The Revised Report on the C° Programming Language

The programming assignments in this course will all be components of a single major project. The ultimate goal of this project will be to construct a complete compiler for a simple programming language. C° (pronounced "C diminished") is one of the languages you may implement as a project for this course.

C° is a not very general purpose programming language whose syntax derives largely from C. C° is, however, semantically more similar to Pascal than C. In designing C°, I have attempted to eliminate as many unnecessary details as possible while retaining enough features to ensure that the construction of a C° compiler will expose you to many of the major issues one must address when constructing a compiler for a real language.

This document includes a formal description of the syntax of C° and an informal discussion of its semantics.

1 Lexical Issues

1.1 Keywords

The following strings are recognized as keywords of the C° language. The case of the characters in a keyword is not significant. That is, the strings 'void', 'VOID' and 'VoId' are all recognized as forms of the keyword **void**.

do	**else**	**for**	**if**
int	**program**	**return**	**struct**
typedef	**void**	**while**	

1.2 Identifiers

An identifier is a sequence of characters other than a keyword that begins with an alphabetic character and is composed of alphabetic characters and decimal digits. The case of alphabetic characters in identifiers is not significant.

1.3 Punctuation

The comma (',') and semicolon (';') are used as punctuation in contexts where the syntax of C° calls for a list of items. When used, commas separate list items while semicolons terminate items.

Parentheses ('(' and ')') are used to delimit the formal and actual parameter lists associated with functions. Square brackets ('[' and ']') are used around subscript expressions and array size specifiers. Braces ('{' and '}') are placed before and after statement lists, the bodies of functions and the body of each complete program.

1.4 Operators

The following symbols and pairs of symbols are recognized as operators of the C° language.

```
+    -    /    *
%    <    <=   >
>=   ==   !=   !
&&   ||   =    .
```

1.5 Constants

An integer constant is an unbroken string of decimal digits.

A character constant is formed by placing a single ASCII character or one of the escape sequences described below between a pair of single quotes.

Within character constants the following escape sequences can be used to specify that the special character shown should be used as the value of the constant.

escape sequence	designated character
\n	newline
\\	backslash
\'	single quote

Note that C° does not recognize character string constants (or provide any other support for manipulating character strings).

1.6 White Space

Space characters and tabs are considered *white space* characters. Their presence is ignored except in character constants and in cases where they terminate an identifier or integer constant. Newlines are also considered space characters with the further restriction that no lexical item can contain a newline character.

1.7 Comments

A pair of adjacent minus signs ('- -') indicates a comment. The minus signs themselves and all characters following them on a line are ignored.

2 Program Structure

< program > → **program** {
< type definition list >
< body >
}

< body > → < variable declaration list > < function definition list > < statement list >

Each program consists of a possibly empty list of type definitions, and a < body >. A < body > is composed of optional lists of variable declarations and function definitions followed by a list of executable statements. Note that type definitions can only appear in the header of the main program.

3 Types

< type definition list > → < type definition list > < type definition >
| ϵ

< type definition > → **typedef** < type specification > < identifier > ;

< type specification > → < type name > [< integer constant >]
| **struct** { < field list > }

< type name > → < identifier >
| **int**

< field list > → < field specification >
| < field list > < field specification >

< field specification > → < type name > < identifier > ;

A < type definition > associates an identifier with a structured type. This name can then be used to declare variables or parameters of the type and to describe other structured types that have the elements of the defined type as sub-components. All type names must be declared before they can be used. In particular, C° does not support recursive types.

Note, that in C°, one must introduce and use type names for all structured types you wish to use in a program. The language does not allow one to use explicit array or record type specifications outside of the type definition part of the main program.

The only scalar type provided in C° is the type integer. As noted below, character constants are treated as integer values and the values 0 and 1 are used to represent the Boolean values 'true' and 'false'.

The type name included in an array specification determines the element type of the array. The integer constant in an array specification specifies the number of elements in the array. The elements of an array are numbered starting at zero. Thus, the specification

int[5]

describes an array of integers with elements 0, 1, 2, 3 and 4 but no element 5. Arrays are single dimensional, but the element type of an array may be some earlier defined array type. Thus, arrays of arrays of ... may be defined.

