Ocean Programming Language Definition

Lawrence A. Crowl

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 94-60-08

December 1994

Abstract

Ocean is a simple imperative programming language with the flavor of the C programming language. This report describes the Ocean programming language, particularly as it compares with the C programming language. The Ocean programming language has two goals:

1. to demonstrate by example that a language can be simpler and safer than the C programming language while still preserving the flavor and expressiveness of C; and

2. to require a compiler that is simple enough for students to implement in a single course while still touching on most compilation issues.

Contents

1 Introduction

Ocean is intended as an example of a language with the conciseness and expressiveness of the C programming language, but without the syntactic or semantic pitfalls of C. As such, it is similar to C in many ways. Ocean is mostly similar to K&R C [Kernighan and Ritchie, 1978], but it has elements of ANSI C [ANSI, 1988; Kernighan and Ritchie, 1988; Plauger and Brodie, 1992].

Ocean is also intended as for use in a compiler course. As such, it is simple to compile, but still touches on many of the issues in compiling imperative languages. Furthermore, it is relatively easily translatable to C.

This report defines the Ocean programming language. As a language definition, this report primarily highlights the differences between C and Ocean. Note however, that this report does not generally provide the rationale for specific design features.

Ocean is a block-structured, imperative programming language. Its primary features are as follows.

- There are a variety of primitive data types: boolean, character, word, integer, floating-point, and pointer. However, there is only one size for each "kind" of data type. (Multiple sizes are a straightforward extension.)
- The unsigned integral type is implicitly castable to the signed integral type, but not vice-versa. That is, the unsigned type is lower on the type hierarchy. Ocean's integral type order is reversed from that of C, but better reflects the mathematical order.
- Pointers are "arithmetic" and act much as they do in C.
- Operators for primitive data types are similar to those of C, but the number of precedence levels is reduced to 10. This is accomplished primarily through merging the bitwise boolean operators into the standard arithmetic precedence hierarchy. The result makes expressions easy to read and write.
- There are three data structuring types, arrays, structures, and unions. Unlike arrays in C, arrays in Ocean are first-class types. Structures and unions are also first-class.
- Ocean also explicitly recognizes both sized and unsized data structure types. Unsized types enable access to data without knowing the exact size. However, variables cannot be defined with an unsized type.
- The case labels for the switch statement may specify a range.
- The do-while statement permits a post-condition statement in addition to the pre-condition statement.
- The for-loop statement uses a full statement for iteration control.
- The goto statement is available.
- Variables may have initializers.

- Compound statements may contain variable declarations anywhere within the compound statement. Compound statements have nested scope.
- Functions may not be nested.
- Manifest constants and in-line functions replace the preprocessor.

The main body of this report is generally informal about the syntax of Ocean; the appendix contains the formal syntax.

2 Lexical Structure

Ocean programs consist of a sequence of tokens. Each token is a sequence of one or more characters from the ASCII character set.

Where possible ambiguities exist in determining the tokens, Ocean defines the interpretation to be the longest possible match reading from left to right. Tokens that might otherwise be ambiguous or confusing may be separated with whitespace. White space consists of one or more spaces or tabs. White space is not itself a token; it separates tokens.

Comments start with '`$`' and extend to the end of the line. Comments act as white space. This comment structure permits nested comments on a single line, but relies on a good text editor to comment out sections of programs.

Identifiers consist a sequence of characters starting with a letter (upper or lower case) or an underscore, and followed by zero or more letters, digits, or underscores, excluding those sequences that are keywords.

The remaining tokens are described as needed in the remainder of this document. These tokes are quoted and appear in teletype font, e.g. '`return`' or '`<=`'.

3 Definitions and Declarations

Ocean definitions and declarations are the most complex part of the language. They look similar to those of C, but are simpler. A definition associates an identifier with a type and with a storage area, a value, or both. A declaration only associates an identifier with a type, and does not allocate storage or provide a value. Variables and functions may have only one definition per program, but may have an arbitrary number of declarations, provided that the declarations match the definition. Types, manifest constants, and in-line functions may have only one definition per compilation unit, but may be defined several times within the same program. Types, manifest constants, and in-line functions have no declarations.

Variables and functions must be declared (or defined) before they are used. Identifiers declared local to a function shadow variables declared as parameters, which in turn shadow global identifiers.

