B. Comp. Dissertation

A WEB-BASED VISUALIZER FOR C BASED ON EXPLICIT CONTROL

by

CAI KAI'AN

Department of Computer Science
Schoool of Computing
National University of Singapore
2023/2024

Project No: H0411210
Supervisor: Associate Professor Martin HENZ
Examiner: Assistant Professor Manuel RIGGER

Abstract

Source Academy is a web-based computer-mediated environment for learning programming used at NUS and Uppsala University. It supports sub-languages of JavaScript, TypeScript, and Scheme using an explicit-control evaluator and concomitant environment model visualizer that graphically represents the data structures evolving in a running program. This project aims to develop a web-based explicit-control evaluator for a suitably chosen sublanguage of C and integrate it into Source Academy. The goal is to create an environment for running C programs suitable for first-year computer science students. The mental model encouraged by the visualizer avoids knowledge of a C compiler or intimate details of the underlying hardware, which tend to be obstacles for many learners. At the same time, students will still be exposed to important concepts such as C's memory model, and have access to a standard compliant implementation of C.

Subject Descriptors:

Applied computing∼Education∼Interactive learning environments

Software and its engineering∼Software notations and tools∼Compilers∼Interpreters

Human-centered computing∼Visualization∼Visualization systems and tools

Keywords:

Programming Language Implementation, Systems For Teaching and Learning

Acknowledgments

I would like to express my heartfelt gratitude to my project advisor, Prof. Martin Henz, for the invaluable guidance provided, without which this project would not have been possible. His enthusiasm and passion for computer science education, along with the great pride he takes in his work on Source Academy, has rubbed off on me and spurred me on throughout the project. Next, I would like to thank Prof. Roland Yap, for sharing his expertise on C programming and for providing valuable insights for the project. I also wish to thank Prof. Manuel Rigger, the main evaluator of this project, for providing detailed feedback during the continuous assessment.

Contents

1 Introduction

1.1 Background

The C programming language has stood the test of time, remaining an essential cornerstone in the realm of computer science and computer engineering. Its fine-grained control over hardware and efficient system-level programming capabilities have made it a fundamental language taught to many first-year computer science students. However, teaching C to beginners have always presented a unique set of challenges due to its low-level nature and the necessity for understanding complex concepts such as memory management and compilation. Computer science educators have had to grapple with such difficulties, often relying on traditional teaching methods such as textbooks and compiler-based environments. While these methods have their merits, they fail to provide practical hands-on experience to build a strong foundation in C programming without resorting to the complexities of compilation and linking.

Source Academy [9] is a web-based computer-mediated environment for learning programming used at NUS and Uppsala University. It supports sub-languages of JavaScript, TypeScript, and Scheme using an explicit-control evaluator and concomitant environment model visualizer that graphically represents the data structures evolving in a running program. Such tools have been used to great success in teaching introductory programming methodology courses at NUS. We look to build upon this success and explore visualizations of C programs, which present vastly different challenges and implications.

1.2 Objectives

The goal of this project is to develop a web-based explicit-control evaluator for a suitably chosen sub-language of C and integrate it into Source Academy. The tool developed should be able to accept any user supplied C program that follows the sub-language chosen and evaluate the program. During the evaluation, the evaluator keeps track of the state of the evaluation through 4 main components: control (a stack of instructions that represents the control flow of the program), stash (a stack of temporary values the arises in evaluating certain statements), stack (a stack of stack frames keeping track of local variables for each function call), and heap (a region of memory that contains dynamically allocated objects). The evaluation of any C program can be seen as a sequence of such states, and the tool will be able to jump to and visualize the states at any point in the evaluation.

In addition to visualizing how the program is evaluated, the tool also aims to provide visualizations on various memory concepts such as how `structs` are laid out in memory and how pointers work. Furthermore, settings, such as endianness, will be made available to users that allow them to play around with how their program is evaluated. They can then get to see how differences in underlying hardware may affect the correctness of their program. Key concepts in C such as undefined behaviours, strict aliasing, and use-after-free will be able to be demonstrated using this tool as well.

We envision the result of the project to be an environment for running simple C programs suitable for first-year computer science students. The mental model encouraged by the visualizer avoids knowledge of a C compiler or intimate details of the underlying hardware, which tend to be obstacles for many learners. Students can see the evaluation of their programs as the incremental decomposition into its parts: definitions, function calls, assignments, etc. At the same time, they will still be exposed to important concepts such as C's memory model, and have access to a standard compliant implementation of C. By deliberately "leaking" details of the underlying system and hardware architecture into the observable state, we hope to improve learning outcomes and develop lower level intuition for first year students.

The rest of this report is organized as follows. In Chapter 2, we take a look at some of the existing tools and their potential limitations. Some of the features of the tool are demonstrated in Chapter 3 and Chapter 4 discusses the design consideration and implementation details of the system. Chapter 5 summarizes the report and evaluates the current tool for its limitations and future improvements.

2

Literature Review

The tools that seek to visualize the execution of C programs along with its memory, or part thereof, can be broadly classified into 2 categories: educational and debugging. While this project is part of the former, we also look to C debugging tools that have a visualization component and evaluate their effectiveness in teaching C programming.

