Crowbar Specification

Release e82766f specify requirements more conventionally

boringcactus (Melody Horn)

Dec 23, 2020

Table of Contents

Crowbar: the good parts of C, with a little bit extra.

This is entirely a work-in-progress, and should not be relied upon to be stable (or even accurate) in any way.

Crowbar is a language that is derived from (and, wherever possible, interoperable with) C, and aims to remove as many footguns and as much needless complexity from C as possible while still being familiar to C developers.

Ideally, a typical C codebase should be straightforward to rewrite in Crowbar, and any atypical C constructions not supported by Crowbar can be left as C.

This site hosts the Crowbar specification at https://crowbar-lang.org and at gemini://crowbar-lang.org. Additional resources you may be interested in:

- sr.ht project hub
- specification PDF
- specification EPUB
- reference compiler

Chapter 1

Motivation

- Rust is not a good C replacement

Chapter 2

Journal

- Crowbar: Defining a good C replacement
- Crowbar: Simplifying C's type names
- Crowbar: Turns out, language development is hard

Chapter 3

Discuss

- announcement mailing list
- permanent discussion mailing list
- ephemeral discussions via IRC: #crowbar-lang on freenode (join via irc, join via web)

Chapters

4.1 Comparison to C

What differentiates Crowbar from C?

4.1.1 Removals

Some of the footguns and complexity in C come from misfeatures that can simply not be used.

Footguns

Some constructs in C are almost always the wrong thing.

- `goto`
- Wide characters
- Digraphs
- Prefix `++` and `--`
- Chaining mixed left and right shifts (e.g. `x << 3 >> 2`)
- Chaining relational/equality operators (e.g. `3 < x == 2`)
- Mixed chains of bitwise or logical operators (e.g. `2 & x && 4 ^ y`)
- Subtly variable-size integer types (`int` instead of `int32_t`, etc)
- The comma operator `,`

Some constructs in C exhibit implicit behavior that should instead be made explicit.

- `typedef`
- Octal escape sequences
- Using an assignment operator (`=`, `+=`, etc) or (postfix) `++` and `--` as components in a larger expression
- The conditional operator `?:`
- Preprocessor macros (but constants are fine)

Needless Complexity

Some type modifiers in C exist solely for the purpose of enabling optimizations which most compilers can

do already.

- `inline`
- `register`

Some type modifiers in C only apply in very specific circumstances and so aren't important.

- `restrict`
- `volatile`
- `_Imaginary`

4.1.2 Adjustments

Some C features are footguns by default, so Crowbar ensures that they are only used correctly.

- Unions are not robust by default. Crowbar offers two types of union declarations: a *`tagged union`* (the default) and a *`fragile union`* for interoperability purposes.

C's syntax isn't perfect, but it's usually pretty good. However, sometimes it just sucks, and in those cases Crowbar makes changes.

- C's variable declaration syntax is far from intuitive in nontrivial cases (function pointers, pointer-to-`const` vs `const`-pointer, etc). Crowbar uses *`simplified type syntax`* to keep types and variable names distinct.
- `_Bool` is just `bool`, `_Complex` is just `complex` (why drag the preprocessor into it?)
- Adding a _ to numeric literals as a separator
- All string literals, char literals, etc are UTF-8
- Octal literals have a `0o` prefix (never `0O` because that looks nasty)

4.1.3 Additions

Anti-Footguns

- C is generous with memory in ways that are unreliable by default. Crowbar adds *memory safety conventions* to make correctness the default behavior.
- C's conventions for error handling are unreliable by default. Crowbar adds *error propagation* to make correctness the default behavior.

Trivial Room For Improvement

- Binary literals, prefixed with `0b`/`0B`

4.2 Memory Safety

In general, Crowbar does its best to ensure that code will not exhibit any of the following memory errors (pulled from Wikipedia's list of memory errors. However, sometimes the compiler knows less than the programmer, and so code that looks dangerous is actually fine. Crowbar allows programmers to suspend the memory safety checks with the *fragile* keyword.