Types need not be defined before being used, but no operations may be applied to objects of this type until the type is defined. This rule enables forward references to types and enables the implementation of opaque types.

3.1 General Form

Ocean definitions and declarations consist of a mode, a base type, and one or more comma separated declarators. Modes are similar to the storage classes of C, except that some names are different and the mode is required. The modes for definitions are '`public`', '`private`', '`static`', '`auto`', '`register`', '`manifest`', and '`inline`'. The modes for declarations are '`extern`' and '`forward`'. The base type is usually a simple identifier, but may be a structure or union declaration (described later). Declarators give the name and final type of variables (or types, or functions) being declared. They consist of the identifier followed by any number of *derived-type indicators* for arrays ('`[`' *number-of-elements* '`]`'), pointers ('`^`'), and functions ('`(`' *parameters* '`)`'). These indicators are the same as the operators used to access the types, so as in C, variables are "declared by example of use". In Ocean, the pointer derefence is a postfix '`^`', and not a prefix '`*`', which means that all derived-type indicators are uniformly postfix. For example, an array of integers and a pointer to integers are declared as:

```
static int ids[ 8 ], ptr^ ;
```

3.2 Variables

Modes for global variable definitions are '`public`' and '`private`'. Public means that the variable names are given to the linker, and private means that they are not. Modes for local variable definitions are '`static`', '`auto`' (automatic), and '`register`'. Static means that the variables have only one instance, which is exists throughout the lifetime of the program. Automatic means that the variables have one instance per execution of the compound statement in which the definition appears. Their lifetime is exactly that of the compound statement. Register means the same as automatic, except that the compiler should place the variable in a register rather than on the stack frame. Compilers are free to implement a register variable as an automatic variable. Also, because registers (generally) have no address, the "address of" operator does not apply to register variables.

Definitions may have initializers, as in C, except that the assignment operator is '`:=`' rather than '`=`'. The initializers for '`public`', '`private`', and '`static`' definitions must be constant expressions (evaluable at compile time), but the '`auto`' and '`register`' initializers may be dynamic expressions.

3.3 Externals and Forwards

Variables and functions defined in another compilation unit are declared in the current compilation unit with the external statement, which is like that of a global variable definition, except that the mode is '`extern`' and initializers are not allowed.

Variables and functions defined later in the current compilation unit are declared in the current compilation unit with the forward statement, which is like that of external declarations except that the mode is '`forward`'.

3.4 Manifest Constants

Manifest constants are symbolic names for compile-time constants. The syntax is like that of variable definition, except that the mode is '`manifest`' and initializers are required. Furthermore, local manifest constants must have constant (exression) initializers; dynamic initializers are not permitted. For example,

```
manifest int a := 8*20 ;
```

3.5 Members and Parameters

The declaration of parameters, structure members, and union members share the same form, which is a base type followed by a comma separated list of declarators. For example,

```
int alpha, beta ;
float gamma ;
```

3.6 Functions

The function definitions are like those of global variables (i.e. '`public`' and '`private`'), except the initializer is a compound statement rather than an expression and there is no assignment operator. The function type is derived from its return type and its parameter types. Unlike K&R C, Ocean functions declarations include their parameters. Like K&R C, Ocean function definitions include their parameters. For example,

```
extern void malloc( int n )^ ; $ returns a pointer to void
forward int double( int n ) ;
public int quadruple( int n ) { return double( double( n ) ) ; }
```

3.7 In-Line Functions

The in-line function syntax is like that of regular function definitions, except that the mode is '`inline`' and initializers are required. In-line functions have the semantics of regular functions, not the semantics of macro substitution. There are several consequences to this interpretation of in-line functions.

- Because regular functions can not be defined locally, inline functions also cannot be defined locally.
- All symbols used in the inline function must have been previously declared.
- Parameters are pass-by-value, not pass-by-name.
- Compilers can implement an in-line function the same way as an out-of-line function without breaking anything. (Of course, implementing in-line functions this way defeats the purpose.)

3.8 Type Definitions

The definition of symbolic names for types is similar to that of variables, except that the mode is '`typedef`' and initializers are not allowed. Type definitions may appear locally as well as globally. Type definitions define an *alias* for a type, they do not define a new type. As in C, Ocean types are structurally equivalent.

