).
- Statements can be anything including multiple statements.
- Multiple segments are executed unless a break statement used. This is how multiple values are associated with the same statement.
- Default clause allows an action for unrepresented values, but if it’s not present execution will fall through the switch without executing any of the statements.
The Ada case statement is of the form:

```plaintext
case expression is
  when constant_list_1 => statement_1;
  when constant_list_n => statement_n;
  when others => statement_n+1;
end case;
```

Features:
- Constants can include subranges (10..15) and boolean OR operators (1..5 \(\mid\) 15..20).
- Control expression must be exhaustive.
- Other clause allows an action for unrepresented values.
- Only one segment can be executed.

Discussion

The case statements in Pascal and Ada work in very similar ways although Ada allows a richer set of values to be constructed for comparison than Pascal does. In each, only a single segment of the statement can be executed unlike the C++ `switch` where, unless otherwise indicated (with a `break`), control continues at the next segment of the statement. It is often the case that some test conditions, including `default` or `other`, will execute an empty statement. Ada provides the `null` statement for this. In C++ the programmer can simply `break` at this point. In Pascal a semicolon is used to denote an empty statement.

These constructs can make code more readable than multiple `if-then-else` statements and in certain cases (in C for example) may even speed execution if the structure is implemented as a jump table or other structure (a tree of comparisons for example).

3. It can be shown that, in principle, any program can be written using just these control structures:
   - grouping statements into a block
   - `if...then...else...`
   - `while...do...`

Show how other control structures (such as the case statement or a repeat loop) can be represented using the above minimal set. Discuss why in practice a richer set of control structures is provided by a programming language.

Solution

Consider the following `switch` statement from a larger piece of C++ code:
switch (input) {
    case 1: x = 1;
    break;
    case 2: y = 1;
    break;
    default: z = 1;
}

This can be handled in several ways. One example is:

if (input == 1)
    x = 1;
else if (input == 2)
    y = 1;
else z = 1;

Consider also the following trivial repeat loop in MicroWorlds Logo:

global [x]
setx 0
repeat 100 [setx x + 1]

this produces the same result (x = 100) as

global [x]
setx 0
loop [setx x + 1 if x = 100 stop]

Languages have additional control structures because it makes it easier to read and write programs. In some cases they reduce the size of source code files by eliminating the need for duplicate code. They allow the programmer to choose a structure that more precisely expresses the problem that she is solving. This makes the form of the code a more accurate reflection of the logic involved in the solution and reduces the possibility of logical errors. Different structures can also give cues to the compiler that may allow it to do a better job of optimization.

Discussion

An example of how having a rich set of control structures reduces the size of source code files can be illustrated by looking at how a do-while loop in C would have to be written if the language only provided a while loop. The do-while guarantees that the loop code will be executed at least once. In order to have that same guarantee with the while loop, the block of code would need to be duplicated as in this example:

do {
    <my code>
} while (my condition);
Would need to be

```c
{ <my code> } 
while (condition) {
    <my code>
}
```

A variety of structures makes it easier for a programmer to code a problem in the same way she thinks about the problem. For example, an if-then-else statement might be the way the programmer understands the problem and so would be a logical choice for the code. If none were available (for example only an if-then or a while loop (see note below)), the same results might be achieved, but the difficulty in assuring their correctness would be greatly increased.

And while it’s true that most of the benefits obtained from having a rich set of control structures go directly to programmers/testers/maintainers, execution time and space requirements for a program can also benefit from the information that a compiler might have available about different structures.

It is also interesting to note that, strictly speaking, the if-then-else construct is not needed since that functionality can be done with a while loop. For this we need to remember that in C the value of false returned by booleans is 0 and the value of true returned by booleans is 1. Consider the following example in C:

```c
if (i == 0)
    i = 10;
else
    i = 1 + 1;
```

This can be done equivalently with two while loops:

```c
bool flag, guard;
flag = (i == 0);
guard = true;
while (flag && guard) { /* if (i == 0) */
    i = 10;
    guard = false; /* terminate the loop */
}

guard = true;
while (!flag && guard) { /* else */
    i = i + 1;
    guard = false;
}
```
16. In the producer-consumer example implemented with semaphores in Section 4.8.3.1, suppose that \( V(\text{mutex}) \) is written incorrectly as \( P(\text{mutex}) \) in process producer. How does the system behave?