4.2.1 Access errors

Buffer overflow

Crowbar addresses buffer overflow with bounds checking. In C, the type `uint8_t *` can point to a single byte, a null-terminated string of unknown length, a buffer of fixed size, or nothing at all. In Crowbar, the type `uint8 *` can only point to either a single byte or nothing at all. If a buffer is declared as `uint8[50] name;` then it has type `uint8[50]`, and can be implicitly converted to `(uint8[50])*`, a pointer-to-50-bytes. If memory is dynamically allocated, it works as follows:

```
void process(uintsize bufferSize, uint8[bufferSize] buffer) {
    // do some work with buffer, given that we know its size
}

int8 main(uintsize argc, (uint8[1024?])[argc] argv) {
    uintsize bufferSize = getBufferSize();
    (uint8[bufferSize])* buffer = malloc(bufferSize);
    process(bufferSize, buffer);
    free(buffer);
}
```

Note that `malloc` as part of the Crowbar standard library has signature `(void[size])* malloc(uintsize size);` and so no cast is needed above. In C, `buffer` in `main` would have type pointer-to-VLA-of-char, but `buffer` in `process` would have type VLA-of-char, and this conversion would emit a compiler warning. However, in Crowbar, a `(T[N])*` is always implicitly convertible to `T[N]`, so no warning exists.

Note as well that the type of `argv` is complicated. This is because the elements of `argv` have unconstrained size.

Todo

figure out if that's the right way to handle that

Buffer over-read

bounds checking again

Race condition

uhhhhh idk

Page fault

bounds checking, dubious-pointer checking

Use after free

`free(&x);` will set `x = NULL;` `owned` and `borrowed` keywords

4.2.2 Uninitialized variables

forbid them in syntax

Null pointer dereference

dubious-pointer checking

Wild pointers

dubious-pointer checking

4.2.3 Memory leak

Stack exhaustion

uhhhhhh idk

Heap exhaustion

that counts as error handling, just the *malloc*-shaped kind

Double free

this is just use-after-free but the use is calling free on it

Invalid free

don't do that

Mismatched free

how does that even happen

Unwanted aliasing

uhhh don't do that?

4.3 Error Handling

TODO

4.4 Language

The syntax of Crowbar is designed to be similar to the syntax of C.

A Crowbar source file is UTF-8. Unless otherwise specified, a *character* in this specification refers to a Unicode scalar value. Crowbar source files can come in two varieties:

A Crowbar source file is read into memory in two phases: *scanning* (which converts text into an unstructured sequence of tokens) and *parsing* (which converts an unstructured sequence of tokens into a parse tree).

Syntax elements in this document are given in the form of parsing expression grammar rules.

4.4.1 Scanning

token
: A single atomic unit in a Crowbar source file. May be a *keyword*, an *identifier*, a *constant*, a *string literal*, or a *punctuator*. Keywords, identifiers, and constants (except for *character constant*s) must have either whitespace or a comment separating them. Punctuators, string literals, and character constants do not require explicit separation from adjacent tokens.

keyword
: One of the literal words `bool`, `break`, `case`, `const`, `continue`, `default`, `do`, `else`, `enum`, `false`,

`float32`, `float64`, `for`, `fragile`, `function`, `if`, *`include`*, `int8`, `int16`, `int32`, `int64`, `intaddr`, `intmax`, `intsize`, `opaque`, `return`, `sizeof`, `struct`, `switch`, `true`, `uint8`, `uint16`, `uint32`, `uint64`, `uintaddr`, `uintmax`, `uintsize`, `union`, `void`, or `while`.

identifier
A nonempty sequence of characters blah blah blah

Todo

figure out https://www.unicode.org/reports/tr31/tr31-33.html

---

constant
A numeric (or numeric-equivalent) value specified directly within the code. May be a *decimal constant*, a *binary constant* , an *octal constant*, a *hexadecimal constant*, a *floating-point constant*, a *hexadecimal floating-point constant*, or a *character constant*. Any of these except for the character constant may contain underscores; these are ignored by the compiler and only meaningful to humans reading the code.