3.9 Type Specifiers

Some Ocean constructs require a type specifier independent of any specific declaration. In these cases, the syntax for a type specifier is like that of a member declaration, except that the identifier is omitted. That is, a type specifier consists of a base type and a sequence of zero or more derived-type indicators. So, the type "pointer to an array of integers" has the specifier '`int ^[]`'.

4 Types

Ocean provides seven primitive types. void, boolean, character, word, integer, floating-point, and pointer. The word, integer, and floating-point types are collectively called numeric types. The word, integer, floating-point, and pointer types are collectively called linear types (because they may be compared). The C language overloads the concepts of booleans, characters, machine words, and integers onto one type, the machine word (called an (`unsigned`) `int`). Ocean separates those concepts into distinct numeric types.

Ocean provides three data structuring types, the array, the structure, and the union. Within these types, Ocean distinguishes between *sized* types and *unsized* types. Sized types have a size associated with them, and as such may be allocated space and copied. Unsized types have no size, so they cannot be allocated space or copied. A consequence is that variables, parameters, and function results may not have an unsized type. Furthermore, unsized types may not be assigned. Unsized types must be handled indirectly, via a pointer. Pointers are always sized types, even if what they point to is unsized.

Ocean types are structurally equivalent, as in C.

See section 5 for a description of the operations that apply to these types.

4.1 Void

Void is the primitive unsized type. Unlike all other unsized types, void is a legal function return value. This is used to indicate functions that have no useful return value.

4.2 Boolean

The boolean type is identified by the keyword '`bool`'. It's literals are '`true`' and '`false`'.

4.3 Character

The character type is identified by '`char`'. Character literals consist of a printable character (including space and tab but excluding newline) enclosed within single quotes. Examples are '`'H'`' and '`'"'`'.

Within quotes, the '`\`' character begins an escaped character. In both character and string literals, the following sequences represent a single, escaped character: '`\b`' (backspace), '`\e`' (escape), '`\n`' (newline), '`\t`' (horizontal tab), '`\v`' (verticle tab), '`\'`' (single quote), '`\"`' (double quote), '`\\`'

(backslash), and '\w.' (the character with the code w where w is a word number constant, see below). All other backslash sequences are illegal.

Simple character string literals consist of zero or more printable or escaped characters within double quotes. As in C, the value of a character string literal is a pointer to an array of characters containing the given characters and terminated by the null character, ''\0.''. A complex character string literal consists of more than one adjacent simple string literals, and represents the concatenation of the simple literals. This enables the definition of long string literals with relatively short source lines.

4.4 Word

Ocean represents machine words explicitly with the 'word' type identifier. Word values are unsigned binary numbers of a fixed number of digits, i.e. a bit string. All arithmetic operations on words are done modulo the word size of the computer. No overflow is possible.

Word literals consist of one or more digits, in binary, octal, decimal or hexadecimal bases. Binary, octal, and hexadecimal constants are specified by placing '0b', '0q', and '0x' before the digits, respectively. The hexidecimal digits 'A'–'F' and 'a'–'f' represent 10–15, respectively.

4.5 Integer

The signed integer type is identified by 'int'. Integers do not have a bit string interpretation because the representation of sign may vary. The result of an overflow for integer operations is undefined, and may include aborting the program or ignoring the overflow.

There are no integer literals. You can achieve the effect of an integer literal by placing a '-' sign before a word literal. Alternatively, you can cast the word value to an integer value.

Floating-Point The signed floating-point number type is identified by 'float'. The result of an overflow or an underflow for floating-point operations is undefined, and may include aborting the program or ignoring the overflow.

Floating-point literals consist of one or more digits followed by a decimal part, an exponent part, or both. Decimal parts consist of a decimal point '.' followed by zero or more digits. Exponent parts consist of the letter 'E' (or 'e'), an optional sign, '+' or '-', and one or more digits. Again, there are no negative literals, but a unary '-' will produce a negative constant.

4.6 Pointer

Pointers are declared as in C, except that the derived-type indicator is a postfix '^' rather than a prefix '*'. Likewise the dereference operator is a postfix '^'. For example, an automatic integer pointer variable 'ip' is defined with

```
auto int ip^;
```

And is dereferenced as in

```
i := ip^;
```

The only pointer literal is '`nil`', which is a nil pointer to the void type. To obtain nil pointers to other types, one must cast '`nil`' to the appropriate type.

Pointers are implicitly converted to pointers to void when necessary to obtain a legal interpretation of an expression.

