Daniel Benamy - Nicolas Viennot

COMS W4115 - C Flat Reference Manual

1 Introduction

The C Flat language is mostly a subset of the C language. Some of the core functionalities of C has been stripped: there is no preprocessor, no structs, no strings, not even pointers. it's goal is purely educational. Originally Nico and Dan were working on two separate languages. The two projects merged, taking some features from each, and this is the resulting language. This document is inspired by the C Reference Manuel by Dennis Ritchie.

2 Lexical conventions

2.1 Whitespace

A tab, a space or a new line is a whitespace. At least one of these charaters is required to separate adjacent identifiers, constants, and certain oparator-pairs.

2.2 Comments

There are two ways to place comments: `//` introduces a comment which ends with a end of line. `/*` also introduces a comment which ends with `*/`, they can be nested. A `//` inside a `/* */` comment is ignored.

2.3 Identifiers

An identifier is a sequence of letters and digits. The first charater must be alphabetic. The underscore counts as alphabetic. An identifier is case sensitive.

2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
return break continue if else for while try catch throw
```

2.5 Constants

There is only one kind of constant: a 32 bits signed integer. Such a constant is a sequence of digits represented in its decimal form.

3 Expressions

An expression evaluates to a 32 bits signed integer. The precedence of operators is described in the syntax summary.

3.1 identifier

An identifier evaluates to the value of the corresponding variable.

3.2 literal

A decimal number is an expression.

3.3 (expression)

A parenthesized expression evaluates to the parenthesized expression.

3.4 identifier (expression-list$_{\text{opt}}$)

A function call is an expression. The arguments are optional and separated with a comma. They are evaluated from left to right before the call (applicative order). The value returned by the function is the value the callee returns with a `return` statement.

3.5 -expression

The result is the negative of the expression.

3.6 +expression

The result is the expression itself.

3.7 !expression

The result of the logical negation operator `!` is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero.

3.8 ~expression

The ~ operator yields the one's complement of its operand.

3.9 identifier++

The referred variable is incremented when evaluated. The expression evalutaes to the value of the variable before the increment.
Note that the statement "`{a = 0; b = 0; b = a++ + a++; }`" sets the value of `a` to 2, and `b` to 1.

3.10 identifier--

The referred variable is decremented when evaluated. The expression evaluates to the value of the variable before the decrement.

3.11 ++identifier

The referred variable is incremented when evaluated. The expression evaluates to the value of the variable after the increment.

3.12 --identifier

The referred variable is decremented when evaluated. The expression evaluates to the value of the variable after the decrement.

3.13 expression * expression

The binary `*` operator indicates multiplication.

3.14 expression / expression

The binary /operator indicates division.

3.15 expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. The remainder has the same sign as the dividend.

3.16 expression + expression

The result is the sum of the expressions.

3.17 expression - expression

The result is the difference of the expressions.

3.18 expression >> expression
expression << expression

The value of the right hand side operand should be non-negative and less than 32, if not the result is undefined.
The value of "E1 >> E2" is E1 arithmetically right-shifted by E2 bit positions. Vacated bits are filled by a copy of the sign bit of the first expression.
The value of "E1 << E2" is R1 left-shifted by E2 bit positions. Vacated bits are 0-filled.

3.19 expression < expression
expression > expression
expression <= expression
expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

3.20 expression == expression
expression != expression

The operators == (equal to) and the != (not equal to) yield 0 if the specified relation is false, 1 if it is true.

3.21 expression & expression

The & operator yield the bitwise `and` function of the operands.

3.22 expression ^ expression

The & operator yield the bitwise exclusive `or` function of the operands.

3.23 expression | expression

The | operator yield the bitwise inclusive `or` function of the operands.

3.24 expression && expression

The && operator returns 1 if both operands are non-zero, 0 otherwise. Both operands are always evaluated.

3.25 expression || expression

The || operator returns 1 if either of its operands is non-zero, 0 otherwise. Both operands are always evaluated.

3.26 identifier = expression

The value of the referred variable is replaced by the value of the expression.

3.27 identifier += expression
identifier -= expression
identifier *= expression
identifier /= expression
identifier %= expression
identifier >>= expression
identifier <<= expression
identifier &= expression
identifier ^= expression
identifier |= expression

An expression of the form "id op= expr" is equivalent to "id = id op expr".

4 Statements

Statements are executed in sequence.

4.1 Expression statement

Most statement are expression statements, which have the form
expression ;

4.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

compound-statement:

{ statement-list$_{opt}$ }

statement-list:

statement

statement statement-list

4.3 Conditional statement

The two forms of the conditional statement are

if (expression) statement

if (expression) statement else statement

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an else with the last encountered elseless if.

4.4 While statement

The `while` statement has the form

`while (` expression `)` statement

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

4.5 For statement

The `for` statement has the form

`for (` expression-1_{opt} `;` expression-2_{opt} `;` expression-3_{opt} `)` statement

This statement is equivalent to