decimal constant
A sequence of characters matching the regular expression `[0-9_]+`. Denotes the numeric value of the given sequence of decimal digits.

binary constant
A sequence of characters matching the regular expression `0[bB][01_]+`. Denotes the numeric value of the given sequence of binary digits (after the `0[bB]` prefix has been removed).

octal constant
A sequence of characters matching the regular expression `0o[0-7_]+`. Denotes the numeric value of the given sequence of octal digits (after the `0o` prefix has been removed).

hexadecimal constant
A sequence of characters matching the regular expression `0[xX][0-9a-fA-F]+`. Denotes the numeric value of the given sequence of hexadecimal digits (after the `0[xX]` prefix has been removed).

floating-point constant
A sequence of characters matching the regular expression `[0-9_]+\.[0-9_]+([eE][+-]?[0-9_]+)?`.

Note Unlike in C and many other languages, `6e3` in Crowbar is not a valid floating-point constant. The Crowbar-compatible spelling is `6.0e3`.

Denotes the numeric value of the given decimal number, optionally expressed in scientific notation. That is, `XeY` denotes `X * 10^Y`.

hexadecimal floating-point constant
A sequence of characters matching the regular expression `0(fx|FX)[0-9a-fA-F_]+\.[0-9a-fA-F_]+[pP][+-]?[0-9_]+`. Denotes the numeric value of the given hexadecimal number expressed in binary scientific notation. That is, `XpY` denotes `X * 2^Y`.

character constant
A pair of single quotes `'` surrounding either a single character or an *escape sequence*. The single character may not be a single quote or a backslash `\`. Denotes the Unicode scalar value for either the single surrounded character or the character denoted by the escape sequence.

escape sequence
One of the following pairs of characters:

- `\'`, denoting the single quote `'`
- `\"`, denoting the double quote `"`
- `\\`, denoting the backslash `\`
- `\r`, denoting the carriage return (U+000D)
- `\n`, denoting the line feed, or newline (U+000A)
- `\t`, denoting the (horizontal) tab (U+0009)
- `\0`, denoting a null character (U+0000)

Or a sequence of characters matching one of the following regular expressions:

- `\\x[0-9a-fA-F]{2}`, denoting the numeric value of the given two hexadecimal digits
- `\\u[0-9a-fA-F]{4}`, denoting the numeric value of the given four hexadecimal digits
- `\\U[0-9a-fA-F]{8}`, denoting the numeric value of the given eight hexadecimal digits

string literal
A pair of double quotes `"` surrounding a sequence whose elements are either single characters or escape sequences. No single-character element may be the double quote or the backslash. Denotes the UTF-8-encoded sequence of bytes representing the sequence of characters which, either directly or via an escape sequence, are specified between the quotes.

punctuator
One of the literal sequences of characters `[`, `]`, `(`, `)`, `{`, `}`, `.`, `,`, `+`, `-`, `*`, `/`, `%`, `;`, `:`, `!`, `&`, `|`, `^`, `~`, `>`, `<`, `=`, `->`, `++`, `--`, `>>`, `<<`, `<=`, `>=`, `==`, `!=`, `&&`, `||`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, or `^=`.

whitespace
A nonempty sequence of characters that each has a Unicode general category of either Control (`Cc`) or Separator (`Z`). Separates tokens.

comment
Text that the compiler should ignore. May be a *line comment* or a *block comment*.

line comment
A sequence of characters beginning with the characters `//` (outside of a *string literal* or *comment*) and ending with a newline character U+000A.

block comment
A sequence of characters beginning with the characters `/*` (outside of a *string literal* or *comment*) and ending with the characters `*/`.

4.4.2 Source Files

HeaderFile <- *IncludeStatement** *HeaderFileElement*+
HeaderFileElement <- *TypeDefinition* / *FunctionDeclaration* / *VariableDefinition* / *VariableDeclaration*
A Crowbar header file defines an API boundary, either at the surface of a library or between pieces of a library or application. *IncludeStatement*s can only appear at the beginning of the header file, and header files cannot define behavior directly. Conventionally, a header file has a `.hro` file extension.