The arithmetic operators '`+`' and '`-`' may be applied to pointers with a sized base type, but not an unsized base type. Pointers to which '`+`' and '`-`' operations are applied must point to elements of an array.

4.7 Array

Arrays are declared as in C, but they are first-class objects. The first-class nature of arrays means you may assign arrays and pass arrays as parameters. It also means that there is no implicit conversion from the name of an array to a pointer to its first element.

The array derived-type indicator is '`[`' *expression* '`]`' where the expression is a word (non-negative) constant specifying the number of elements in the array. The elements are indexed from 0 to $n-1$. The array type includes the element type, which must be a sized type. Array types may also include the number of elements as part of their type. These are sized array types. Programs may assign arrays and pass array parameters so long as the number of elements in both source and destination is identical.

For example, an automatic integer array variable '`ia`' is defined with

```
auto int ia[3];
```

And is indexed as in

```
i := ia[j];
```

If the array declaration omits the number of elements, the array is *unsized.* Unsized arrays have restricted use, as discussed earlier. Where necessary, pointers to sized arrays are implicitly converted to pointers to unsized arrays. This is useful for passing parameters to library functions. For example, the C function `strcpy` has the Ocean declaration and use:

```
extern char strcpy( char s1^[], s2^[] )^[];
static char source[ 10 ], target[ 20 ];
strcpy( target@, source@ );
```

(The '`@`' operator is a postfix operator equivalent to the C prefix '`&`' operator.)

4.8 Structure and Union

Structures and unions are as in C, excluding tags. The description consists of a '`struct`' or '`union`' followed by the member declarations (as discussed earlier) enclosed in braces. The last '`;`' in the member declarations may be omitted. For example, the following defines two structure variables, each containing three members.

```
public struct { int i, j; float x } s, r;
```

All but the last member of a structure must have a sized type. If the last member has a sized type, the structure has a sized type. If the last member is unsized, the structure has an unsized type. Any member of a union may be unsized, in which case the union is unsized. In both cases, the same restrictions apply to unsized structures and unions as apply to unsized arrays. An example of an unsized structure is a counted character string.

```
typedef struct { bits length; char text[] } counted_string;
```

4.9 Functions

Functions are as in C, but there is no implicit conversion from the name of a function to a pointer to that function. Arguments to a function must match the types of the parameters to that function. Parameters are pass-by-value. A Ocean function is call compatible with an equivalent C function. This means that Ocean functions can call C functions. C functions can also call Ocean functions, except those functions with array parameters or results.

Where ANSI C would define a function:

```
int foobar( int n, int m, float x ) { int o = n+m; return o+x; }
```

Ocean declares the function parameters using the structure member syntax:

```
public int foobar( int n, m; float x ) { auto int o := n+m; return o+x; }
```

5 Expressions

Expressions compute values or designate variables. Expressions are composed of variable names, literal values, and prefix, infix, and postfix operators. Operators have several precedence levels:

highest	lvalue	data structure access
	primary	function calls, casts, etc.
	unary	negation, bitwise invert, etc.
	power	power, logarithm, shift
	product	multiply, divide, modulo, bitwise and
	sum	add, subtract, bitwise ior, bitwise xor
	relation	comparisons
	conjunction	logical, short-circuit and
	disjunction	logical, short-circuit or
lowest	assignment	assignments

The unary and assign operators associate right-to-left, and all other operators associate left-to-right. In addition, expressions may be enclosed in parentheses to override the operator precedence or associativity.

5.1 Operations

This section defines the operators and their allowable operands. Within the tables of this section, operators are defined with an indication of the type and form of their arguments, as follows.

indicator	meaning	indicator	meaning
a	array rvalue	b	boolean rvalue
c	character rvalue	f	floating-point rvalue
g	function rvalue	i	integer rvalue
l	linear rvalue	n	numeric rvalue
o	an operator	p	pointer rvalue
s	structure rvalue	t	type specifications
u	union rvalues	w	word rvalue
z	any rvalue		

The upper case variants indicate an lvalue rather than an rvalue.

