C0 Reference

15-122: Principles of Imperative Computation
Frank Pfenning

August 2010
Compiler revision 546 (Sat Aug 28 2010)

1 Introduction

The programming language C0 is a carefully crafted subset of the C aimed at teaching introductory algorithms and imperative programming. It is restricted enough to permit a straightforward safe implementation in which attempts to access an array element out of bounds array can be reliably detected. This eliminates a class of insidious bugs that are difficult to diagnose or detect through testing, as numerous security problems with commercial software attest. As a consequence the language soundly supports a conservative garbage collector to reclaim memory, rather than relying on error-prone explicit memory management. It is intended that all operations are completed defined, although even correct programs may abort when they run out of memory. The combination of these features allow us to soundly reason about contracts and loop invariants, which constitute a new language layer. More about the differences and the transition to C in a separate document; in this document we introduce the language itself. We assume the reader has some familiarity with programming, but not necessarily with C. This document is not intended as a tutorial, but as a concise reference.

2 Types

Programming languages can be characterized by the types that they support, and C0 is no exception. We briefly list the types and explain their meaning before discussing the constructions associated with each form of type in turn.

`int` The only numeric type supported in C0 is `int`. Values of this type are 32-bit words, interpreted as integers according to two's complement representation. Computations are performed modulo 2^{32}.

`bool` The type of booleans has just two elements, `true` and `false`. They are used for conditions, as well as contracts.

`char` The type of characters contains ASCII (not Unicode) characters written as `'c'`.

`string` The type of strings contains constant strings of characters, written as `"this is a string"`.

t `[]` For any type t, we can form the type t`[]`, the type of arrays of values of type t. A value of this type is a reference to an array stored in memory. An array A has an intrinsic length n determined at the type of allocation; its elements can be referenced as $A[0]$ through $A[n-1]$.

t `*` For any type t, we can form the type t`*`, the type of pointers to values of type t stored in memory. Its values are addresses and a special value `NULL`.

`struct` s Structs are aggregates whose members can be accessed through field names. The fields and their types for each structure named s have to be explicitly declared. Structures are allocated in memory. Unlike the elements of other types, structures cannot be assigned to variables or passed as function arguments because they can have arbitrary size. Instead, we pass pointers to structs or arrays of structs.

Functions and commands There are no explicit types for functions and commands, because the language does not allow them to be passed as arguments or stored explicitly. Of course, the language has means to define and invoke functions, and execute commands including variable assignments, conditionals and loops.

Contracts Again, there is no explicit type for contracts, but C0 supports contracts governing the permissible invocations, return values, and effects of functions. Currently, they can only be checked dynamically, although some tools for static checking are under development.

3 Integers

The type of integers is `int`. The values are 32-bit words, interpreted according to two's complement arithmetic. This means arithmetic is modulo 2^{32}, with the minimal representable integer being $-2^{31} = -2147483648$ and the maximal being $2^{31} - 1 = 2147483647$. Numerical constants c in a program must be in the range $0 \leq c \leq 2^{31}$, where $2^{31} = -2^{31}$ according to modular arithmetic.

Integer operations are the usual binary `+` (addition), `-` (subtraction), `*` (multiplication), which operate modulo 2^{32}. In addition we have integer division `n/k` and modulus `n%k`. Division truncates towards zero, and both division and modulus raise an overflow exception if $k = 0$ or $n = -2^{31}$ and $k = -1$. If n is negative, the result of the modulus will be negative, so that `(n/k)*k + n%k == n` when the left-hand side doesn't overflow.

Comparisons `<`, `<=`, `>=`, `>` return a boolean when applied to integers, as do `==` (equality) and `!=` (disequality).

We can also view and manipulate values of type `int` as 32-bit words. For this purpose, we have a hexadecimal input format. A number constant in hexadecimal form starts with `0x` and contains digits `0` through `9` and `a` through `f`. Hexadecimal digits are not case sensitive, so we can also use `X` and `A` through `F`.