ImplementationFile <- *IncludeStatement** *ImplementationFileElement*+
ImplementationFileElement <- *TypeDefinition* / *VariableDefinition* / *FunctionDefinition*
A Crowbar implementation file defines the actual behavior of some piece of a library or application.

It can also define internal types, functions, and variables. *IncludeStatement*s can only appear at the beginning of the implementation file. Conventionally, an implementation file has a `.cro` file extension.

4.4.3 Including Headers

IncludeStatement <- 'include' *string-literal* ';'

Compile-time Behavior:

When encountering this statement at the beginning of a file, the compiler should interpret the string literal as a relative file path, look up the corresponding file in an implementation-defined way, and load the definitions from the given *HeaderFile*.

Runtime Behavior:

This statement has no runtime behavior.

4.4.4 Defining Types

TypeDefinition <- *StructDefinition* / *EnumDefinition* / *UnionDefinition*

Crowbar has three different kinds of user-defined types.

Compile-time Behavior:

When a type is defined, the compiler must then allow that type to be used.

Runtime Behavior:

The definition of a type has no runtime behavior.

StructDefinition <- *NormalStructDefinition* / *OpaqueStructDefinition*
NormalStructDefinition <- 'struct' *identifier* '{' *VariableDeclaration*+ '}'

A `struct` defines a composite type with several members. Its members are stored in the order in which they are defined, and they each take up the space they normally would.

Todo

figure out alignment & padding

OpaqueStructDefinition <- 'opaque' 'struct' *identifier* ';'

An *opaque struct* is a struct whose name is part of an API boundary but whose contents are not. Its size is left unspecified, and it can only be used as the target of a pointer.

EnumDefinition <- 'enum' *identifier* '{' *EnumMember* (',' *EnumMember*)* ','? '}'
EnumMember <- *identifier* ('=' *Expression*)?

An `enum` defines a type which can take one of several specified values.

Todo

define enum value assignment, type-related behavior

UnionDefinition <- *RobustUnionDefinition* / *FragileUnionDefinition*

Unions as implemented in C are not robust by default, and care must be taken to ensure that they are only used robustly. However, for the purpose of interoperability with C, Crowbar unions may be defined as robust or as fragile.

RobustUnionDefinition <- 'union' *identifier* '{' *VariableDeclaration* *UnionBody* '}'
UnionBody <- 'switch' '(' *identifier* ')' '{' *UnionBodySet*+ '}'
UnionBodySet <- *CaseSpecifier*+ (*VariableDeclaration* / ';')

A robust union, or simply union, in Crowbar is what is known more broadly as a tagged union. It's a way to package some data alongside an `enum` but have the type of data depend on the value of the `enum`. Since the enum value indicates which data is present, the enum value is also known as a *tag*. The top-level variable declaration creates the tag. The tag must have a type which is some `enum`. The `switch` parameter must be the name of the tag, and the cases will declare the data associated with a given value of the tag. This allows for storing extra data alongside enum values while using minimal additional space in memory. (All the fields under the `switch` overlap as stored in memory, so it's important to use the tag to specify which field is available.)

For example:

```
enum TokenType {
    Identifier,
    Constant,
    Operator,
    Whitespace,
}

union Token {
    enum TokenType type;

    switch (type) {
        case Identifier: (const byte) * name;
        case Constant: intmax value;
        case Operator: (const byte) * op;
        case Whitespace: ;
    }
}
```

defines a `union Token` type, where the `type` field controls which of the other fields in the union is valid.

Todo

go into more depth about how tagged unions work

FragileUnionDefinition <- 'fragile' 'union' *identifier* '{' *VariableDeclaration*+ '}'

A fragile union also allows for storing one of several different types of data. However, there is no internal indication of which type of data is actually being stored in the union. As such, in non-trivial cases no compiler can predict which field is or is not valid, and any statement which reads a field of a fragile union must itself be a *FragileStatement*.