Structure types in C° are similar to strutures in C except that, in the interest of syntactic simplicity, each field must be specified separately (i.e. one cannot include a list of identifiers in a field specification). A particular < identifier > may only name one field of a given record type, but can be reused as a variable name or as the name of a field of any other record type.

4 Variable Declarations

< variable declaration list > → < variable declaration list > < variable declaration >
| ϵ

< variable declaration > → < type name > < identifier > ;

A variable declaration introduces a name under which the program can store values during execution and specifies the type of values that can be associated with the name during execution. Variable declarations may appear in the body of the main program or in the body of any function. The scope of a declaration is the entire < body > within which it occurs. The names of all variables and functions declared within a single body must be unique.

5 Functions

< function definition list > → < function definition list > < function definition >
| ϵ

< function definition > → **void** < identifier > < formals part > {
< body >
}

| **int** < identifier > < formals part > {
< body >
} ;

< formals part > → (< formals list >)
| ( )

< formals list > → < formals list >, < formal declaration>
| <formal declaration>

< formal declaration > → < type name > * < identifier >
| **integer** < identifier >

Functions may be defined in any < body > within a program. Functions may only return integer values as results. Functions declared to return **void** are just procedures.

The scope of the name associated with a function is the entire < body > in which the function's definition occurs. In particular, there is no restriction against forward references to functions. Functions may be called recursively.

Parameters may be passed to a function using either call-by-value or call-by-reference. The inclusion of a * in a formal declaration indicates that call-by-reference should be used when handling that parameter. All other parameters are passed by value. Only scalar values (integers) can be passed by value.

For the purposes of scoping, formal declarations are treated as if they occurred within the body of the function. This implies that the formal parameter names of a function must be distinct from the names of variables and functions declared within its body.

6 Statements

< statement list > → < statement list > < statement >
| < statement >

< statement > → < assignment statement > ;
| < call statement > ;
| < return statement > ;
| < if statement >
| < while statement >
| < for statement >

6.1 Assignment

< assignment statement > → < variable > = < expressions >

The value associated with the variable is replaced by the value obtained by evaluating the expression. The variable and the expression must both be of type integer. Assignments of structured values is not supported. The interpretation of the variable is completed before the expression is evaluated.

6.2 Function Calls

< call statement > → < identifier > (<actual list >)
| < identifier > ( )

< actual list > → < actual list > , < expression >
| < expression >

The number of actual parameter expressions included in a call statement must match the number of formal parameters declared in the function's definition. The type of each actual parameter expression must match the type of the corresponding formal. If any of the called function's formal parameters are specified as call-by-reference parameters, then the corresponding actual parameters must actually be variables as defined in section 7.

6.3 Return Statements

< return statement > → **return**
| **return** < expression >

A return statement causes return of control to the caller of the function in which it is executed. Executing a return statement in the main program causes program termination.

Return statements placed in functions declared to return **void** may not include an expression. Return statements placed in all other functions must include an expression of type integer. This expression is evaluated and its value returned as the value of the function call.

There is an implicit return statement at the end of the body of each function that returns void.

6.4 If Statements

< if statement > → **if** (< expression >)< statement part >
< else part >

< else part > → **else** < statement part >

$$\mid \epsilon$$

$$< \text{statement part} > \rightarrow \{ < \text{statement list} > \}$$
$$\mid \ < \text{statement} >$$

C° does not recognize Boolean values as a separate type. Instead, it treats the integer value 0 as 'false' and any non-zero value as 'true'. Thus, to execute an if statement, the expression is first evaluated. It must evaluate to an integer. If its value is non-zero, the statement or statement list following the expression is executed after which execution resumes with the statement following the if statement. If the expression's value is zero and a non-empty else part was included, the statement or statement list found in the else part is executed. If the else part is empty and the expression's value is zero, then control simply passes to the statement after the if statement.

6.5 While Statements

$$< \text{while statement} > \rightarrow \textbf{while} \ (< \text{expression} >)$$
$$< \text{statement part} >$$

The execution of a while loop begins with the evaluation of the expression included in the statement's header. If this expression's value is zero, execution of the while loop terminates immediately. Otherwise, the statement or statement list included is executed repeatedly until the value of the expression becomes zero.