Where needed to provide a legal interpretation of an expression, there are implicit conversions from '`char`' to '`word`', from '`word`' to '`int`', and from '`int`' to '`float`'.

precedence	operation	result	meaning and notes
lvalue	S`.ident`	Z	— select a structure member
	U`.ident`	Z	— select a union member
	ZA`[`i`]`	Z	`(`ZA `@ !`zt`^! +` i`)^` — index an array
	ZP`[`i`]`	Z	`(`ZP `+` i`)^` — index a pointer
	Zp`^`	Z	— dereference a pointer

precedence	operation	result	meaning and notes
primary	Z`@`	zp	— a pointer to the storage of the lvalue
	G`(...)`	z	— call a function, argument types must be implicitly castable to parameter types
	L`++`	l	`((`L `+= 1) - 1)` — increment, return old value
	L`--`	l	`((`L `-= 1) + 1)` — decrement, return old value
	`sizeof !`t`!`	w	— size of the type in addressable units
	c `!word!`	w	— the ordinal position of the character
	w `!int!`	i	— the integral value of the word, may overflow
	w `!float!`	f	— the floating-point value of the word, large words may also lose precision
	i `!float!`	f	— the floating-point value of the integer, large integers may also lose precision
	c `!int!`	i	c `!word! !int!`
	c `!float!`	f	c `!word! !float!`
	f `!int!`	i	if $f > 0$ then $\lfloor f \rfloor$ else $\lceil f \rceil$ — truncated
	i `!word!`	w	$i \pmod{2^{\text{wordsize}}}$
	w `!char!`	c	— the character with ordinal position w
	f `!word!`	w	$\lfloor f \pmod{2^{\text{wordsize}}} \rfloor$
	f `!char!`	c	f `!word! !char!`
	i `!char!`	c	i `!word! !char!`
	z_1p `!`z_2tp`!`	z_2p	— casting the pointer to the given type
	z_1 `!`z_2`!`	z_2	— any cast structurally equivalent to the above
	p `isnil`	b	— '`true`' if p is nil, '`false`' otherwise

precedence	operation	result	meaning and notes
unary	`~`b	b	— the boolean one's complement
	`~`w	w	— the bit-by-bit one's complement
	`-`i	i	— integer negation
	`-`f	f	— floating-point negation
	`++`L	l	`(`L `+= 1)` — increment, return new value
	`--`L	l	`(`L `-= 1)` — decrement, return new value
	`sizeof` z	w	— size of the value in addressable units

precedence	operation	result	meaning and notes
power	$w_1\texttt{<<}w_2$	w	$w_1 \times 2^{w_2} \pmod{2^{\text{wordsize}}}$ — left shift
	$i\texttt{<<}w$	i	$i \times 2^{w}$ — left shift
	$f\texttt{<<}w$	f	$f \times 2^{w}$ — add to exponent
	$w_1\texttt{>>}w_2$	w	$\lfloor w_1 \div 2^{w_2} \rfloor$ — right shift
	$i\texttt{>>}w$	i	$\lfloor i \div 2^{w} \rfloor$ — right shift
	$f\texttt{>>}w$	f	$f \div 2^{w}$ — subtract from exponent
	$w_1\texttt{**}w_2$	w	$w_1^{w_2} \pmod{2^{\text{wordsize}}}$
	$i\texttt{**}w$	i	i^{w}
	$f_1\texttt{**}f_2$	f	$f_1^{f_2}$
	$w_1\texttt{//}w_2$	w	$\lfloor \log_{w_2} w_1 \rfloor$ — undefined if $w_1 = 0 \vee w_2 \leq 1$
			— $w\texttt{//}2$ is the index of the highest set bit
	$f_1\texttt{//}f_2$	f	$\log_{f_2} f_1$ — undefined if $f_1 \leq 0 \vee f_2 \leq 1$

precedence	operation	result	meaning and notes
product	$w_1\texttt{*}w_2$	w	$w_1 \times w_2 \pmod{2^{\text{wordsize}}}$
	$i_1\texttt{*}i_2$	i	$i_1 \times i_2$
	$f_1\texttt{*}f_2$	f	$f_1 \times f_2$
	$w_1\texttt{/}w_2$	w	$\lfloor w_1 \div w_2 \rfloor$ — undefined if $w_2 = 0$
	$i_1\texttt{/}i_2$	i	truncate($i_1 \div i_2$) — undefined if $i_2 = 0$
	$f_1\texttt{/}f_2$	f	$f_1 \div f_2$ — undefined if $f_2 = 0$
	$w_1\texttt{\%}w_2$	w	$w_1 \bmod w_2$ — undefined if $w_2 = 0$
	$b_1\texttt{\&}b_2$	b	boolean $b_1 \vee b_2$
	$w_1\texttt{\&}w_2$	w	bitwise $w_1 \vee w_2$