The size of a fragile union is the largest size of any of its members. The address of each member is the address of the union object itself. The member which was most recently set will retain its value. Reading another member with size no larger than the most recently set member will interpret the first bytes of the most recently set member as a value of the type of the member being read.

For example, the functions `test1` and `test2` are equivalent:

```
fragile union Example {
    float32 float_data;
    uint32 uint_data;
```

```
}

uint32 test1(float32 arg) {
    union Example temp;
    temp.float_data = arg;
    fragile return temp.uint_data;
}

uint32 test2(float32 arg) {
    float32* temp = &arg;
    fragile uint32* temp_casted = (uint32*)temp;
    return *temp_casted;
}
```

4.4.5 Functions

FunctionDeclaration <- *FunctionSignature* ';'

Compile-time Behavior:

Provides a declaration of a function with the name, return type, and arguments specified by the signature, but does not specify any behavior. This is generally used as part of an API boundary.

Runtime Behavior:

A function declaration has no runtime behavior.

FunctionDefinition <- *FunctionSignature Block*

Compile-time Behavior:

Provides the actual behavior of a function, which may have been declared previously or may not. If the function was declared in some `.hro` file which was *include*d, the function must be exported and available for external use in the compiler's output. Otherwise, the function should not be exported.

If the function signature specifies a return type other than `void`, but there are paths through the block that do not execute a *ReturnStatement*, the compiler must give an error.

Runtime Behavior:

When the function is called, the arguments must be populated and the block must be executed.

FunctionSignature <- *Type identifier* '(' *SignatureArguments*? ')'
SignatureArguments <- *Type identifier* (',' *Type identifier*)* ','?

Compile-time Behavior:

A function signature specifies the return type, name, and arguments of a function.

Runtime Behavior:

A function signature has no runtime behavior.

4.4.6 Statements

Block <- '{' *Statement** '}'

Compile-time Behavior:

A block is a possibly-empty sequence of statements surrounded by curly braces. Any declaration or definition within the block must not be visible outside of the block.

Runtime Behavior:

When a block is executed, each of the containing statements, in linear order, is executed.

Statement <- *VariableDefinition* / *StructureStatement* / *FlowControlStatement* / *AssignmentStatement* / *FragileStatement* / *ExpressionStatement* / *EmptyStatement*

Crowbar has many different types of statement.

EmptyStatement <- ';'

Compile-time Behavior:
None.
Runtime Behavior:
None.

FragileStatement <- 'fragile' *Statement*

Some behaviors are difficult to ensure the robustness of at compile time, and these behaviors are defined in this specification as *fragile*. Fragile behaviors used outside of fragile statements should produce a compiler error.
Compile-time Behavior:
Fragile behaviors used inside a fragile statement must not produce a compiler error for their fragility. Nesting fragile statements should produce a compiler error.
Runtime Behavior:
The contained statement is executed.

ExpressionStatement <- *Expression* ';'

Compile-time Behavior:
If the expression is not a function call, the compiler may emit a warning.
Runtime Behavior:
The expression is evaluated and the resulting value is discarded. Function calls must be fully evaluated, but expressions that are not function calls may be optimized out.

Variables

VariableDeclaration <- *Type identifier* ';'

Compile-time Behavior:
A variable declaration specifies the type and name of a variable but not its initial value. This is only used in *HeaderFile*s as part of API boundaries.
Runtime Behavior:
A variable declaration has no runtime behavior.

VariableDefinition <- *Type identifier* '=' *Expression* ';'

Compile-time Behavior:
A variable definition specifies the type, name, and initial value of a variable. If the expression has a type which is not the type specified for the variable, an error must be emitted.
Runtime Behavior:
When a variable definition is executed, the expression is evaluated, and its result is made available with the given name.

Structure Statements

StructureStatement <- *IfStatement* / *SwitchStatement* / *WhileStatement* / *DoWhileStatement* / *ForStatement*

A structure statement creates some nonlinear control structure. There are several types of these structures.

IfStatement <- 'if' '(' *Expression* ')' *Block* ('else' *Block*)?

An if statement allows some action to be performed only sometimes, based on the value of the expression.
Compile-time Behavior:
If the expression does not have type bool, the compiler must emit an error.