Binary bitwise operations on values of type `int` are `&` (and), `^` (exclusive or), `|` (or), and we also have unary bitwise complement `~`. The hybrid shift operators `n << k` and `n >> k` shift the bits of n by k. Here, k is masked to the lower 5 bits, so that the actual shift is always in the range from 0 to 31. On the left shift, the lower bits are filled with 0; on the right shift the higher bit is copied. This means that left shift by k is equal to multiplication by 2^k, and right shift k is like division by 2^k, except that it truncates towards $-\infty$ rather than 0.

The default value for integers, which is needed for some allocation operations, is 0.

The precedence and associativity of the operators is shown in Figure 3. In general, expressions are guaranteed to be evaluated from left-to-right so that, for example, in `f(x) + g(x)` we first call f and then g. Any effects such as input/output are guaranteed to happen in the specified order.

4 Booleans

The type `bool` is inhabited by the two values `true` and `false`.

Booleans can be combined with logical (as opposed to bit-wise) conjunction `&&` (and) and disjunction `||` (or), which are binary operators. Their evaluation short-circuits in the sense that in `b && c`, if b evaluates to `false`, then c is not evaluated. Similarly, in `b || c`, if b evaluates to `true`, then c is not evaluated. There is also a unary operator of logical negation `!` (not).

Booleans can be tested with a ternary operator `b ? e1 : e2` which first evaluates b. If b is `true`, it then evaluates e_1 and returns its value, otherwise it evaluates e_2 and returns its value.

Booleans can be compared for equality (==) and disequality (`!=`).

The default value is `false`.

5 Functions

Functions are not first-class in C0, but can only be declared or defined at the top-level of a file. A function definition has the form

```
t g (t1 x1, ..., tn xn) { body }
```

where t is the result type of the function called g which takes n arguments of type $t_1, \ldots, t_n$. The scope of parameters $x_1, \ldots, x_n$ is $body$, which is a block consisting of a sequence of additional local variable declarations followed by a sequence of statements. Note that function definitions are *not* terminated by a semi-colon. The scope of the function name g include $body$ and the remainder of the compilation unit, typically a file. Currently, if multiple files are given to the compiler they are concatenated sequentially into a single compilation unit.

Argument and result types must be *small*, which means that they cannot be structs. Instead of structs, programs should pass either pointers to structs or arrays containing structs.

Functions may be declared without giving a definition in the form

```
t g (t1 x1, ..., tn xn);
```

which allows the use of g in subsequent functions in the same compilation unit.

A function may be declared multiple times in a compilation unit, but must be defined exactly once. Multiple declarations must be consistent, and consistent with a possible definition, but can differ in the name of the parameters.

Library functions are special in that they may be declared in a library header file `<lib>.h0` for library `<lib>`, but they cannot be defined. Libraries can be included on the command line using the switch `-l<lib>`. See a separate description of the compiler interface.

Expressions denoting function calls have the form `g(e1,...,en)`. The arguments $e_1, \ldots, e_n$ are evaluated in sequence from left to right and the resulting values passed to the function g.

6 Commands

Commands are not first-class in C0, but occur in the bodies of functions. We have *assignments*, *conditionals*, *loops*, *blocks*, and *returns*.

6.1 Assignments

Basic assignments `x = e;` assign to x the value of the expression e. The types of x and e must match for the assignment to be legal.

More generally, the left-hand side of an assignment can be an *lvalue* which includes additional ways of referencing memory. Besides variables, the other possible lvalues are explained below for arrays (`lv[e]`), pointers (`*lv`), and structs (`lv.f`). In assignment `lv = e`, the left-hand side lv is evaluated first, then e, and then the assignment is attempted (which may fail based on the form of lv, for arrays or pointers).

There are also compound assignments of the form `lv op= e` which translate to `lv = lv op e` where *op* is a binary operator among `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `<<`, or `>>`, except that lv is evaluated only once.

Finally, there compound assignments `lv++` and `lv--` which desugar into `lv += 1` and `lv -= 1`, respectively.

6.2 Expressions as Statements

An expression e; can be used as a statement. Such a statement evaluates e, incurring all of its effects, and then discards the return value if there is any.

