

Subroutines and Control Abstraction

8.9 Exercises

- 8.35 Suppose you wish to minimize the size of closures in a language implementation that uses a display to access nonlocal objects. Assuming a language like Pascal or Ada, in which subroutines have limited extent, explain how an appropriate display for a formal subroutine can be calculated when that routine is finally called, starting with only (1) the value of the frame pointer, saved in the closure at the time that the closure was created, (2) the subroutine return addresses found in the stack at the time the formal subroutine is finally called, and (3) static tables created by the compiler. How costly is your scheme?
- 8.36 Elaborate on the reasons why parameters on the MIPS may need to have locations in the stack. Consider all the cases in which it may not suffice to keep a parameter in a register throughout its lifetime.
- 8.37 Most versions of the C library include a function, `alloca`, that dynamically allocates space within the current stack frame.³ It has two advantages over the usual `malloc`, which allocates space in the stack: it's usually very fast, and the space it allocates is reclaimed automatically when the current subroutine returns. Assuming the programmer *wants* deallocation to happen then, it's convenient to be able to skip the explicit `free` operations. How might you implement `alloca` in conjunction with the MIPSpro calling conventions described in Section ©8.2.2?
- 8.38 Explain how to extend the conventions of Figure ©8.11 and Section ©8.2.2 to accommodate arrays whose bounds are not known until elaboration time (as discussed in Section 7.4.2). What ramifications does this have for the use of separate stack and frame pointers?

³ Unfortunately, `alloca` is not POSIX compliant, and implementations vary greatly in their semantics and even in details of the interface. Portable programs are wise to avoid this routine.

- 8.39** In both the MIPSpro and gpc case studies, arguments were placed into the argument build area in “reverse” order, with the first argument at the top. Explain why this is important. (Hint: Consider subroutines with variable numbers of arguments, as discussed in Section 8.3.3.)
- 8.40** How would you implement nested subroutines as parameters on a machine that doesn’t let you execute code in the stack? Can you pass a simple code address, or do you require that closures be interpreted at run time?
- 8.41** Explain how you might implement `setjmp` and `longjmp` on a SPARC.
- 8.42** Continuing Example ©8.71, the call

```
csNames.consider(null);
```

will generate a run-time exception, because `String.compareTo` is not designed to take `null` arguments.

- (a) Modify Figure ©8.14 to guard against this possibility by including a predicate `public Boolean valid(T a);` in the `Chooser<T>` interface, and by modifying `consider` to make an appropriate call to this predicate. Modify class `CaseSensitive` accordingly.
 - (b) Suggest how to make similar modifications to the C# `Arbiter` of Figure ©8.16 and Example ©8.78. How should you handle lower bounds when you need both `Better` and `Valid`?
- 8.43** (a) Modify your solution to Exercise 8.22 so that the comparison routine is an explicit generic parameter, reminiscent of the `chooser` of Figure ©8.13.
- (b) Give an alternative solution in which the comparison routine is an extra parameter to `sort`.
- 8.44** Consider the C++ program shown in Figure ©8.18. Explain why the final call to `first_n` generates a compile-time error, but the call to `last_n` does not. (Note that `first_n` is generic but `last_n` is not.) Show how to modify the final call to `first_n` so that the compiler will accept it.
- 8.45** Following the code in Figure 6.5, and assuming a single-stack implementation of iterators, trace the contents of the stack during the execution of a `for` loop that iterates over all nodes of a complete, three-level (six-node) binary tree.
- 8.46** Build a preorder iterator for binary trees in Java, C#, or Python. Do not use a true iterator or an explicit stack of tree nodes. Rather, create nested iterator objects on demand, linking them together as a C# compiler might if it were building the iterator object equivalent of a true preorder iterator.
- 8.47** One source of inaccuracy in the traffic simulation of Section ©8.6.4 has to do with the timing at traffic signals. If a signal is currently green in the EW direction, but the queue of waiting cars is empty, the `signal` coroutine will go to sleep until `current_time + EW_duration`. If a car arrives before the coroutine wakes up again, it will needlessly wait. Discuss how you might remedy this problem.

```

#include <iostream>
#include <list>
using std::cout;
using std::list;

template<class T> void first_n(list<T> p, int n) {
    for (typename list<T>::iterator li = p.begin(); li != p.end(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

void last_n(list<int> p, int n) {
    for (list<int>::reverse_iterator li = p.rbegin(); li != p.rend(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

class int_list_box {
    list<int> content;
public:
    int_list_box(list<int> l) { content = l; }
    operator list<int>() { return content; }
    // user-supplied operator for coercion/conversion
};

int main() {
    int i = 5;
    list<int> l;

    for (int i = 0; i < 10; i++) l.push_back(i);
    int_list_box b(l);

    first_n(l, i);           // works
    last_n(b, i);           // works (coerces b)
    first_n(b, i);          // static semantic error
}

```

Figure 8.18 Coercion and generics in C++. The compiler refuses to accept the final call to `first_n`.