(The example is incorrectly given. The question really concerns Figure 4.10 in section 4.8.2.1.)

Solution

When \( P(\text{mutex}) \) is called on entry into the producer process, the value of \( \text{mutex} \) becomes 0. Calling \( P(\text{mutex}) \) a second time suspends the producer and leaves \( \text{mutex} = 0 \). When consumer is called it too is suspended and the system is deadlocked.

Discussion

The incorrect call produces a deadlock situation very quickly. This question points out the need to be absolutely certain that if only one process can access a resource at any given time that the process releases the resource when it is finished with it. Amazingly enough, the authors never use the term deadlock nor do they discuss its consequences to any degree.

17. When semaphores are used to implement mutual exclusion, it is possible to associate a semaphore \( \text{SR} \) with each resource \( R \). Each access to \( R \) can then be written as

\[
\begin{align*}
P(\text{SR}); \\
\text{access } R; \\
V(\text{SR})
\end{align*}
\]

What should the initial value of \( \text{SR} \) be?

Solution

Using the definitions of \( P \) and \( V \) given in the text [1, p. 220], the initial value of \( \text{SR} \) should be 1.

Discussion

The purpose behind mutual exclusion is to ensure that only one process has access to a resource at any given time. If the initial value of \( \text{SR} \) is greater than 1 then multiple processes would have access to \( R \) simultaneously.
Chapter 5 Exercises

Chapter overview.

Chapter 5 covers program organization. The topics which the authors stress are software design methods (5.1), encapsulation, interface, separate compilation, and module libraries (5.2), case studies (5.3), and language support for generic units (5.4).

Problems, solutions and discussions.

1. Discuss the effect of global variables on the writability and readability of large programs.

Solution

They decrease readability by making it necessary to focus on multiple scopes when trying to understand the code in a block. Their use also makes it difficult to accurately determine what the value of the variable is at any given place in the code since globals can be modified anywhere. Their use makes it difficult to track both the value and effect of global variables as programs become larger. This becomes especially problematic in large programs. Conversely, their use can make writing code easier. Globals reduce the number of arguments that must be passed to (and from) functions and are one way of getting around things like the limitation of a single return value in languages like C. In some languages (for example C), globals are allocated statically which makes it easier to determine the amount of memory that will be needed. And there is some data which truly is global in scope and needs to handled in this way.

Discussion

In general, the use of global variables is considered poor programming style. This is due in large part to the readability and hidden dependency problems mentioned above. It can also lead to sloppy programming styles as well as poorly structured code. But if used carefully and in appropriate ways, they can have a number of benefits. In real-time systems, their use can reduce the overhead in function calls, making the system faster. Their use for truly global data (for example singular resources such as a device driver, the event queue, or system parameter) should not be discounted. The main points that students should mention are:

Problems

- It is hard to track their values.
- It is hard to track their usage.
- They can cause dependencies between pieces of code in different blocks which may have no obvious dependencies and may be separated by a considerable distance, even being in different files.

Benefits:

- They reduce the amount of information that needs to be passed around.
- They can provide a mechanism for modifying multiple values external to a function.
- They can be more efficient in many cases.
They are useful for storing truly global information.

11. This exercise is about achieving visibility in C units. If a variable is declared in a function, which units have access to it? If a variable is declared outside of functions, which units have access to it? If a variable is declared as extern, where is it defined? If a variable is defined as static, which units have access to it?

Solution

Variables declared in functions are only available with the function. Moreover, within that function, access is limited to the block in which it is declared.

C defines static scoping of variables outside of functions in terms of the source file that contains the declaration. Only the units in the source file have access to a variable defined outside of any function. Additionally, only units defined after the variable declaration can see the variable.

A variable declared as extern is defined in some other unit. Its scope is the file in which it is declared as well as any units in the program that reference it as extern.

If a variable is defined as static within a function, only that function has access to it. If it is defined outside of any function, then it is known only within the remainder of the source file in which it is declared, but not in any other file.