precedence	operation	result	meaning and notes
sum	$w_1\texttt{+}w_2$	w	$w_1 + w_2 \pmod{2^{\text{wordsize}}}$
	$i_1\texttt{+}i_2$	i	$i_1 + i_2$
	$f_1\texttt{+}f_2$	f	$f_1 + f_2$
	$p\texttt{+}i$	p	— the i^{th} element higher in memory
	$w_1\texttt{-}w_2$	w	$w_1 - w_2 \pmod{2^{\text{wordsize}}}$
	$i_1\texttt{-}i_2$	i	$i_1 - i_2$
	$f_1\texttt{-}f_2$	f	$f_1 - f_2$
	$p_1\texttt{-}p_2$	p	$i_1 - i_2$ where $p_1{=}p_3 + i_1$ and $p_2{=}p_3 + i_2$
	$b_1 \vert b_2$	b	boolean $b_1 \wedge b_2$
	$w_1 \vert w_2$	w	bitwise $w_1 \wedge w_2$
	$b_1\texttt{\~{}}b_2$	b	boolean $b_1 \oplus b_2$
	$w_1\texttt{\~{}}w_2$	w	bitwise $w_1 \oplus w_2$

precedence	operation	result	meaning and notes
relation	n_1`<`n_2	b	$n_1 < n_2$
	p_1`<`p_2	b	$i_1 < i_2$ where $p_1 = p_3 + i_1$ and $p_2 = p_3 + i_2$
	n_1`>`n_2	b	$n_1 > n_2$
	p_1`>`p_2	b	$i_1 > i_2$ where $p_1 = p_3 + i_1$ and $p_2 = p_3 + i_2$
	n_1`==`n_2	b	$n_1 = n_2$
	p_1`==`p_2	b	— p_1 and p_2 are nil or point to the same location
	l_1`<=`l_2	b	`~(`l_1`>`l_2`)`
	l_1`>=`l_2	b	`~(`l_1`<`l_2`)`
	l_1`<>`l_2	b	`~(`l_1`==`l_2`)`

precedence	operation	result	meaning and notes
conjunction	b_1`&&`b_2	b	if b_1 then b_2 else `false` — short-circuit

precedence	operation	result	meaning and notes
disjunction	b_1`||`b_2	b	if b_1 then `true` else b_2 — short-circuit

precedence	operation	result	meaning and notes
assignment	Z_1`:=`z_2	z_1	— copy z_2`!`$z_1 t$`!` to the location Z_1, the result is the assigned value z_1
	Z_1 o`=`z_2	z_1	Z_1`:=`Z_1 o `(`z_2`)` — Z_1 is evaluated only once and o is one of '`<<`', '`>>`', '`**`', '`//`', '`*`', '`/`', '`%`', '`&`', '`+`', '`-`', '`|`', or '`~`'

5.2 Predefined Functions

Implementations must provide the following predefined functions.

```
extern char get_char( ); $ get the next character from stdin
extern void put_char( char character ); $ put the character to stdout
extern word word_abs( int number ); $ the absolute value of the number
extern int ceiling( float number ); $ the ceiling of the number
extern int floor( float number ); $ the floor of the number
extern int truncate( float number ); $ the trunctation of the number
extern int round( float number ); $ the number rounded, prefering evens
extern float float_abs( float number ); $ the absolute value of the number
extern void allocate( word space )^; $ allocate new memory for space
```

6 Statements

Statements specify the sequence of expressions to evaluate. Their value is in controlling the side-effects of expressions.

6.1 Empty

A single semicolon is an empty statement. No action will be performed.

6.2 Expression

An expression followed by a semicolon is an expression statement. The result of the expression is discarded. A warning if the result is not void or not the result of an assignment is desireable.

6.3 Compound

A compound statement is a sequence of zero or more statements or local declarations enclosed by braces. Unlike C, local declarations may appear anywhere in the block, not just at the beginning. However, variables must still be defined before use.

6.4 If

The if statement has the forms:

> '`if`' '`(`' boolean-expression '`)`' statement
> '`if`' '`(`' boolean-expression '`)`' statement '`else`' statement

The ambiguity of statements like:

```
if ( a < b ) if ( c < d ) e := f ; else g := h ;
```

is resolved by associating each '`else`' with the nearest possible '`if`'.