2.1 Educational Tools

2.1.1 Source Academy Environment Visualizer

Source Academy [9] is a web-based computer-mediated environment for learning programming used at NUS and Uppsala University. It supports sub-languages of JavaScript, TypeScript, and Scheme using an explicit-control evaluator and concomitant environment model visualizer [1] that graphically represents the data structures evolving in a running program. The environment visualizer has an option to visualize the control and stash (as defined in Chapter 1) in addition to the environment (Figure 2.1).

While the semantics of the control and stash are very similar to this project, the current tool mainly targets sub-languages of JavaScript, which has significant differences with C. JavaScript combines lexical scoping with first-class functions, which necessitates a unique mental model and tool to reason about how environment structures evolve over an execution of a program. JavaScript also automatically allocates everything on the heap and implements a mark-and-sweep garbage collector. On the other hand, C does not have first-class functions or closures, and hence require

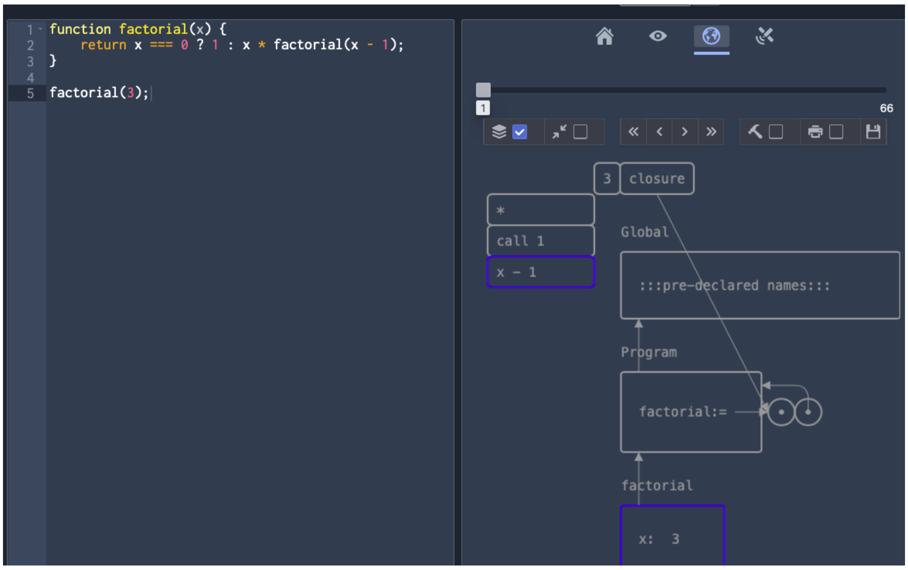


Figure 2.1: A visualization of an execution of a factorial program

the traditional stack and heap model to reason about scopes and object lifetime. Furthermore, C gives the programmer much more control over the underlying memory with dynamic allocations using `malloc` or memory operations using pointers. Hence, it is clear that a drastically different approach needs to be taken when it comes to visualizing C programs.

2.1.2 Python Tutor

Python Tutor [4] is a popular tool providing visualization of the execution of Python, Java, C, C++ and JavaScript programs. It is able to provide high level overviews of the stack and heap memory as the program runs, and has a simple arrow that points to the current line of execution to show control flow. This is done by compiling the C program on the server and running it using the Valgrind Memcheck plugin [10], allowing for insights into how the program memory looks like as it runs line to line.

However, by nature of this implementation, there exists several limitations. Some of these include: the return value of a function, if not stored in a variable, will not be shown; memory that is leaked cannot be visualized; function pointers cannot be visualized as function code is not shown. Most of the limitations can be attributed to the fact that the visualization relies on actually compiling and executing the

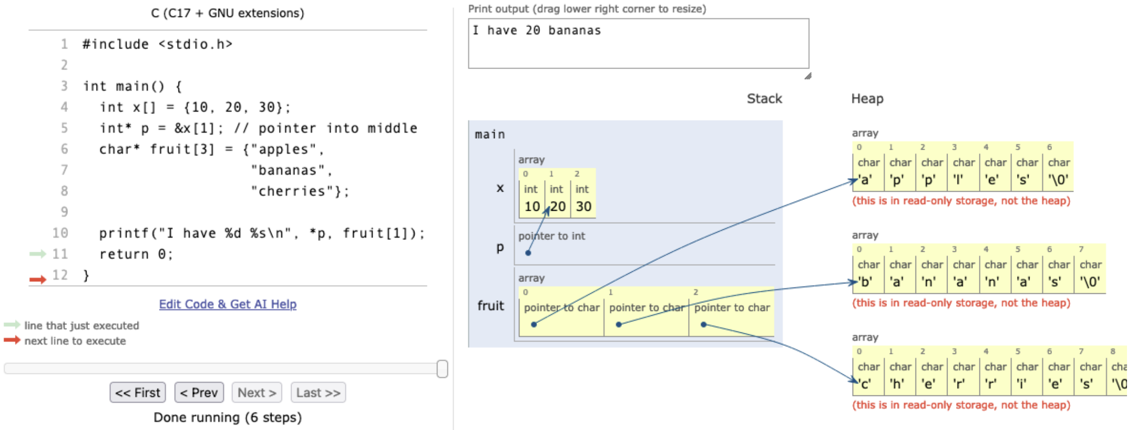


Figure 2.2: Python Tutor

C program, which results in numerous restrictions imposed by the compiler and operating system. Indeed, this implementation allows for a visualization of the program that more closely resembles an actual execution on actual hardware, but in the context of an environment suitable for first-year computer science students learning C, it may only confuse and hinder understanding. Furthermore, control flow in Python Tutor is represented as an arrow that jumps between lines of code, which is neither granular nor self explanatory.