Discussion

In C the unit of encapsulation is the file, and variable visibility is dependent on this. Variables declared at the head of a file are visible to functions in the file and can also be visible to functions in other files if those files declare them to be extern. It can be a good idea to declare global variables in a .c file which is included at compile time. This has the benefit of putting all globals into a single file, ensuring that two globals do not have the same name and reducing the possibility of failing to include a needed file. At compile time, variables declared extern tell the compiler that space for the variable will be allocated when the file containing its declaration is encountered.

The authors’ explanation of static is somewhat simplistic. static variables may be either internal or external (the situation mentioned by the authors). Internal static variables act just like normal (automatic) variables except that they are only initialized once and retain their last value from one call to the next (see also chapter 2, question 17). An external static variable is known within the remainder of the source file in which it is declared, but not in any other file [5, p.80].
18. Without generics, if we have $m$ data types, $n$ algorithms, and $p$ data structures, we need to write on the order of $m \times n \times p$ library components to support all possible algorithms on all possible data structures for all possible data types. For example, we need a sort routine for arrays of integers and a sort routine for lists of reals. Explain how the use of generics reduces the number of library components that need to be written. Assuming that we could write all the $m \times n \times p$ components without using generics, what other problems do we face in terms of documentation, use, and maintenance of these routines?

**Solution**

Generics allow a single data structure description and associated algorithms to apply to a range of types. In the best case, a single data structure can be developed to contain any type, for example, a linked list data structure which could be used with any type. Algorithms could then be implemented for use on that generic data structure. In this way, generics can dramatically reduce the number of data structures and algorithms that need to be created.

Without generics a number of issues become important with respect to documentation, use and maintenance of routines.

1. More code has to be written increasing the possibility of introducing bugs and making it harder to find and fix them.
2. Changes and fixes must be made in multiple places in the code, often in multiple files. More effort and time are required to locate all occurrences of code for these tasks and to ensure that all changes are correct.
3. There is more code and documentation to write which makes it harder to keep both of high quality.
4. By increasing the quantity of source code for a project, the percentage of the code that can be understood by any one person is reduced.
5. It becomes harder to ensure that interfaces into similar sections of the code are consistent, making the source harder to understand.
6. Libraries can swell in size further increasing the difficulty of constructing a high quality system.

The availability of generics addresses all of these issues and can lead to a smaller, more consistent, more provably correct, and higher quality system.

**Discussion**

There are other advantages to using generics. By not having to worry about code duplication, the designer can better use her time creating robust and efficient code. Debugging becomes more productive since a bug fixed in a generic will propagate to every instance of the code, not just the instances for one type. Having larger parts of the system comprehensible to an individual can reveal relationships among components that can be of great benefit during debugging and maintenance.
19. Ada defines two kinds of types: private type and limited private type. (Check these features in the language manual.) What is the difference between these two? Is there a similar concept in C++? If not, why not? Does their absence imply a lack of functionality in C++?

Solution

Private types are similar in functionality to classes in C++. They have a public interface and a private section to hide implementation details. Limited private types completely restrict primitive operations like assignment and boolean comparison to those specified in the visible part of the package. There is no equivalent in C++ as there is no convenient way to override the default operations available to classes, but this does not imply a lack of functionality in C++.

Discussion

The Ada package is roughly equivalent to the C++ or Java class. Packages were created to group data and operations on those data as well as for information hiding. A package consists of two parts: a visible interface to the package and a hidden implementation section. There are several ways a package can be implemented, but here we’re interested in private types. Packages support the creation of types whose construction details are hidden from the user and in this use is very similar to classes in Java and C++. The declaration is of the form (Ada has no predefined constructor or destructor forms)

```ada
package Special_Numbers is
    type Special is private;
    |: constant Special;
    function foo (X: Special) return Special;
    function bar (X: Special) return Special;
    ...

private
    type Special is
        record
            ...
        end record;
    |: constant Special;
end;
```

This provides the functions `foo` and `bar` in the package interface, but fully hides the implementation of special numbers [11, pp 208-209]. Objects of this type can be given initial values at instantiation.