6.5 Switch

The switch statement has the form:

> '`switch`' '`(`' integer-expression '`)`' statement

where the sub-statement is typically a compound statement. The switch statement is like that of C, with switch cases labelled the same way, except that Ocean provides for case ranges:

> '`case`' constant-integer-expression '`:`' statement
> '`case`' constant-integer-expression '`to`' constant-integer-expression '`:`' statement
> '`default`' '`:`' statement

In addition, the break statement will exit the (innermost enclosing) switch statement:

> '`break`' '`;`'

6.6 While

The following statement provides for repetitive execution.

‘`while`’ ‘(’ boolean-expression ‘)’ statement

The break statement will exit the while statement. The continue statement will restart the while statement at the condition.

‘`continue`’ ‘;’

6.7 Do-While

Input/output often prefers conditions in the middle of a loop, rather than at the beginning or end. Ocean provides middle-condition loops with the do-while statement.

‘`do`’ statement_1 ‘`while`’ ‘(’ boolean-expression ‘)’ statement_2

The do-while statement is equivalent to:

statement_1 ‘`while`’ ‘(’ boolean-expression ‘)’ ‘{’ statement_2 statement_1 ‘}’

except that the continue will restart at the (innermost enclosing) ‘`do`’, i.e. at statement_1. The break statement will exit the (innermost enclosing) while statement.

An Ocean do-while statement with an empty statement_1 is equivalent to the C while statement; an Ocean do-while statement with an empty statement_2 is equivalent to the C do-while statement.

The do-while statement is intended to be used in situations like the following:

```
do                  c := getchar( );
while ( c <> EOF )  putchar( c );
```

Implementing these semantics in C requires more complicated code.

6.8 For-Loop

Iteration is provided by the for-loop statement.

‘`for`’ ‘(’ bookean-expression ‘)’ statement_1 ‘`loop`’ statement_2

The for-loop statement is equivalent to:

‘`while`’ ‘(’ boolean-expression ‘)’ ‘{’ statement_2 statement_1 ‘}’

except that the continue will restart at statement_1 of the (innermost enclosing) for-loop statement. The break statement will exit the (innermost enclosing) for-loop statement.

The for-loop is intended to be used as follows:

```
{ int i := 0;
  for ( i < 100 ) i++;
  loop            j += a[i];
}
```

The Ocean for-loop (used as intended) provides two advantages over the C for statement. It puts declaration and initialization of iteration variables in the same place and it permits an arbitrary statement as the iterator. The latter advantage more than makes up for the loss of the C '`,`' and '`?:`' operators.

6.9 Return

The following statements provide a function return value and transfer control back to the calling function.

> '`return`' '`;`'
> '`return`' expression '`;`'

The first form is for void functions. The second form is for non-void functions, and the type of the expression must match that of the function definition.

6.10 Goto

The following statement unconditionally transfers control.

> '`goto`' label-identifier '`;`'

Since the goto exists, statements may be labeled, as in C.

> label-identifier '`:`' statement

A Grammar

Within this section, required items are grouped by parentheses, ( ), optional items are enclosed in square brackets, [ ], zero or more occurences of items are enclosed in curly braces, { }, and alternative items are separated by vertical bars, |.

A.1 Tokens

Character sets are enclosed by angle brackets, ⟨ ⟩. Within character sets, characters from a contiguous range are specified by the numerically extreme points separated by an en-dash, –. For example, the lower case letters are ⟨`a`–`z`⟩.

boolean = ‘true’ | ‘false’
decimal = ⟨0–9⟩ { ⟨0–9_⟩ }
word = decimal | ‘0b’ ⟨01⟩ { ⟨01_⟩ } | ‘0q’ ⟨0–7⟩ { ⟨0–7_⟩ } | ‘0x’ ⟨0–9A–Fa–f⟩ { ⟨0–9A–Fa–f_⟩}
float = decimal ‘.’ decimal | decimal [‘.’ decimal] ⟨Ee⟩ [⟨+-⟩] decimal
identifier = ⟨A–Za–z_⟩ { ⟨0–9A–Za–z_⟩ }
character = ‘'’ (⟨!"#$%&()*+,./0-9:;<=>?@A-Z[]^_‘a-z{}~-|⟩
| space | tab | ‘\’ ⟨betnv\'"⟩ | ‘\’ word ‘.’) ‘'’
string = ‘"’ { ⟨!#$%&'()*+,./0-9:;<=>?@A-Z[]^_‘a-z{}~-|⟩
| space | tab | ‘\’ ⟨betnv\'"⟩ | ‘\’ word ‘.’ } ‘"’