6.3 Conditionals

Conditionals have the form `if (e) s1 else s2`. Note that conditionals (like loops) are *not* terminated by a semi-colon. The condition e must be of type `bool`. It is evaluated first, followed either by s_1 (if e is true) or s_2 (if e is false).

There is a shorthand, `if (e) s1`, which omits the `else`-clause, which translates into `if (e) s1 else {}` where `{}` is the empty block which has no effect. The possibility to omit `else`-clauses creates an ambiguity because with two `if`s and only one `else` it may be unclear which condition the `else` is associated with. For example,

```
if (e1) if (e2) s1 else s2
```

could be read as

```
if (e1) {if (e2) s1} else s2
```

or

```
if (e1) {if (e2) s1 else s2}
```

The rule is that an `else`-clause is matched up with the most recent `if` that does not have an `else`-clause while respecting explicit grouping into blocks, so the second reading is the correct one.

6.4 Loops

There are two forms of loops.

```
while (e) s
```

begins by evaluating e. If e evaluates to `true` it continues with the execution of s, subsequently testing e again. If e evaluates to `false` we finish the `while` loop and continue with the next statement.

```
for (s1; e; s2) s3
```

begins by evaluating the loop initializer s_1 which must be an assignment (including one of its compound forms) or an expression. Then it evaluates e. If e is true, we execute the body s_3 followed by the step expression s_2 (which must again be an assignment or an expression), followed in turn by the exit test e. Both s_1 and s_3 may be omitted, in which case the act like the empty command which immediately finishes without an effect.

There are two control commands that can effect the execution of a loop.

`continue` immediately short-circuits the body of the innermost loop that contains it and proceeds with the exit test (in a `while` loop) or the step command followed by the exit test (in a `for` loop).

`break` immediately exits the innermost loop that contains it and proceeds with the statement following the loop.

6.5 Blocks

Blocks have the form `{ds ss}`, where ds is a (possibly empty) sequence of variable declarations and ss is a (possibly empty) sequence of statements. First, the declarations are processed in order and then the statements in ss are executed in order.

A declaration has one of the two forms

```
t x;
```

where t is a type and x is a variable, or

```
t x = e;
```

where t is a type, x is a variable, and e is an expression initializing x which must have type t.

In either form, the scope of x consists of the remaining declarations and statements in the block containing the declaration.

Variables declared in an outer scope (either as function parameters of an enclosing block) can not be declared again in an inner block with the same name.

6.6 Returns

Anywhere in the body of a function there can be a `return` statement, either of the form `return e;` for an expression e or just `return;`.

In the form `return e;`, the type of e must match the result type of the function. In the form `return;`, the result type of the function must be the special type `void` that can only be used to indicate that a function does not return a value. Such functions can only be called as expressions that appear as statements.

7 Characters

Characters are a special type to represent components of strings. They are input in the form `'c'`, where c can be any printable ASCII character, as well as the following escape sequences `\t` (tab), `\r` (return), `\f` (formfeed), `\a` (alert), `\b` (backspace), `\n` (newline), `\v` (vertical tab), `\'` (quote), `\0` (null). The default value for characters is `\0`.

8 Strings

Strings have the form `"c1...cn"`, where $c_1, \ldots, c_n$ is an ASCII character as above, including the legal escape sequences except for null, which may not appear in strings. The double-quote character itself `"` must be quoted as `\"` so it is not interpreted as the end of the string. The default value for type `string` is the empty string `""`.

9 Arrays

The type of arrays with elements of type t is denoted by `t []`. Arrays must be explicitly allocated in memory, but they do not need to be deallocated, a function performed by the garbage collector. For this purpose we have a new expression

```
alloc_array(t, e)
```

which returns a reference to a new array of type `t []`. The expression e must evaluate to a non-negative integer n denoting the length of the allocated array. Elements of an array A allocated in this way are accessed as `A[0]` through `A[n-1]`. Attempt to index an array with a negative number or a number greater or equal to n will result in an array bounds violation that will terminate the program.

Array references can also be used as lvalues. For example, `A[0] = 1` will write 1 to the first element of A which must be an integer array, and `A[2]++` will increment the third element in the array.