Our project provides more detailed visualizations of the stack and heap, with concepts such as alignment and padding. It also introduces a high level mental model involving control and stash that does away with compilation, providing a better understanding of control flow for first year learners. This is, however, done at the expense of accuracy with regards to an actual execution of the program, which requires various compiler optimizations and is dependent on the actual hardware used. With that being said, by implementing components such as memory by ourselves, this affords us the freedom to play around with concepts such endianness, which might otherwise be hard to change.

2.2 Debugging Tools with Visualizations

It is important to note that these tools have vastly different purposes and use cases as compared to previously mentioned works. Debuggers are mainly used to run actual programs under controlled conditions that allow programmers to track its execution and monitor certain effects on computer resources. They can be used to catch existing bugs in actual code bases. On the other hand, educational tools like this project is the most effective when used on "toy" programs or smaller pieces of code with specific purposes and learning outcomes in mind. They may be used instead to teach programmers about common bugs and certain unexpected behaviours.

2.2.1 GNU Data Display Debugger

GNU Data Display Debugger (DDD) [3] is a graphical front end for the command-line debugger GDB. It is most commonly used for its interactive graphical data display, where data structures are displayed as graphs (Figure 2.3).

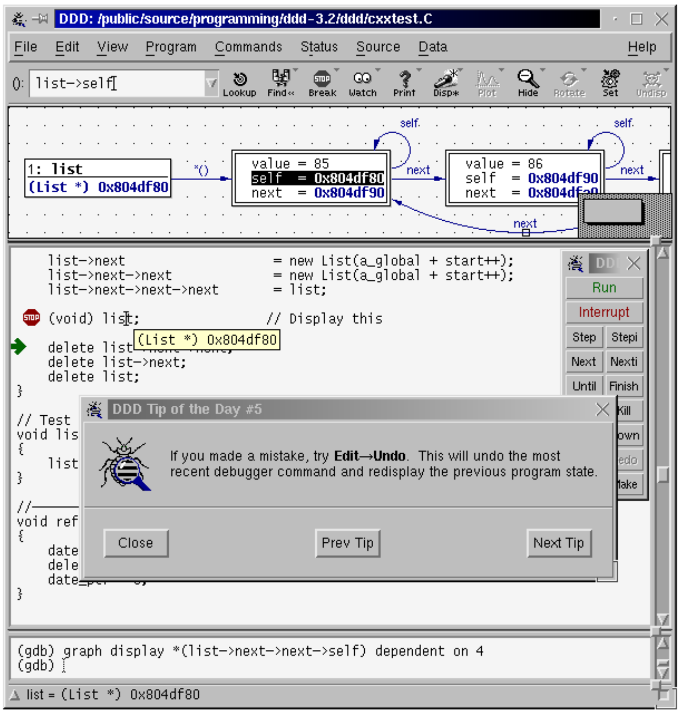


Figure 2.3: GNU Data Display Debugger

The limitations of DDD as an educational tool for introductory programmers are similar to those of Python Tutor. Control flow is represented as arrows jumping around in the source code and the restrictions that come with having to compile and execute the binary on hardware have been discussed previously.

2.3 Summary

Indeed, none of the current tools clearly visualizes how control flow, along with stack and heap memory, evolves during the execution of a C program in a manner suitable for instruction. The novel contribution of this project is to develop a web-based interpreter for C programs that is able to visualize stack and heap memory, along with a high level mental model describing control flow. By detaching ourselves from needing to compile and run the program on actual hardware, we are afforded much more control over the semantics of the language as well as how various elements can be visualized.

3
Features

3.1 Overview

Figure 3.1 shows the visual layout of the tool. It can be sectioned into the left part, which contains the code editor and settings, and the right part, which contains 5 panes corresponding to Control, Stash, Text & Data, Stack and Heap. When the user clicks on "Run", detailed information about the execution of the program will be displayed in the corresponding panes. These panes are motivated by the actual components of process memory in the Operating System.