A.2 Grammar

lvalue = identifier | lvalue ‘.’ identifier | lvalue ‘[’ expression ‘]’ | primary ‘^’
literal = boolean | character | word | float | string { string }
primary = literal | lvalue | lvalue ‘@’ | lvalue ‘(’ expressions ‘)’ | lvalue (‘++’ | ‘--’)
| ‘sizeof’ ‘!’ typespec ‘!’ | primary ‘!’ typespec ‘!’ | ‘nil’ | primary ‘isnil’
| ‘(’ expression ‘)’
unary = primary | (‘++’ | ‘--’) lvalue | (‘~’ | ‘-’) unary | ‘sizeof’ unary
power = unary | power (‘<<’ | ‘>>’ | ‘**’ | ‘//’) unary
product = power | product (‘*’ | ‘/’ | ‘%’ | ‘&’) power
sum = product | sum (‘+’ | ‘-’ | ‘~’ | ‘|’) product
relation = sum | sum (‘<’ | ‘<=’ | ‘>=’ | ‘>’ | ‘==’ | ‘<>’) sum
conjunction = relation | conjunction ‘&&’ relation
disjunction = conjunction | disjunction ‘||’ conjunction
expression = disjunction | lvalue (‘:=’ | ‘<<=’ | ‘>>=’ | ‘**=’ | ‘//=’ | ‘*=’ | ‘/=’
| ‘%=’ | ‘&=’ | ‘+=’ | ‘-=’ | ‘~=’ | ‘|=’) expression
expressions = [{ expression ‘,’ } expression]
statement = ‘;’ | expression ‘;’
| ‘break’ ‘;’ | ‘continue’ ‘;’ | ‘goto’ identifier ‘;’ | ‘return’ [expression] ‘;’
| (identifier | ‘case’ expression [‘to’ expression] | ‘default’) ‘:’ statement
| ‘switch’ ‘(’ expression ‘)’ statement
| ‘if’ ‘(’ expression ‘)’ statement [‘else’ statement]
| [‘do’ statement] ‘while’ ‘(’ expression ‘)’ statement
| ‘for’ ‘(’ expression ‘)’ statement ‘loop’ statement
| compound
value = expression | ‘{’ [value { ‘,’ value }] ‘}’
initializer = [‘:=’ value]
typebase = identifier | ‘bool’ | ‘char’ | ‘word’ | ‘int’ | ‘float’ | ‘void’
| ‘struct’ ‘{’ parameters ‘}’ | ‘union’ ‘{’ parameters ‘}’
typespec = typebase { ‘^’ | ‘[’ ‘]’ | ‘[’ expression ‘]’ | ‘(’ parameters ‘)’ }
declarator = identifier { ‘^’ | ‘[’ ‘]’ | ‘[’ expression ‘]’ | ‘(’ parameters ‘)’ }
member = typebase declarator { ‘,’ declarator }
parameters = [member { ‘;’ member } [‘;’]]
declaration = (‘extern’ | ‘forward’ | ‘typedef’) typebase declarator { ‘,’ declarator }

manifest = '`manifest`' typebase declarator '`:=`' value { '`,`' declarator '`:=`' value }
localdef = ('`auto`' | '`register`' | '`static`') typebase declarator initializer { '`,`' declarator initializer }
compound = '`{`' { declaration '`;`' | manifest '`;`' | localdef '`;`' | statement } '`}`'
globaldef = ('`public`' | '`private`') typebase declarator initializer { '`,`' declarator initializer }
funcdef = ('`inline`' | '`public`' | '`private`') typebase declarator compound
compilation = { declaration '`;`' | manifest '`;`' | globaldef '`;`' | funcdef }

References

[ANSI, 1988] American National Standards Institute, *Preliminary Draft Proposed Standard — The C Language*, 1988.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, 1978.

[Kernighan and Ritchie, 1988] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632, second edition, 1988.

[Plauger and Brodie, 1992] P. J. Plauger and Jim Brodie, *ANSI and ISO Standard C Programmers Reference*, Microsoft Press, One Microsoft Way, Redmond, Washington 98052-6399, 1992.