Limited private types differ structurally by having as the second line of the declaration

```ada
type Special is limited private;
```
In these packages assignment (see chapter 3, question 23) is not available between objects and the boolean operators = and /= cannot be used to compare to limited private objects. The instantiation of a limited private object cannot include an initial value. Limited private types give the package writer total control over these objects [11, p. 223], but also require her to specify operations like assignment and comparison is needed.

There is no equivalent to limited private types in C++. While the programmer could create her own class specific assignment or comparison methods, their existence would not block the built in operations for classes. So while the programmer could specify how these operations would be done for a class (as in limited private types) she would be unable to enforce that only those methods be used. In a broad sense, the lack of any equivalent to limited private types does imply a lack of functionality, but in practice it probably makes very little difference.

20. What is the difference between overloaded functions and generic functions in C++?

Solution

Overloaded functions are a form of ad hoc polymorphism where different functions with the same name are created for every type on which the function will operate. Generic functions are a form of polymetric polymorphism where one function operates on multiple types which match the signature required by the function.

Discussion

This question points out the important differences between ad hoc and universal polymorphism. Ad hoc (in the form of overloading) merely gives the impression of polymorphism while requiring a separate code block for each version of the overloaded function. This can make writing, debugging and maintaining code more time consuming and difficult. For example, bugs found in one function version may or may not be duplicated in others. Additionally, it may be that all function versions are not grouped in the code making it hard to know where to look for them. This can impact both the readability (and therefore understandability) and the maintainability of the code.

Universal polymorphism (in the form of generics) overcomes these problems by allowing a single function to operate on a variety of types whose signatures are the same. This may require some additional up-front planning by the developer, but can pay off in the long term.
Chapter 6 Exercises

Chapter overview.

Chapter 6 covers object-oriented languages. The topics which the authors stress are the basic concepts of object-oriented programming (6.1), the relationship between inheritance and the type system of a language (6.2), and how object-orientation is supported in C++, Eiffel, Ada 95, Smalltalk, and Java (6.3).

5. In Section 6.1.4, we suggested that a language that does not support dynamic binding may use case statements or function pointers to achieve the same result. Explain how this can be done and discuss the drawbacks of such solutions.

Solution

A language without dynamic binding can use function pointers to build a table of member functions for each class. A call to a member function for a class is then implemented by looking up the correct function in the table, and executing that function. The drawback of this approach is the need to do some kind of lookup into the table. The lookup can be done using a fixed offset, which is relatively fast but requires that a member function’s name be stored in the same offset for each class to which that member functions can be applied. Optionally, a lookup by name can be used, but this is much slower.

Another way to do this is through the use of case statements. In this method there is a single entry point for each member function. Within each member function a case statement is used to choose which code should be executed based on the type of the object that call is being applied to. Assuming an efficient implementation of the case statement, this method may be faster than the table method but might require more code to implement. When using this approach, adding a new class requires that every case statement be regenerated and recompiled for each member function implemented in the new class.

Discussion

Using a lookup table for function pointers is, in essence, implementing a form of late binding for method calls. This approach is similar in many regards to how languages which allow dynamic binding work. One common way these languages do this is by having a record who’s first field contains the address of a virtual method table (vtable) for the object’s class. The vtable is an array whose $i$th entry indicates the address of the code for the object’s $i$th virtual method. All objects of a given class share a common vtable.

The key to making the case statement approach work is to have an efficient means of implementation. Jump tables can be fast, but they can also be quite large which will make the implementation larger than it would be if using the table approach.
6. Let us define a relation \( s \leq t \) to mean that \( s \) is a subtype of \( t \). We want to define a subtype relation for function signatures. First we represent a signature as \( f : t_1 \rightarrow t_2 \), indicating a function \( f \) that takes an argument of type \( t_1 \) and yields a result of type \( t_2 \). How would you state the covariance and contravariance requirements using the subtype relation? That is, complete the equivalence relation below by replacing the question marks with appropriated relations: Given \((f_1 : s_1 \rightarrow t_1) \) and \((f_2 : s_2 \rightarrow t_2)\), \( f_1 \leq f_2 \) iff \((s_1 ? t_1) ^ (s_2 ? t_2) \). How would you describe the Eiffel rule on redefinitions?

