12

Imperative Programming

Exercises

12.1 Compile-time complexity is a minor issue. In the case insensitive case, the Lexer can map all upper case letters into lower case. The rest of the compiler then remains ignorant of the issue.

Having case insensitive identifiers probably causes more compile-time errors for misspelled identifiers, since the programmer has to remember both the exact spelling and the use of upper vs. lower case letters. This is at odds with the practice in natural language.

Reliability can probably be argued both ways. It is nice to use case to vary the spelling of a reserved word so that it can be used as an identifier. On the other hand, without a clear and simple policy, case sensitive languages probably invite more compile-time errors.

12.2 For C the table is given in the text (Table 2.4). Here it is for Ada.

Operation (decreasing precedence)	**Associa-tivity**	**Ada**
Exponentiation	non	`**`
Logical not	right	`not`
Absolute value		`abs`
Multiply	left	`*`
Divide	left	`/`
Modulo	left	`mod, rem`
Unary minus	right	`-`
Unary plus	right	`+`

Operation (decreasing precedence)	**Associa-tivity**	**Ada**
Add	left	`+`
Subtract	left	`-`
Less than	non	`<`
Less than or equal	non	`<=`
Greater than	non	`>`
Greater than or equal	non	`>=`
Equal	non	`=`
Not equal	non	`/=`
Logical And	left	`and`
Logical or	left	`or`
Logical xor	left	`xor`
Short-circuit and	left	`and then`
Short-circuit or	left	`or else`

Notes:

a. Note that C splits the relational operators into two levels (vs. Ada or Perl), and makes the operators left associative (as does Perl). Ada makes the relationals non-associative, which is more secure, since $a < x < b$ is not the same as $a < x \wedge x < b$.

b. C splits logical *and* and *or* into separate precedence levels, while Ada does not.

12.3 Here's a table.

Function	**C**	**Ada**	**Perl**
compare	strcmp strncmp	= /= etc.	eq ne etc.
length	strlen	count	length
find substring	index	index	index
find rightmost	rindex	index	rindex
concatenation	strcat strncat	&	.
copy/assign	strcpy strncpy	=	=
category: isLetter, etc.	11	12	
find token		find_token	
translate		translate	tr
replace slice		replace_slice	
trim		trim	chomp
delete		delete	
insert		insert	
join			join
lowercase uppercase	tolower toupper		lc uc
substring			substr
pattern matching			split m// s///

In C strings are arrays of characters terminated by a `nul` character (ASCII code 0). Thus, a common error is:

```
char s[4] = "abcd";  /* length should be 5 */
```

For many of the C string functions, there are two versions: one which uses the null terminator (e.g., strcpy) and one which does not (e.g., strncpy). But in general there is no escaping the fact that strings are arrays of characters with a null byte terminator.

In contrast, strings in Ada are also arrays of characters, but once created, they cannot vary in length. Thus the string `s: string(1..7);` will always hold exactly seven characters.

In contrast, strings in Perl are truly dynamic; a string can always vary in size. Perl also supports regular expression operations on strings.

12.4 In C and Ada arrays can be statically sized or dynamically sized using a malloc or new operation. In Perl all arrays are dynamic; you can change the size of an array by assigning a value at an arbitrary index; the values at unassigned indexes are *undefined*.

Similarly, Perl provides the operations push/pop which shift new values on/off the high end of an array. Similarly Perl also provides shift/unshift operations which accomplish the same at the low end of an array.

In C and Ada an array must be homogeneous; that is, all values must be of the same type (e.g., int). In Perl different subscript positions within the array may be of different types. In fact, some subscripts positions may have scalar values while others have list values (or even objects).

Perl provides a `foreach` statement for iterating over the values of an array. Neither C nor Ada provides this.

12.5 One major difference among the three versions is in typing. Another is the syntax of function declarations.

The original version of C was based on a language named B The original version of Unix was developed for an 8K (18 bit word) PDP-7 and was written in PDP-7 assembler. Thompson then developed a language named B which was based on BCPL. Both B and BCPL were basicallly typeless with memory viewed as a contiguous array of words; there was no direct support for characters or strings. Variables were delcared as either static or dynamic as:

```
auto a, b, c;
static i, j, k;
```

The arrival of the PDP-11 exposed several inadequacies of B's semantic model. Its character handling facilities were clumsy, with characters being packed and unpacked into words. This was both silly and awkward on a byte addressable machine. Second, newer models of the PDP-11 were to support floating point; but a 16-bit word was inadequate for holding a floating point number. Third, the B/BCPL model implied overhead in dealing with pointers.

Consider the following code fragment:

```
int *a;
...
a = malloc(sizeof(int) * size);  /* K&R C */
/* a = (int *) malloc(sizeof(int) * size);  ANSI C */
/* a = new int[size];  C++ */
```

In the transition from K&R C to ANSI C, pointers became typed. K&R C added the type `void`.

Consider the following function headers:

```
double atof(s);  /* K&R C */
char s[ ];
{ ... }

double atof(char s[])  /* ANSI C */
{ ... }
```

12.6 One solution is to use safe libraries; one such library is the `strlcpy` and `strlcat` functions provided with the OpenBSD source libraries.

See en.Wikipedia.org/wiki/Buffer_overflow for a description of the problem and an overview of its solution.

12.7 scanf, fscanf, scanf.

12.8 Write a main program and a sort.c where sort.h defines the arguments to be the size of the array, followed by the array to be sorted. Have sort.c use the opposite order.

12.9 See the program `queens.c` in the *ch12code* directory.

12.10

12.11

12.12

12.13

12.14 See [Wirth, 1976, p. 143ff] for a Pascal program.

12.15 Most characters in a regular expression match themselves. One must be careful of the following meta characters: \ | ( ) [ ] { ^$ * + ? . The use of the period to represent a single wildcard character is especially unfortunate, although it follows the awk convention.

12.16 If you execute the command:

```
perl mygrep.pl string  file1 file2
cat file1 file2 | perl mygrep.pl string
```

where the string `string` occurs in both files, the line numbers in `file2` continue where `file1` left off. That is, the correct line numbers in `file2` are the reported line number minus the number of lines in file1. That is, the wo commands above give the same result.

You can fix the program by adding an outer loop, which continually shifts filenames off the command line and opens each file, resets the line number, then processes the lines in the file, and finally closes the file:

```
my $string = shift;
while (@ARGV > 0) {
   my $fname = shift;
   open(IN, "<fname");
   my $ct = 0;
   while (<IN>) {
       $ct++;
       print "$ct:\t$_" if index($_, $string) > -1;
   }
   close(IN):
}
exit;
```

12.17 `use strict;`

```
for ($_ =0; $_ < 3; $_++) {
    &countToks();
    print $_, "\n";
}
exit;

sub countToks {
    $_ = 6;
    print "tok=", $_, "\n";
}
```

12.18 `# Create a basic array.`

```
my @a = (1, 2, 3, 4, 5);
print "@a\n";

# Reference "a" as a scalar.
my $b = @a;
print "$b\n";
```

```
# Reference "b" ("a") as a list again.
my @c = @a;
print "@c\n";
```

12.19 See the program `queens.pl` in the *ch12code* directory.