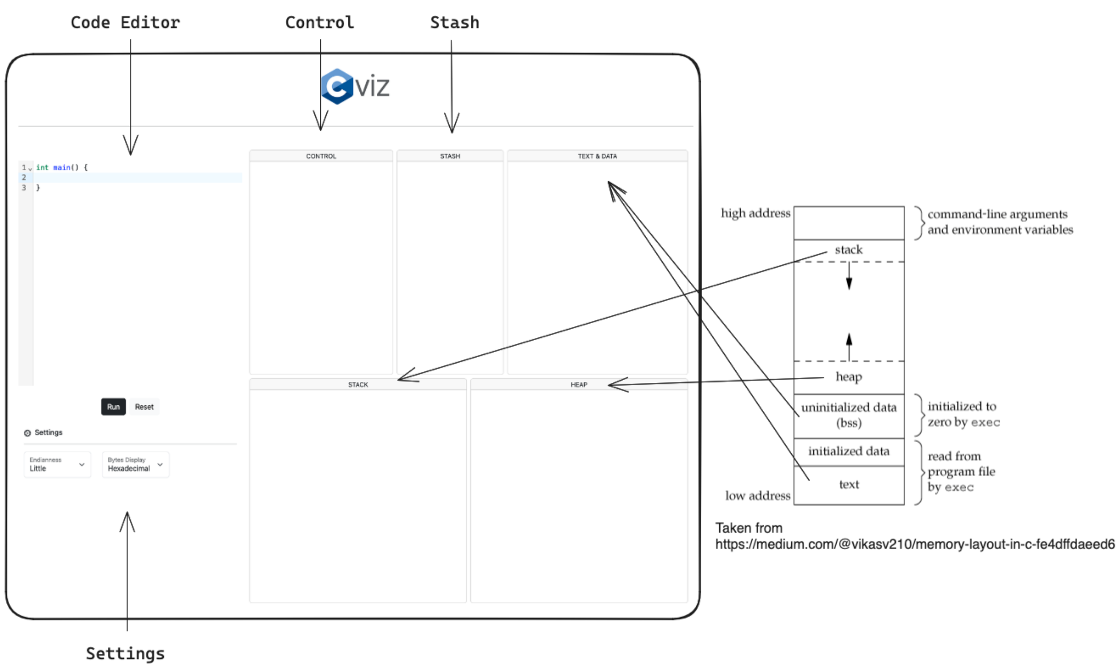


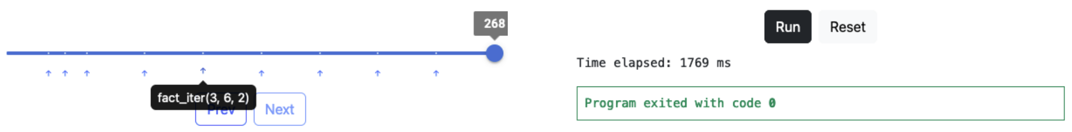
Figure 3.1: Overview of the tool

For the rest of this section, we will look at the visualization of Code 3.1.

```
int b = -2, d = 0x0afa0101;
unsigned long long e = 'a';
int fact_iter(int i, int n, int acc) {
  return i > n ? acc : fact_iter(i + 1, n, acc * i);
}
int fact(int n) {
  return fact_iter(1, n, 1);
}
int main() {
  fact(2 * 3);
}
```

Code 3.1: Static variables and calculating 6!

After the user runs the program, 2 new elements pop up in the left section of the tool: navigation and statistics (Figure 3.2). The navigation gives the user a slider to jump around to any point in the execution and also provides "bookmarks" on the timeline corresponding to function calls. On hover, the values of the arguments are shown. Statistics such as time elapsed and program exit code are displayed as well. In the case of errors during parsing, type checking or evaluation, an appropriate error message with line numbers is displayed here.



(a) Navigation (b) Statistics

Figure 3.2: Navigation and Statistics

The "Control" pane (Figure 3.3) displays the stack of instructions yet to be executed. Instructions are either AST nodes corresponding to some piece of the program or some operation defined by the interpreter that consumes values from the stash and alters the program state. From some state s in the execution, the interpreter pops the next instruction to be executed from the top of the Control and looks up the type of the AST node or operation. Every instruction type has its own semantics on how it transitions the program state from s to s'. For example, for the

AST node of type `PrimaryExprConstant`, the interpreter consumes this AST node and pushes a temporary value onto the stash, with value and type as specified in the C standard.