Solution

(The problem contains a typographical error. The subtyping relationships are between \( s_1 \) and \( s_2 \) and between \( t_1 \) and \( t_2 \). See the discussion section for details.)

Given \((f_1 : s_1 \rightarrow t_1) \) and \((f_2 : s_2 \rightarrow t_2)\), \( f_1 \leq f_2 \) iff \((s_2 \leq s_1) ^ (t_1 \leq t_2) \). Eiffel’s’ rule for redefinition requires that the redefining function have exactly the same signature as the function that is redefined. For this reason the parameters of the redefining routine must be assignment compatible with the parameters of the redefined routine which in the above statement would mean \( f_1 \leq f_2 \) iff \((s_1 \leq s_2) ^ (t_1 \leq t_2) \).

Discussion

This question deals with the weak form of the principle of substitutability which says that \( S \) is a subtype of \( T \) if substituting an object of type \( S \) wherever an object of type \( T \) is expected does not introduce the possibility of type errors occurring at run time [3, p. 208]. This principle can be guaranteed by the following conditions:
1. All services in the superclass are present in the subclass.
2. Additional services and attributes may be present in the subclass
3. If a service is redefined in the subclass, then the new service must be compatible with the original service in the superclass.

This further requires a definition of compatible [3, p.209]. Two services are compatible if:
1. They have the same number of parameters.
2. The type of each input parameter in the redefined service is a supertype of (or the same type as) the corresponding parameter in the original service.
3. The type of each output parameter in a redefined service, including the returned value (if any), is a subtype of (or the same type as) the corresponding parameter in the original service.

The second condition is the contravariance rule (the subtyping relationship for the input parameters is the opposite of that for the functions) and the third condition is called the covariance rule (the subtyping relationship for the result parameters is the same as that for the functions). Hence contra- indicating opposite and co- indicating sameness.
This pattern is not present in Eiffel. Instead, Eiffel requires that subtyping relationships for both the input and result parameters mimic the subtyping relationship of the functions. This means that the covariance rule holds for both input and result parameters [1, p. 312].

The typographic error in the problem is in asking the student to compare the input and return types within the functions rather than across them, so \((s_1 \ ? \ t_1) ^\land (s_2 \ ? \ t_2)\) needs to be \((s_1 \ ? \ s_2)^\land (t_1 \ ? \ t_2)\).

9. Sometimes inheritance is used improperly. For example, consider defining an automobile class. We have and existing class window. Since automobiles have windows, we have two options; we can derive automobile from window, ensuring that the automobile will have a window, or we can define the class automobile and use window as a member of the class. Why is the second solution better? Explain the proper use of inheritance in terms of the is-a relation.

Solution

The solution to this problem becomes more clear if we extend the example a little. Consider a class hierarchy with two top level classes (i.e., classes with no parents), vehicle and opening. Using the is-a relation, let train be a child class of vehicle along with boat and airplane. Again using the is-a relation, let window be a child class of opening along with trapdoor and fire escape. Now if we derive an automobile class we find it has more in common with the children of vehicle (carries passengers, travels distances, has velocity, has windows, etc) than it does with the children of opening. The first solution leads to inherited behavior that may not make sense which in turn may lead to a complex and confusing class hierarchy. The second solution leads to an inheritance hierarchy that will make sense.

The proper use of the is-a relationship is to ask two questions about the classes in question. In this example we ask, “is a car a kind of window?” and “does a car have windows?”. Choosing the question for which the answer is “yes” tells us what the inheritance should be.

Discussion

This problem addresses a classic error made by beginners when confronted with inheritance. The authors briefly discuss the differences between subtypes and subclasses with respect to the is-a relation. They state that the subtype relationship is generalizable to user-defined types such as classes, saying that if the is-a relationship holds between a subclass S and a class C (S is-a C), then the type defined by the subclass is a subtype of the type defined by the parent class. They further contend that the is-a relationship holds if a derived class only adds member variables and functions or redefines existing functions in a compatible (see discussion for question 6 above) way [1, p. 294].
The flip side of the is-a relation is the has-a relation and determining the structure of a hierarchy in a language with multiple inheritance can become very messy when the two kinds of relationships become intermixed. Java attempts to deal with this through the use of single inheritance with interfaces and succeeds to a large extent (see question 16 below). Snyder [12] does an excellent exposition on inheritance.