For every type t there is a distinguished zero-sized array of type `t []` which serves as the default. Because its size is zero, the only operations that can be performed on this element are comparisons for equality (`==`) and disequality (`!=`).

It is important to remember that comparisons of variables of type `t []` are comparisons of array references, and not the array elements, and similarly for argument passing and variable assignment.

Because of its roots in C, one cannot determine the length of arrays in programs. This allows an unsafe implementation in which array bounds of accesses are not checked, a low-level efficiency improvement that can make a significant difference in certain kinds of highly numerical code. On the other hand, contracts must be able to mention the length of arrays to ensure the absence of runtime errors. For that purpose there is a special function `\length(e)` that can only be used in contracts. When contracts are to be checked dynamically, the compiler will take care to represent arrays such that the length is stored.

10 Pointers

The type of pointers of type t is denoted by `t*`. We obtain a pointer to a memory location holding a value of type t using the new expression

```
alloc(t)
```

We dereference pointers using the expression `*e` which evaluates to a value of type t if e evaluates to a pointer of type `t*`.

Pointers also introduce a new lvalue `*lv` which references the memory location or variable denoted by `lv`.

For each type `t` there is a special pointer `NULL` of type `t*`. Attempts to dereference `NULL` will result in a runtime exception that terminates the program. `NULL` is the default value for pointers at each type.

The constant `NULL` introduces a type ambiguity which can be locally resolved in expressions where relevant. For example, a function call `f(NULL)` is well-typed if `f` expects an argument of type `int*` or `bool*` or `int[]*`, etc. The one exception to this rule is code of the form `*NULL` which could be used at an arbitary type t. In order to avoid this typical ambiguity, it is an error to write `*NULL` in programs.

11 Structs

Structs are the only types that can aggregate values of different type. We write `struct s` for the type of structs named `s`. Structure names occupy their own name space, as do the names of the fields of structs; neither can conflict with names of variables, functions, or other fields or struct names. Structs with name s are defined with

```
struct s { t1 f1; ... tn fn; };
```

and have fields named $f_1, \ldots, f_n$ of types $t_1, \ldots, t_n$, respectively. After such a declaration, the field f_i of a struct denoted by e of type `struct s` can be accessed with the expression `e.fi` and has type t_i.

Structs must be allocated in memory. Because they may be of large size, their value can not be held in variable or passed as an argument to a function, or returned from a function. We therefore call `struct s` a *large* type, while all other types in the language are *small*. In particular, array types `t[]` are small because a value of this type is a reference to an array, not the array itself. In contrast, a value of type `struct s` is the struct itself. This means that programs mostly manipulate either pointers to structs `struct s*` or arrays of structs `struct s[]`. As a result there is no special form to allocate a struct: structs will be allocated as the result of allocating memory with `alloc(struct s)` or `alloc_array(struct s, e)` or other data types with embedded structs. Each of the fields of a struct allocated in this way is initialized with default values according to their type.

Because pointers to structs are common, there are two constructs supporting the idiomatic use of pointers to structs. The first is the expression `e->f` which stands for `(*e).f`.

The second is a general form of type definition written as

```
typedef t a
```

where t is a type and a is a type name. This definition can appear only at the top-level and the scope of the type name a is the rest of the compilation unit. In order avoid certain ambiguities in the grammar, type names a occupy the same name space as variables and functions. It is a conflict to declare or define a function or a variable with the same name as a type.

The idiomatic, but not the only use of the above, has the form

```
typedef struct s* s
```

after which the type name s represents pointers to `struct s`.

Struct types `struct s` can be used before they are defined, but they can also be explicitly declared as

```
struct s;
```

Since the fields of such a struct are not known, they cannot be accessed by a program. Nonetheless, pointers to elements of such type have a uniform size and can therefore be passed as arguments even without knowing the precise representation of s. This allows a very weak form of polymorphism in C0.

12 Contracts

Contracts collectively refer to assertions made about the code. Contracts are never necessary to execute the code, but it is possible to check the adherence to contracts dynamically by compiling the code with a special flag.