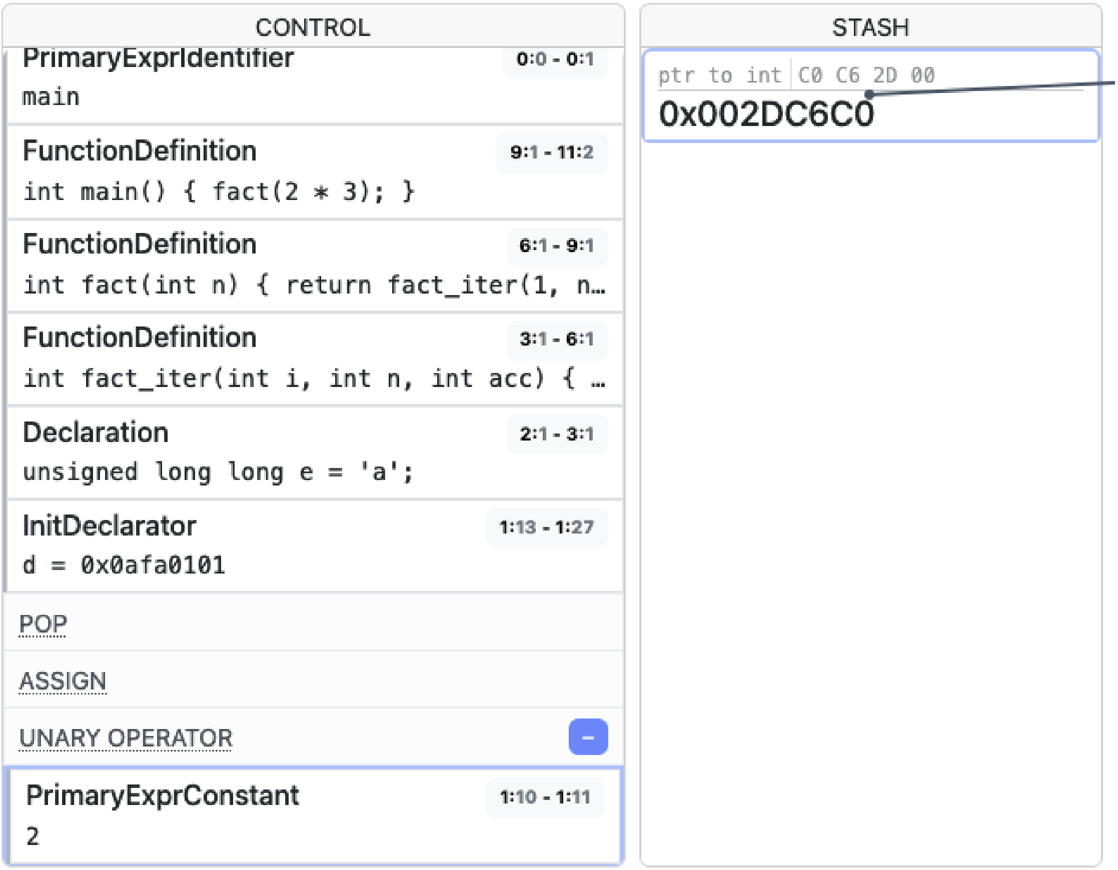


Figure 3.3: Control and Stash

The visualization of AST nodes in Control displays their corresponding type, source code and line numbers. These nodes can be clicked to display a modal containing more detailed information. For operations, we display the type, arguments (if there is any), as well as helpful tool tips about what the operation does.

The Stash (Figure 3.3) contains temporary values that arise as a result of evaluating certain expressions. We display their types and hexadecimal representation as well.

The "Text & Data" pane (Figure 3.4) contains what the C standard refers to as External Declarations. These are declarations that lie outside of function bodies and hence have file scope and static storage duration. The contents of this pane include either Function Designators or objects. For function designators, we show the function name, its address in the text portion of memory, its source code as well as argument and return types. For objects, we show its identifier, type, address in the data portion of memory, as well as its contents both as hexadecimal values as well as human readable form. We can click on objects as well to reveal a modal that contains more information about that object. This will be shown in later examples.

Figure 3.4: Text & Data

The "Stack" pane (Figure 3.5) contains the stack frames that arise from function calls. When a function is called, the size and layout of the stack frame is determined by scanning through the various declarations and their types. For each frame, the corresponding function name is displayed, in addition to the base address for each frame and the objects contained in that frame.

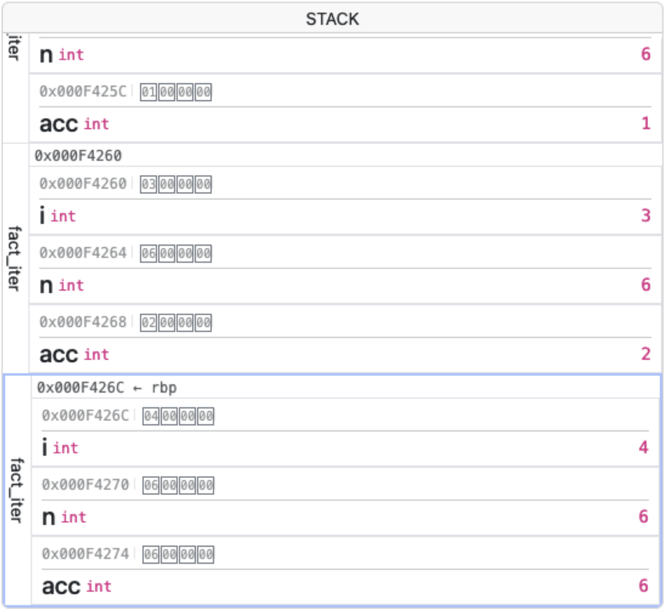


Figure 3.5: Stack

3.2 Interpreter and Visualisation Settings

The tool allows users to change certain visualization and interpreter settings. For now, the supported settings are "Bytes Display" - either "Binary" or "Hexadecimal" - and "Endianness" - either "Little" or "Big". The bytes display is a visualizer setting that only modifies how bytes are displayed on the frontend, while endianness affects the actual interpretation of the program.

The goal is for students to be able to play around with such settings and observe how it might interact with their programs. In Chapter 5 we discuss interesting settings that can be included here in the future.

3.3 Structures

To demonstrate how structures are visualized, we enter the following program.

```
struct test {
  short ss; char c[3][2]; int a; char * color; char d;
  struct inner { short s; _Bool w; } s;
} xs[3];
int main() {}
```

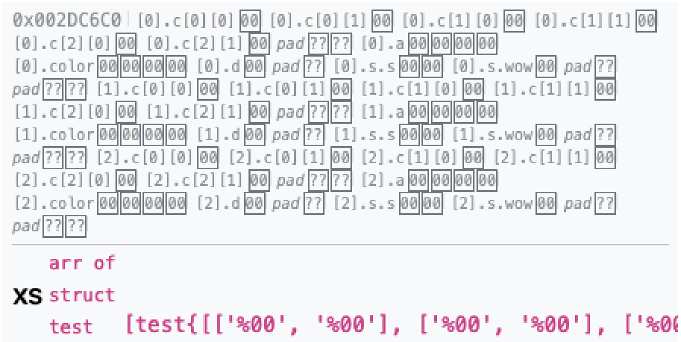


Figure 3.6: Struct (inline display)