6.6 For Statements

$$< \text{for statement} > \rightarrow \textbf{for} \ (< \text{assignment statement} > ; < \text{expression} > ; < \text{assignment statement} >)$$
$$< \text{statement part} >$$

The syntax of the **for** statement in C° is based on the most common use of the corresponding statement in C rather than on the actual syntax of the C for statement. In particular, the first and third items in the header of the for can only be assignment statements.

The first step in the execution of a for statement is the execution of the first assignment in its header. After this is done, the expression included in the header is evaluated. If its value is 0, the execution of the statement is complete. Otherwise, the statement part is executed and the assignment included as the third component of the header is executed repeatedly until evaluation of the expression yields 0.

7 Variables

$$< \text{variable} > \rightarrow < \text{identifier} >$$
$$\mid \ < \text{variable} > . < \text{identifier} >$$
$$\mid \ < \text{variable} > [< \text{expression} >]$$

A variable is either 1) a simple identifier, 2) a sub-variable followed by a field selector, or 3) a sub-variable followed by an array subscript expression. If a variable is a simple identifier the identifier must be bound by a variable declaration or a formal parameter declaration in an enclosing scope. If the variable is a sub-variable followed by a field selector, the sub-variable must denote an object of a record type that includes the component specified by the selector. If the variable is a sub-variable followed

by a subscript expression, the sub-variable must denote an array and the value of the expression at run-time must be non-negative and less than the number of elements in the array.

8 Expressions

8.1 Logical Expressions

< expression > → < logical term >
| < expression > `||` < logical term >

< logical term > → < logical factor >
| < logical term > `&&` < logical factor >

< logical factor > → < relational expression >
| `!` <relational expression>

C° provides the logical 'and' (`&&`), 'or' (`||`) and 'not' (`!`) operators. In C°, these operators are actually applied to integers. They interpret their operands using the usual scheme: 0 represents 'false' and any non-zero value represents 'true'. The logical and relational operators of C° all produce the value 1 when the result of an operation is 'true'.

The `!` operator has the highest precedence among the logical operators followed by `&&` and finally `||`. Operators of equal precedence are grouped from left to right.

The `&&` and `||` operators are evaluated lazily in C°. That is, if the first operand of an `&&` operator is zero, a result of zero is returned without ever evaluating the second operand. Similarly, if the first operand of an `||` operator is non-zero, the second operand will not be evaluated.

8.2 Relational Expressions

< relational expression > → < arithmetic expr. >
| < arithmetic expr. > < relational > < arithmetic expr. >

<relational> → `<` |`>` |`=` |`<=` |`>=` |`!=`

Relational expressions can be used to compare the values produced by two expressions. Note that the syntax of the language prohibits the expression

```
a < b < c
```

although it allows the essentially equivalent expression

```
(a < b) < c
```

8.3 Arithmetic expressions

< arithmetic expr. > → < term >
| < arithmetic expr. > < adding operator > < term >

< adding operator > → `+` |`-`

< term > → < factor >

| < term > < multiplying operator > < factor >

< multiplying operator > → * | / | %

< factor > → (< expression >)
| < adding operator > < factor >
| < variable >
| < constant >
| < function call >

C° provides the usual addition, subtraction and multiplication operations on integers. In addition, the quotient and remainder of integer division operations can be obtained by using the '/' and '%' operators respectively.

The unary '+' and '-' have the highest precedence among arithmetic operators. Multiplying operators have the next highest precedence followed by adding operators. Operators of equal precedence group from left to right.

All expressions must have type integer. In particular, this implies that any variable used as an expression must have type integer.

8.4 Constants

< constant > → < integer constant >
| < character constant >

Character constants produce values of type integer. The value of a given character constant is the integer value of the number used to represent the character in the ASCII code.

8.5 Function Calls

< function call > → < identifier > (< actual list >)
| < identifier > ( )

A function call causes the invocation of the named function at run-time. The value returned by the function is used as the value of the function call expression.

As in the call statement, the number of actual parameter expressions included in a function call must match the number of formal parameters declared in the function's definition. The type of each actual parameter expression must match the type of the corresponding formal. If any of the called function's formal parameters are specified as **var** parameters, then the corresponding actual parameters must actually be variables as defined in section 7.

A call of a function declared to return **void** can not be used as an expression.

9 Input/Output

In the proud tradition of Algol 60, the definition of C° includes no mechanisms for input or output. It is assumed that any implementation of the language will include appropriate built-in functions to provide the needed facilities.