Contracts specify either pre- and post-conditions for functions, loop invariants, or preconditions for statements.

From the syntactical point of view, contracts appear as special comments or *annotations* that can be ignored by a compiler that does not support them. As such, they constitute a separate language layer which is entirely absent from C. Annotations start with `//@` and extend to the end of the line, or delimited by `/*@` and `@*/`. For illustration purposes below we use the single-line form.

Contracts should never have store effects and should terminate, although the compiler currently does not check that. It is permissible for contracts to raise exceptions, including the exception that the contract was not satisfied. **Disclaimer:** this section of the language specification is still somewhat in flux, so your implementation may differ in small ways.

12.1 Function Contracts

For functions to work correctly, they often impose conditions on their input. For example, an integer square root may require its argument to be non-negative, or a dictionary insertion function may require the dictionary to be sorted. Conversely, it will make some guarantees on its outputs. For example, the integer square root should really return the root and perhaps more specifically the positive one, and the insertion function should really insert the new word into the dictionary into its proper place. The former is called a *precondition* for the function, specified with `@requires`; the latter is a *postcondition* specified with `@ensures`.

A function definition than has the general form

```
t g (t1 x1, ..., tn xn)
contracts
{ body }
```

where a contract is one of the following

```
//@requires e;
//@ensures e;
```

The expression e, which must have type `bool` can mention the parameters $x_1, \ldots, x_n$. It can also mention the special function `\length(e)` mentioned above, the special variable `\result` in `@ensures` clauses. Finally, contracts have the special expression `\old(e)` in `@ensures` clauses which refers to the value of the expression e at the time of the function invocation.

Contracts must be in single-line or multi-line comments, as in the following example.[1]

```
int exp (int k, int n)
//@requires n >= 0;
//@ensures \result >= 1;
/*@ensures \result > n; @*/
{ int res = 1; int i = 0;
  while (i < n) {
    res = res * k;
    i = i + 1;
  }
  return res;
}
```

When dynamic checking of contracts is enabled, `@requires e;` specifications are checked just before the function body and `@ensures e;` are checked just before the return, with the special variable `\result` bound to the return value. Any subexpressions `\old(e)` in `@ensures` clauses are computed after the `@requires` annotations but before the function body and bound to unique new variables.

12.2 Loop Invariants

Loop invariant annotations have the form

```
//@loop_invariant e;
```

where e has type `bool`. The general form of `while` and `for` loops is

```
while (e) invs s;
for (s1; e; s2) invs s;
```

where `invs` is a possibly empty sequence of invariants. As for function contracts, they must be stylized single-line or delimited comments.

When dynamic checking is enabled, the loop invariant is checked on every iteration just before the exit condition e is evaluated and tested.

[1] For modular arithmetic as specified for C0, this contract is *not* satisfied because the result may be negative.

12.3 Assertions

There are two forms of assertions: statements and annotations. An assertion as a statement is simply written as

```
assert(e1, e2);
```

where e_1 is a boolean test and e_2 is a string (the message that is printed). Assertion statements are *always* executed, no matter whether contracts are checked dynamically.

Assertion annotations have the form

```
//@assert e;
```

An assertion annotation must precede another statement and can be seen as guard on that statement. When a function is called correctly, according to its precondition (`//@requires`), the assert annotations should not fail; in that sense they express expected internal invariants of functions, just like loop invariants.

13 Grammar

We now summarize the grammar rules of the language.

13.1 Lexical tokens

We have the following classes of tokens: identifiers, numerical constants, string literals, character literals, operators, and reserved keywords. In addition there is whitespace, which is a regular space, horizontal and vertical tab, newline, formfeed and comments. Whitespace separates tokens, but is otherwise ignored. Other control (non-printing) characters in the input constitute an error.

Comments may be on a single line, starting with `//` and ending with newline, or delimited, starting with `/*` and ending with `*/`. Delimited comments must be properly nested. When annotations are parsed and checked, the first character of a comment must not be `@`, which would start an annotations.

We present the token classes as regular expressions. [Square brackets] surround enumerations of single characters or character ranges like `a-z`, <angle brackets> surround nonterminals in the grammar.