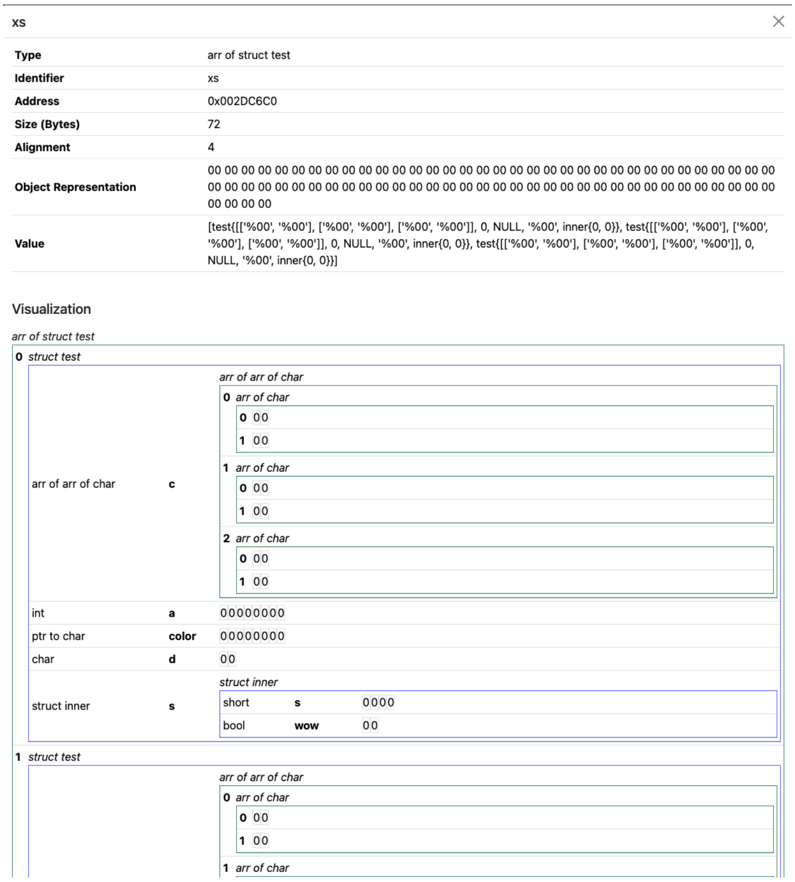


Figure 3.7: Struct (modal)

There are 2 different views for structures. The inline view (Figure 3.6) is always displayed while the modal view (Figure 3.7) appears only upon clicking on the inline view. In the inline view for structures, we can see all the bytes corresponding to that structure, consisting of bytes that take part in the value representation as well as padding bytes. The bytes of each member is labelled on its left with notation borrowed from structure initialization. In the modal view, we see more information, such as the size and alignment of the structure, as well as a visualization of how the members are organized in memory.

3.4 Heap

We use the following code.

```
int * p = (int *) malloc(sizeof(int) * 10);
int * q = (int *) malloc(5); char * c;
int main() {
    c = (char *) p; c = c + 20; *c = 'a';
    p[1] = 10; *q = 5; free(c);
}
```

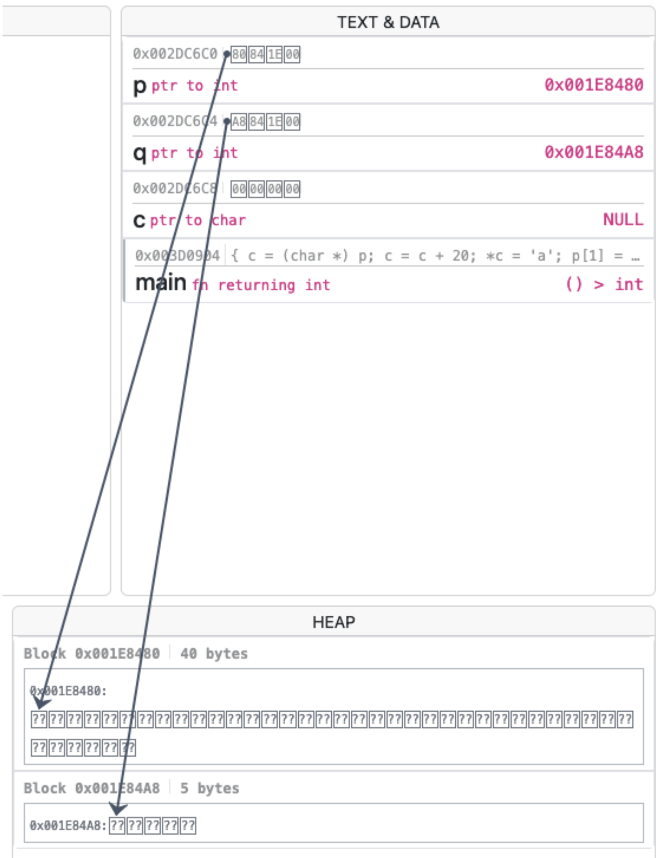


Figure 3.8: Heap (before any write)

The heap (Figure 3.8) is displayed as a list of blocks of allocated memory, sorted by increasing address. Each block has its address and size in bytes displayed, along with its contents. We see that right after each block is allocated, before any writes happen, the memory in the blocks are uninitialized with no effective type. Figure 3.9 shows what the heap looks like after some writes to the memory. As specified by the C standard, an effective type can be acquired by writing to that object first with an lvalue that has a type other than character type. Hence, the block of memory now contains bytes that have no effective type as well as some with effective types.