16. Java supports both interfaces and abstract classes. Both may be used to define specifications. Compare these two concepts as supported in Java.

Solution

Any class in Java which has one or more abstract methods is an abstract class. Abstract classes may also contain regular (non-abstract) methods and data members. Interfaces are purely abstract entities in which all methods are abstract. Neither abstract classes nor interfaces can be instantiated. When a class extends an abstract class, all of the data members and regular methods are acquired by the child class and the subclass is responsible for implementing versions of all the abstract methods. While an abstract class can implement some regular methods, an interface cannot provide any implementation and it is the implementing class’ responsibility to implement all methods defined by the interface. Only one abstract class can be extended in a new class, but multiple interfaces can be implemented.

Discussion

The appropriate use of both abstract classes and interfaces relies heavily on the is-a relationship. When a strong version of the is-a relation holds, then extension of an abstract class is suggested (for example, an abstract polygon class with data members for perimeter and area and abstract perimeter and area calculation methods). When a weaker version of the is-a relation holds for one aspect of the class, then implementing an interface makes sense. For example, implementing the Runnable interface says that a class is a kind of Runnable, but the class is usually much more than that.

The difference between abstract classes and interfaces can also be seen as one aspect of inheritance; is the inheritance for code reuse (abstract classes) or for subtyping (interfaces).
18. In the buffer example of Figure 6.14, both the put and get methods have a statement of the form while (condition) wait(). Explain why we should not replace the “while” loop with an “if” statement.

Solution

It cannot be known when calls to put and get will be made. Using an if statement cannot guarantee that the necessary condition is true when returning from wait().

Discussion

The issue here is that the yield (wait) is relying on a synchronization condition which is not specified by get and put meaning that it has to come from somewhere else. The while loop is, in effect, a spin lock to control this synchronization. To achieve the same functionality, the code could be rewritten using semaphores as in Figure 4.10.

As an aside, the example in Figure 6.14 is Java-like code and is both semantically and syntactically incorrect for Java.
Chapter 7 Exercises

Chapter overview.

Chapter 7 covers functional programming languages. The topics which the authors stress are the characteristics of imperative languages (7.1), principles of functional programming (7.3), and representative functional languages (7.4).

Problems, solutions and discussions.

1. Reduce the lambda expression \([y/x]((\lambda y.x)(\lambda x.x)x)\).

Solution

1. \([y/x]((\lambda y.x)(\lambda x.x)x) =
2. \([y/x](((\lambda y.x)(\lambda x.x))x) =
3. (((y/x)(\lambda y.x) [y/x](\lambda x.x)) [y/x]x)
4. (((y/x)((\lambda z.x) [y/x](\lambda x.x)) [y/x]x)
5. ((\lambda z.y \lambda x.x)(y) =
6. (y y)

Discussion

The goal of this problem is to reduce the lambda expression by substituting \(y\) for \(x\) \(([y/x])\) in all occurrences where \(x\) is not bound. Using the parenthesizing rules [1, p. 339] we let \(e1 = (\lambda y.x)\), \(e2 = (\lambda x.x)\), and \(e3 = x\) which gives us (2). Distributing the substitution across the expressions gives us (3).

We then have to substitute starting in the inner nesting and proceeding outward. \(x\) is free (not bound) in \((\lambda y.x)\) so substituting yields \((\lambda y.y)\). But the bound \(y\) is not the same \(y\) as the one we’ve substituted (which would cause the substituted \(y\) to become newly bound (rule s2 [1, p. 340]) so we can use \(\alpha\)-conversion (rule r1 [1, p. 341]) to rename \(y\) in \((\lambda y.x)\) to \(z\). This gives us (4). We can now substitute \([y/x]\) which gives us (5). Note that \(\lambda x.x\) cannot be substituted in because \(x\) is bound. The substitution is now complete so we can evaluate. The expression \(\lambda z.y\) takes a \(z\) and returns \(y\). In this case \(z\) is the value returned by evaluating \(\lambda x.x\) which is \(x\). So \((\lambda z.y \lambda x.x)\) evaluates to \(y\) which gives us (6).
2. Reduce the lambda expression \((\lambda x.(x x)) \ (\lambda x.(x x))\). What is peculiar about this expression?