Runtime Behavior:

First, the expression is evaluated. If the expression evaluates to a `bool` value of `true`, then the first block will be executed. If the expression evaluates to a `bool` value of `false`, either the second block is executed or nothing is executed.

SwitchStatement <- 'switch' '(' *Expression* ')' '{' (*CaseSpecifier* / *Statement*)+ '}'
CaseSpecifier <- 'case' *Expression* ':' / 'default' ':'

A switch statement allows many different actions to be taken depending on the value of some expression.

Compile-time Behavior:

The expression must have a type which is either some integer type or an enum. The expression in a case specifier must have a value which can always be known at compile time, i.e. its value must be a constant or computed from only constants. Either there must be a case specifier for every valid value in the type of the switch expression, or there must be a default case specifier. At most one default case may be present.

Runtime Behavior:

First, the switch expression is evaluated. Whichever case specifier has the same value, or the default case specifier if none is found, is then selected as the matching case specifier. Any case specifiers immediately following the matching case specifier is ignored. Subsequent statements are then executed, in linear order, until another case specifier is reached. The execution of the switch statement then ends.

WhileStatement <- 'while' '(' *Expression* ')' *Block*

Compile-time Behavior:

The expression must have type bool.

Runtime Behavior:

The expression is evaluated, and, if it evaluates to true, the block is executed. This process repeats until the expression evaluates to false.

DoWhileStatement <- 'do' *Block* 'while' '(' *Expression* ')' ';'

Compile-time Behavior:

The expression must have type bool.

Runtime Behavior:

The block is executed. Then, the expression is evaluated, and if it is true the process repeats.

ForStatement <- 'for' '(' *ForInit*? ';' *Expression* ';' *ForUpdate*? ')' *Block*
ForInit <- *ForInitializer* (',' *ForInitializer*)* ','?
ForInitializer <- *Type identifier* '=' *Expression*
ForUpdate <- *AssignmentBody* (',' *AssignmentBody*)* ','?

Compile-time Behavior:

The individual initializers each have the same behavior as a *VariableDefinition*, but for scope purposes they are treated as though they are inside the block. The top-level expression in the for statement must have type bool, and will be treated for scope purposes as though it is inside the block. The update assignments will be treated for scope purposes as though they are inside the block.

Runtime Behavior:

First, the initializers are executed the same way variable definitions would be, in the order they are presented. Then, the top-level expression is evaluated, and if it is false the for statement ends. If the top-level expression was true, the block is executed, and then the update assignments are executed in the order they are presented. The process repeats starting with expression evaluation.

Flow Control Statements

FlowControlStatement <- *ContinueStatement* / *BreakStatement* / *ReturnStatement*

ContinueStatement <- 'continue' ';'

Compile-time Behavior:

Only valid inside a *WhileStatement*, *DoWhileStatement*, or *ForStatement* block.

Runtime Behavior:

When a continue statement is executed, the innermost loop statement (while, do-while, or for) that contains the continue statement skips the remainder of the execution of its block. For a while or do-while loop, this means skipping to the condition. For a for loop, this means skipping to the update assignments.

BreakStatement <- 'break' ';'

Compile-time Behavior:

Only valid inside a *WhileStatement*, *DoWhileStatement*, or *ForStatement* block.

Runtime Behavior:

When a break statement is executed, the innermost loop statement (while, do-while, or for) that contains the break statement ends entirely, skipping any remaining statements, condition tests, or updates.

ReturnStatement <- 'return' *Expression*? ';'

Compile-time Behavior:

The expression provided must have a type matching the return type of the containing function. If the function has a return type of `void`, the expression must be omitted.

Runtime Behavior:

When a return statement with an expression is executed, the expression is evaluated, and the containing function returns with the value obtained from this evaluation. When a return statement with no expression is executed, the containing function returns.