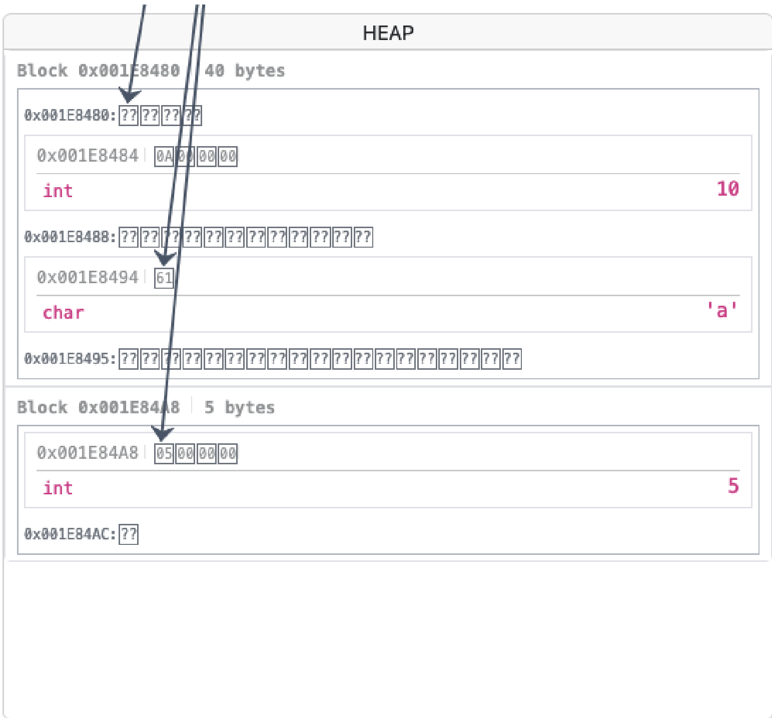


Figure 3.9: Heap (after writes)

3.5 Pointers

For objects of type pointer, we draw arrows from the pointers to the objects to which they point at. This is a natural mental model of how pointers work. In the case where a pointer contains an address that is either misaligned for its type or points to an unallocated space in memory, we give the pointer a red background. This can be instructive for concepts such as use-after-free.

3.6 Undefined Behaviours

Another important concept in C is that of undefined behaviours. The interpreter is currently able to detect some of these, especially more common ones. Some examples include use-after-free, strict aliasing, and signed overflow. In Section 3.4, the code provided also invokes undefined behaviour by calling `free` on an address not returned by `malloc`. Currently, the interpreter stops execution and warns user when such undefined behaviours happen. However, in Chapter 5 we discuss the possibility of adding interpreter settings in which users can control what the interpreter does in events of certain undefined behaviours.

4

System Design and Implementation

The system is structured into 2 repositories, `c-viz` and `c-viz-frontend`. The back-end `c-viz` contains the parser, type-checker as well as the explicit control interpreter, and can be used to run C source code as a standalone program. `c-viz` is published as a `npm`[7] package to be then imported by `c-viz-frontend`. The user interface along with the visualization engine exists in `c-viz-frontend`, which is built as a React [8] application.

Figure 4.1 describes what happens when a user enters their C source code to be visualized.

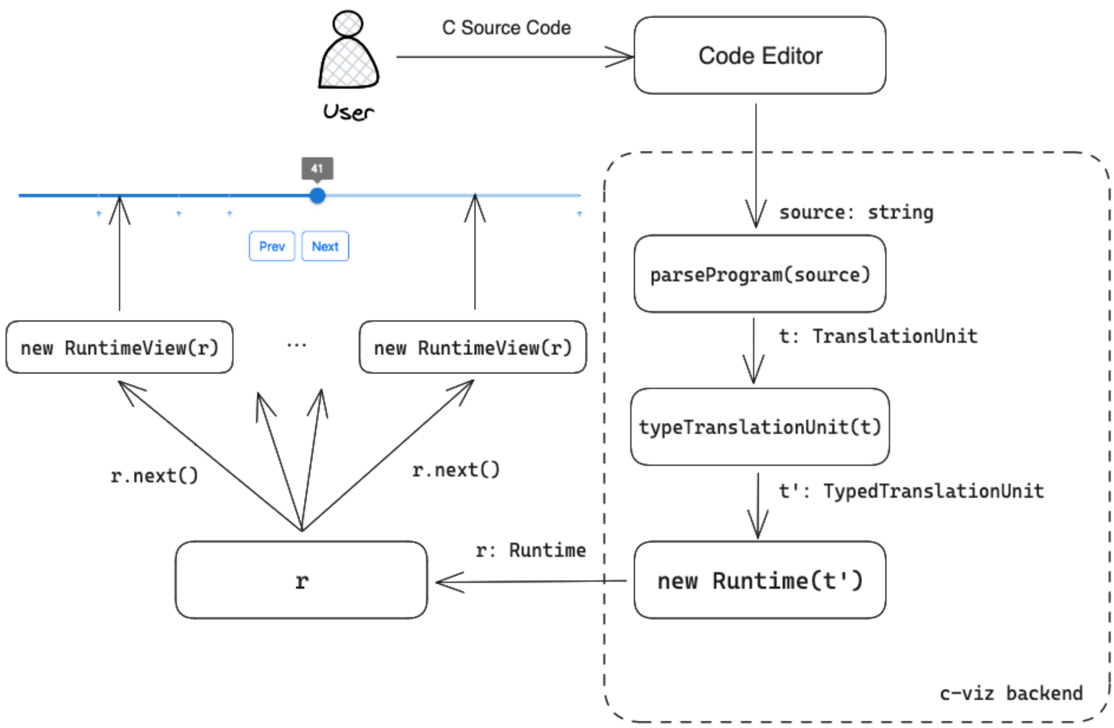


Figure 4.1: How a user program is visualized

The rest of the chapter looks at the design and implementation of the various components. Henceforth, the version of C standard used for reference is ISO/IEC 9899:1999 - n1256, available at [5].