The reserved keywords of the language are:

```
<id>        ::= [A-Za-z_][A-Za-z0-9_]*

<num>       ::= <decnum> | <hexnum>
<decnum>    ::= 0 | [1-9][0-9]*
<hexnum>    ::= 0[xX][0-9a-fA-F]+

<strlit>    ::= "<schar>*"
<chrlit>    ::= ’<cchar>’
<schar>     ::= <nchar> | <esc>
<cchar>     ::= <nchar> | <esc> | " | \0
<nchar>     ::= (normal printing character except ")
<esc>       ::= \n | \t | \v | \b | \r | \f | \a
              | \\ | \? | \’ | \"

<unop>      ::= ! | ~ | - | *
<binop>     ::= + | - | * | / | % | << | >>
              | < | > | == | != | & | ^ | | | && | ||
<asnop>     ::= = | += | -= | *= | /= | %= | <<= | >>=
              | &= | ^= | |=
```

Figure 1: C0 lexical tokens

```
int bool string char struct typedef
if else while for continue break return assert
true false NULL alloc alloc_array
```

13.2 Grammar

We present the grammar in a similarly abbreviated style in Figure 2. Here, [brackets] surround optional constituents. Identifers occupy four name spaces: variables and function names `<vid>`, type names `<aid>`, struct names `<sid>` and field names `<fid>`. Variable and function names may not conflict with type names; otherwise the same identifiers can be reused.

13.3 Annotations

Annotations may be on a single line, starting with `//@` and ending with newline, or delimited, starting with `/*@` and ending with `@*/`. In an annotation, the `@` character is ignored.

The additional reserved keywords are

```
requires ensures loop_invariant \result \length \old
```

```
<prog> ::= (<gdecl> | <gdefn>)*

<gdecl> ::= struct <vid> ;
          | <tp> <vid> ( [<tp> <vid> (, <tp> <vid>)*] ) ;

<gdefn> ::= struct <sid> { (<tp> <fid> ;)* } ;
          | <tp> <vid> ( [<tp> <vid> (, <tp> <vid>)*] ) { <body> }
          | typedef <tp> <aid> ;

<body> ::= <decl>* <stmt>*

<decl> ::= <tp> <vid> [= <exp>] ;

<stmt> ::= <simple> ;
         | if ( <exp> ) <stmt> [ else <stmt> ]
         | while ( <exp> ) <stmt>
         | for ( [simple] ; <exp> ; [simple] ) <stmt>
         | continue ;
         | break ;
         | return [<exp>] ;
         | { <body> }
         | assert ( <exp> , <exp> ) ;
         | ;

<simple> ::= <lv> <asnop> <exp>
           | <lv> ++
           | <lv> --
           | <exp>

<lv> ::= <vid> | <lv> . <fid> | <lv> -> <fid>
       | * <lv> | <lv> [ <exp> ]

<tp> ::= int | bool | string | char
       | <tp> * | <tp> [ ] | struct <sid> | <aid>

<exp> ::= ( <exp> )
        | <num> | <strlit> | <chrlit> | true | false | NULL
        | <vid> | <exp> <binop> <exp> | <unop> <exp>
        | <exp> ? <exp> : <exp>
        | <vid> ( [<exp> (, <exp>)*] )
        | <exp> . <fid> | <exp> -> <fid>
        | <exp> [ <exp> ] | * <exp>
        | alloc ( <tp> ) | alloc_array ( <tp> , <exp> )
```