Assignments

AssignmentStatement <- *AssignmentBody* ';'
AssignmentBody <- *DirectAssignmentBody* / *UpdateAssignmentBody* / *CrementAssignmentBody*

DirectAssignmentBody <- *Expression* '=' *Expression*

Todo

define direct assignment

UpdateAssignmentBody <- *Expression* ('+=' / '-=' / '*=' / '/=' / '%=' / '&=' / '^=' / '|=') *Expression*

CrementAssignmentBody <- *Expression* ('++' / '--')

Todo

define other assignments relative to direct

4.4.7 Types

Type <- *ConstType* / *PointerType* / *ArrayType* / *FunctionType* / *BasicType*

ConstType <- 'const' *BasicType*

PointerType <- *BasicType* '*'

ArrayType <- *BasicType* '[' *Expression* ']'

FunctionType <- *BasicType* 'function' '(' *FunctionTypeArgs*? ')'
FunctionTypeArgs <- *BasicType* (',' *BasicType*)* ','?

Todo

define like any of these

BasicType <- 'void' / 'bool' / 'float32' / 'float64' / 'int8' / 'int16' / 'int32' / 'int64' / 'intaddr' / 'intmax' / 'intsize' / 'uint8' / 'uint16' / 'uint32' / 'uint64' / 'uintaddr' / 'uintmax' / 'uintsize' / 'struct' *identifier* / 'enum' *identifier* / 'union' *identifier* / '(' *Type* ')'

`void` denotes the empty type.

`bool` denotes the Boolean type, with two values: `true` and `false`, represented as 1 and 0, respectively.

`float32` and `float64` denote the binary32 and binary64 IEEE 754 floating-point types, respectively.

`int8`, `int16`, `int32`, and `int64` denote signed, two's-complement integers with sizes 8 bits, 16 bits, 32 bits, and 64 bits, respectively.

`uint8`, `uint16`, `uint32`, and `uint64` denote unsigned integers with sizes 8 bits, 16 bits, 32 bits, and 64 bits, respectively.

`intmax` is a synonym for the largest signed integer type supported by the compiler.

`uintmax` is a synonym for the largest unsigned integer type supported by the compiler.

`uintaddr` is a synonym for an unsigned integer type large enough to hold any memory address valid on the compilation target; the specific type is implementation defined.

`intaddr` is a synonym for a signed integer type at least as large as `uintaddr`.

`uintsize` is a synonym for an unsigned integer type large enough to hold any number of bytes which may be contiguously allocated on the compilation target; the specific type is implementation defined.

`intsize` is a synonym for a signed integer type at least as large as `uintsize`.

`struct`, `enum`, or `union` followed by an identifier denotes the type with the given nature and name, which should be available in the compilation context when used.

Enclosing a *Type* in parentheses does not have semantic significance, but allows for syntactic disambiguation of constructs that would otherwise be visually ambiguous.

Multi-byte integer types should be represented as either big endian or little endian based on the preference of the compilation target platform, i.e. endianness is implementation defined. Compilers targeting less-than-64-bit CPUs *may* omit support for some explicitly sized basic types, but *it would be nice* if they provided software support for types not supported in hardware.