Solution

\[(\lambda x.(x x)) \ (\lambda x.(x x)) = (\lambda x.(x x)) \ (Q) = (Q \ Q) = (\lambda x.(x x)) \ (\lambda x.(x x))\]

This expression reduces to itself.

Discussion

This problem is an example of the y combinator which is one of the fundamental results of recursive procedure theory. Notice that if one were to attempt to apply this expression, the result would be an infinite recurse.

The important thing here is to understand what the leftmost \(\lambda x.(x x)\) expression does. It takes whatever it’s applied to (in this case the rightmost \(\lambda x.(x x)\)) and returns 2 copies of that expression. For illustration, lets replace the rightmost lambda expression with \(Q\) so that what we have is \((\lambda x.(x x)) \ (Q)\). We can now see that reducing this expression takes \(Q\) as an input parameter and returns 2 copies of it. So the result is \((Q \ Q)\). If we now substitute \((\lambda x.(x x))\) back in for \(Q\) the result of the reduction is \(((\lambda x.(x x)) \ (\lambda x.(x x)))\) which, minus theoutermost parentheses is the original expression.

4. Write a LISP program to compute the first 100 prime numbers.

Solution

```lisp
(defun primes (n)
  "find the first n prime numbers"
  (let ((primes-found 0))
    (do ((i 2 (1+ i)))
        ((= primes-found n))
      (block inner
        (do ((j 2 (1+ j)))
            ((>= j i) (print i) (setq primes-found (1+ primes-found)))
          (if (zerop (mod i j))
              (return-from inner))))))

To generate the first 100 primes, call (primes 100).
```
Discussion

This example is an iterative approach to the problem. One could also approach this in a tail-recursive manner as shown in the following code:

```lisp
(defun primes (n)
  (primes-helper n 1))
(defun primes-helper (n i)
  (if (= n 0)
      ()
    (if (is-prime i)
        (progn
          (print i)
          (primes-helper (- n 1) (+ i 1)))
      (primes-helper n (+ i 1))))

(defun is-prime (n) (is-prime-helper n 2))
(defun is-prime-helper (n i)
  (if (>= i n)
      T
    (if (zerop (mod n i))
        ()
      (is-prime-helper n (1+ i)))))
```

Note that both provided approaches are brute force and a more elegant solution might be to obtain the answer through an application of the Sieve of Eratosthenes.

This problem is designed to show a student’s ability to problem solve in a functional language where iteration and recursion are common. The reason that the solution code is not recursive is to present what may be the more common approach taken by students who come from procedural programming backgrounds.

6. What is the type of this ML function?

```ml
fun f(x, y) = if hd(x) = hd(y)
  then f(tl(x), tl(y))
  else false;
```

Solution

fn: list*list -> bool

Discussion

f is a function that takes two lists and compares them element by element for equality. It does this by comparing the first element in each list (the head). If they are the same, it calls itself
recursively with the remainder of the lists (the tail). It keeps doing this until either the elements being compared are not equal (in which case it exits and returns false) or until there is nothing left in the lists to compare in which case it returns true. ML would infer types for this function by first assigning it the signature \( \text{fn}: \text{list*list} \rightarrow \text{any} \) since it still cannot determine the actual return type. Examination of the third line supplies this information since false must be a boolean.
References

This list is composed of three types of references. The first are materials that I used directly to prepare the material in the solution set. These are cited in the solutions or discussions when this occurs. The second are materials that I found useful for reinforcing my knowledge of topics in PL or for providing comparisons of languages with which I was not familiar. The third are materials that helped me (and, I believe, can help a reader) gain a deeper understanding of a particular topic or concept. In almost all cases items listed here fall into two or even all three categories and I have made no attempt to group them. Their positions in the list roughly correlate to the order in which I made use of them, not to their value or quality.