Figure 2: C0 Grammar, without annotations

<table>
<tr><th>Operator</th><th>Associates</th><th>Meaning</th></tr>
<tr><td>() [] -> .</td><td>left</td><td>parens, array subscript, field dereference, field select</td></tr>
<tr><td>! ~ - * ++ --</td><td>right</td><td>logical not, bitwise not, unary minus, pointer dereference increment, decrement</td></tr>
<tr><td>* / %</td><td>left</td><td>integer times, divide, modulo</td></tr>
<tr><td>+ -</td><td>left</td><td>plus, minus</td></tr>
<tr><td><< >></td><td>left</td><td>(arithmetic) shift left, right</td></tr>
<tr><td>< <= >= ></td><td>left</td><td>comparison</td></tr>
<tr><td>== !=</td><td>left</td><td>equality, disequality</td></tr>
<tr><td>&</td><td>left</td><td>bitwise and</td></tr>
<tr><td>^</td><td>left</td><td>bitwise exclusive or</td></tr>
<tr><td>|</td><td>left</td><td>bitwise or</td></tr>
<tr><td>&&</td><td>left</td><td>logical and</td></tr>
<tr><td>||</td><td>left</td><td>logical or</td></tr>
<tr><td>? :</td><td>right</td><td>conditional expression</td></tr>
<tr><td>= += -= *= /= %=
&= ^= |= <<= >>=</td><td>right</td><td>assignment operators</td></tr>
</table>

Figure 3: Operator precedence, from highest to lowest

```
<spec> ::= requires <exp> ;
         | ensures <exp> ;
         | loop_invariant <exp> ;
         | assert <exp> ;

<anno> ::= //@ <spec>* \n
         | /*@ <spec>* @*/

<gdecl> ::= ...
          | <tp> <vid> ( [<tp> <vid> (, <tp> <vid>)*] ) <anno>* ;

<gdefn> ::= ...
          | <tp> <vid> ( [<tp> <vid> (, <tp> <vid>)*] ) <anno>*
            { <body> }

<stmt> ::= ... | <anno>+ stmt

<exp> ::= ... | \result | \length ( <exp> ) | \old ( <exp> )
```

Figure 4: C0 grammar extensions for annotations

The grammar is modified by adding the following cases. The restrictions on annotations are detailed in Section 12.

This extension introduces another ambiguity, because a statment of the form `<anno> <anno> <stmt>` could be one statement with two annotations, or an annotated annotated statement. We resolve this by always interpreting it as a single statement with two annotations, or multiple annotations in the general case.

14 Static Semantics Reference

The static semantics enforces the following conditions.

- All operators and functions are used with the correct number of arguments of the correct type, as explained in the sections on the various language constructs.
- Operators <, <=, >=, and > are overloaded in that they apply to type `int` and `char`. Both sides must have the same type.
- Operators == and `!=` are overloaded in that they apply to types `int`,

bool, char, t [], and t *. They do not apply to arguments of type string and struct s. Both sides must have the same type.

- Structs cannot be passed to or from functions or assigned to variables.
- All control-flow paths in the body of each function end with a return statement of the correct type, unless the function has result type void.
- Every variable must be declared with its type.
- Along each control-flow path in the body of each block in each function, each locally declared variable is initialized before its use.
- Function parameters and locally declared variables with overlaping scopes may not have the same name.
- Names of functions or variables may not collide with the names of defined types.
- Functions may be declared multiple times with consistent types, but must be defined exactly once unless they are library functions, in which case the may not be defined. Structs may be declared multiple times, but may be defined at most once. Structs declared in libraries cannot be defined. Type names may be defined only once (they cannot be declared).
- A function int main(); is implicitly declared and also implicitly used, because this is the function called when an executable resulting from compilation is invoked. Therefore, when a collection of sources is compiled, at least one of them must define main to match the above prototype.
- Field names within each struct must be pairwise distinct.
- Expressions *NULL are disallowed.
- Type void is used only as the return type of functions.
- Undefined structs cannot be allocated.
- continue and break statements can only be used inside loops.
- Integer constants are in the range from 0 to 2^{31}.
- * <lv> ++ and * <lv> -- must be be explicitly parenthesized to override the right-to-left associative interpretation of ++ and --.

In addition we check in annotations:

- `\result` is only legal in `@ensures` clauses.
- `\old` is only legal in `@ensures` clauses.
- `@requires` and `@ensures` can only annotate functions.
- `@loop_invariant` can only precede loop bodies.
- `@assert` can not annotate functions
- Expressions occurring in function annotations can only refer to the functions parameters. Expressions in loop invariants and assertions can also use other local variables in whose scope they occur.