4.4.8 Expressions

AtomicExpression <- *identifier* / *constant* / 'true' / 'false' / *string-literal* / '(' *Expression* ')'

```
ObjectExpression <- AtomicExpression ObjectSuffix* / ArrayLiteral / StructLiteral
ObjectSuffix <- ArrayIndexSuffix / FunctionCallSuffix / StructElementSuffix / StructPointerElementSuffix

ArrayIndexSuffix <- '[' Expression ']'

FunctionCallSuffix <- '(' CommasExpressionList? ')'
CommasExpressionList <- Expression (',' Expression)* ','?

StructElementSuffix <- '.' identifier

StructPointerElementSuffix <- '->' identifier

ArrayLiteral <- '{' CommasExpressionList '}'

StructLiteral <- '{' StructLiteralElement (',' StructLiteralElement)* ','? '}'
StructLiteralElement <- '.' identifier '=' Expression

FactorExpression <- CastExpression / AddressOfExpression / DerefExpression / PositiveExpression / NegativeExpression / BitwiseNotExpression / LogicalNotExpression / SizeofExpression / ObjectExpression

CastExpression <- '(' Type ')' ObjectExpression

AddressOfExpression <- '&' ObjectExpression

DerefExpression <- '*' ObjectExpression

PositiveExpression <- '+' ObjectExpression

NegativeExpression <- '-' ObjectExpression

BitwiseNotExpression <- '~' ObjectExpression

LogicalNotExpression <- '!' ObjectExpression

SizeofExpression <- 'sizeof' ObjectExpression / 'sizeof' Type

TermExpression <- FactorExpression TermSuffix?
TermSuffix <- ('*' FactorExpression)+ / ('/' FactorExpression)+ / ('%' FactorExpression)+

ArithmeticExpression <- TermExpression ArithmeticSuffix?
ArithmeticSuffix <- ('+' TermExpression)+ / ('-' TermExpression)+

BitwiseOpExpression <- ShiftExpression / XorExpression / BitwiseAndExpression / BitwiseOrExpression / ArithmeticExpression

ShiftExpression <- ArithmeticExpression '<<' ArithmeticExpression / ArithmeticExpression '>>' ArithmeticExpression

XorExpression <- ArithmeticExpression '^' ArithmeticExpression

BitwiseAndExpression <- ArithmeticExpression ('&' ArithmeticExpression)+
```

```
BitwiseOrExpression <- ArithmeticExpression ('|' ArithmeticExpression)+

ComparisonExpression <- EqualExpression / NotEqualExpression / LessEqExpression / GreaterEqExpression / LessThanExpression / GreaterThanExpression / BitwiseOpExpression

EqualExpression <- BitwiseOpExpression '==' BitwiseOpExpression

NotEqualExpression <- BitwiseOpExpression '!=' BitwiseOpExpression

LessEqExpression <- BitwiseOpExpression '<=' BitwiseOpExpression

GreaterEqExpression <- BitwiseOpExpression '>=' BitwiseOpExpression

LessThanExpression <- BitwiseOpExpression '<' BitwiseOpExpression

GreaterThanExpression <- BitwiseOpExpression '>' BitwiseOpExpression

LogicalOpExpression <- LogicalAndExpression / LogicalOrExpression / ComparisonExpression

LogicalAndExpression <- ComparisonExpression ('&&' ComparisonExpression)+

LogicalOrExpression <- ComparisonExpression ('||' ComparisonExpression)+

Expression <- LogicalOpExpression
```

Todo

literally all the expression definitions

Todo

figure out if this hierarchy can be tidied up

4.5 License

4.6 TODO

Todo

literally all the expression definitions

(The *original entry* is located in /home/build/crowbar-spec/language/expression.rst, line 79.)

Todo

figure out if this hierarchy can be tidied up

(The *original entry* is located in /home/build/crowbar-spec/language/expression.rst, line 81.)

Todo

figure out https://www.unicode.org/reports/tr31/tr31-33.html

(The *original entry* is located in /home/build/crowbar-spec/language/scanning.rst, line 31.)

Todo

define direct assignment

(The *original entry* is located in /home/build/crowbar-spec/language/statements/assignment.rst, line 9.)

Todo

define other assignments relative to direct

(The *original entry* is located in /home/build/crowbar-spec/language/statements/assignment.rst, line 15.)

Todo

figure out alignment & padding

(The *original entry* is located in /home/build/crowbar-spec/language/type-definition.rst, line 22.)

Todo

define enum value assignment, type-related behavior

(The *original entry* is located in /home/build/crowbar-spec/language/type-definition.rst, line 36.)

Todo

go into more depth about how tagged unions work

(The *original entry* is located in /home/build/crowbar-spec/language/type-definition.rst, line 80.)

Todo

define like any of these

(The *original entry* is located in /home/build/crowbar-spec/language/types.rst, line 16.)

Todo

figure out if that's the right way to handle that

(The *original entry* is located in /home/build/crowbar-spec/safety.rst, line 41.)

4.7 Acknowledgements

- https://matt.sh/howto-c

l

n

o

p

r

s

t

u

v

w

x