```
expression-1;
while ( expression-2 ) {
    statement
    expression-3;
}
```

Any or all the expression may be dropped. A missing expression-2 makes the implied `while` clause equivalent to "while(1)". Other missing expressions are simply dropped from the expansion above.

4.6 Break statement

The statement

`break;`

casuses termination of the smallest enclosing `while` or `for` statement; control passes to the statement following the terminated statement.

4.7 Continue statement

The statement

`continue;`

causes control to pass to the loop-continuation portion of the smallest enclosing `while` or `for` statement; that is to the end of the loop. In case of a `for` loop of the form "`for(e1;e2;e3) {...}`", `e3` is evaluated before checking `e2`.

4.8 Return statement

A function returns to its caller by means of the `return` statement

`return` expression `;`

The value of the expression is returned to the caller of the function.

4.9 Null statement

The null statement has the form

`;`

A null statement is useful to supply a null body to a looping statement such as `while`.

4.10 Try-catch statement

The two form of the try-catch statement are

`try {` statement-list$_{\text{opt}}$ `}catch ( identifier ) {` statement-list$_{\text{opt}}$ `}`

`try {` statement-list$_{\text{opt}}$ `}catch {` statement-list$_{\text{opt}}$ `}`

The statments enclosed in the try block are executed until an exception is thrown. In case no exception is thrown, the statments enclosed in the catch block are not executed. The first form of the try-catch statement allows to assign the value of the exception to a variable. Try-catch statement dynamically nest across function calls.

4.11 Throw statement

The throw statement has the form

`throw` expression `;`

Throwing an exception causes control to pass to the catch block of the nearest dynamically-enclosing try-catch statement. If none is found, it causes the program to terminate with an error. The given expression is the value of the thrown exception.

5 Program definition

A ltc program consists of a sequence of function definition.

program:

function-definition

function-definition program

function-definition:

identifier `(` parameter-list$_{\text{opt}}$ `) {` statement-list$_{\text{opt}}$ `}`

parameter-list:

identifier

identifier `,` parameter-list

the same identifier cannot be used more than once in the parameter list. Within the same program, A function cannot be defined twice (name wise).
All functions return a integer value. A function can return to the caller without an explicit `return` statement, in this case the return value is undefined.

A simple example of a complete function definition:

```
max (a, b, c) {
  if (a > b) m = a; else m = b;
  if (m > c) return m; else return c;
}
```

6 Scope rules

There are no global variables, but only local variables which are statically binded. The scope of a local variable is the whole function where the variable is used. The scope of function parameters is the whole function.

Function scope is the entire program.

7 Declarations

Variables don't need to be declared, they are initialized to 0.
A function call can be made whether or not the function actually exists, the program will simply not link if a call to a non-existing function is made.

8 Namespace

Variables and function use different namespaces. Therefore such a function is correct: "`f() {f=1; return f; }`".

9 Syntax Summary

9.1 Expressions

expression:

identifier
literal
`(` expression `)`
identifier `(` expression-list$_{\text{opt}}$ `)`
`-`expression
`+`expression
`!`expression
∼expression
`++`identifier
`--`identifier
identifier`++`
identifier`--`
expression binop expression
identifier asgnop expression

expression-list:

expression
expression `,` expression-list

The unary operators `- + !` ∼ have higher priority than binary operator.
Binary operators all group left to right and have priority decreasing as indicated:

binop:

`* / %`

`+ -`

`>> <<`

`< > <= >=`

`== !=`

`&`

$\wedge$

`|`

`&&`

`||`

Assignment operator all have the same priority, and all group right to left.

asgnop:

`= += -= *= /= %= >>= <<= &= ` $\wedge$ `= |=`

9.2 Statements

statement:

expression `;`

`{` statement-list$_{\text{opt}}$ `}`

`if (` expression `)` statement

`if (` expression `)` statement `else` statement

`while (` expression `)` statement

`for (` expression$_{\text{opt}}$ `;` expression$_{\text{opt}}$ `;` expression$_{\text{opt}}$ `)` statement

`break;`

`continue;`

`return` expression`;`

`try {` statement-list$_{\text{opt}}$ `}catch {` statement-list$_{\text{opt}}$ `}`

`try {` statement-list$_{\text{opt}}$ `}catch (` identifier `) {` statement-list$_{\text{opt}}$ `}`

`throw` expression`;`

`;`

statement-list:

statement

statement statement-list

9.3 Program definition

program:

function-definition
function-definition program

function-definition:

identifier `(` parameter-list$_{\text{opt}}$ `)` `{` statement-list$_{\text{opt}}$ `}`

parameter-list:

identifier
identifier `,` parameter-list