4.1 Parser

The parser accepts as input the C source code to be evaluated and parses it into an Abstract Syntax Tree (AST). In development of the parser, a couple of parser generators were considered including `Peggy` and `antlr`. After consideration of their features, `Peggy` was chosen for its well documented JavaScript API and excellent error reporting. There was also existing work [6] done on `C++` parsers using `Peggy` that could be referenced. During build time, `Peggy` takes in a Parsing Expression Grammar (PEG) file and generates a parser function that will be used by other components in the back-end.

4.1.1 Parsing Expression Grammar

Parsing expression grammar is a type of analytic formal grammar first described by Bryan Ford in 2004 [2] to address the issue of ambiguity arising from context-free grammars (CFGs) and regular expressions (REs). Even though ambiguity expresses nondeterministic choices and is crucial to the power of such generative grammars, it makes it difficult to express and parse machine-oriented languages. PEGs provide an alternative for describing such syntax by removing ambiguity though prioritized choice: always selecting the first match. Consequently, for PEGs, a string has exactly one valid parse tree or none.

For the project, the parser's PEG file was developed in close accordance to the C language syntax rules found in the standard. Except for cases where the rules lead to ambiguity in the grammar, there is a one to one correspondence. This ensures that any syntactically correct C program entered by users will be parsed, allowing us to then look for specific unsupported language constructs and display a nice error message.

In the case of `Peggy`, we can easily incorporate the construction of the AST into our grammar file. This is done by including a block of JavaScript code right after each grammar rule, which will then be executed by the parser upon a match.

Figure 4.2 demonstrates how a declaration might be parsed.

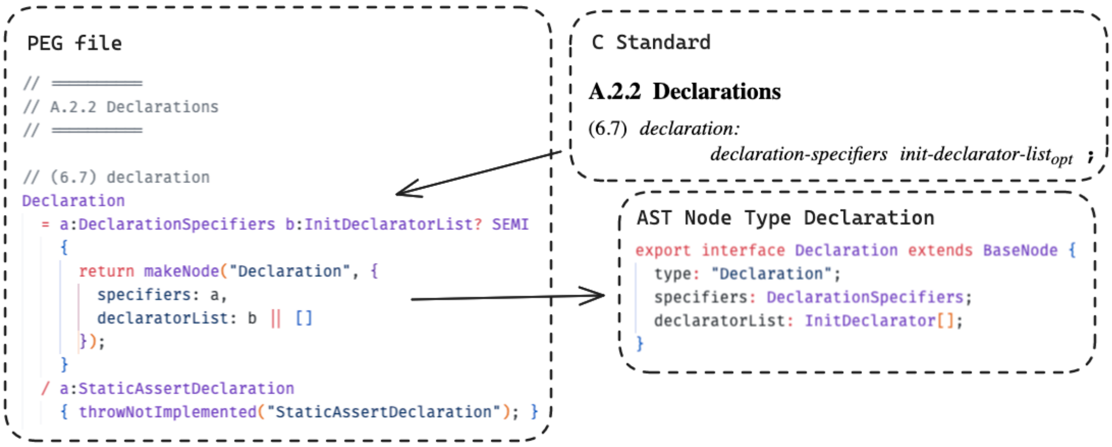


Figure 4.2: Translation of C standard syntax rules into PEG

One notable case where ambiguity arises is in the syntax for `typedef`s. This was resolved by slightly modifying some of the PEG rules and paying attention to the ordering of the rules.

4.2 Type System

The type system developed for this project is extensive and contains features such as

- Arithmetic Types, Arrays, Structures, Pointers, Function Types, Incomplete Types
- Value Representation: how are objects of a certain type stored in memory
- Declarations: parsing declarations and constructing derived types correctly
- Implicit Conversions: such as integer promotions, usual arithmetic conversions, conversion as if by assignment, function to pointer conversion, array to pointer conversion
- Type Environment: with different namespaces for ordinary identifiers and tags of structures

- Type Checking: standard compliant implementation of a type checker for numerous language constructs

4.3 Memory

Memory is implemented as a contiguous byte array using JavaScript's `ArrayBuffer`, with an additional data structure `EffectiveTypeTable` that keeps track of effective types of objects in memory. This way, we get the benefit of using a byte array - similar behaviour to a platform such as x86 - while having the flexibility to check for concepts such as strict aliasing and use-after-free.

5
Conclusion

5.1 Limitations and Future Work

These include

- Formalizing static semantics and small step semantics
- Formal language specification of the subset of C supported
- Floating arithmetic types
- Support for more language constructs
- Support for more library functions (`cmath.h`, etc.)
- Preprocessing
- Interpreter setting to change how undefined behaviour affect execution

Bibliography

[1] K. Cai., M. Henz., K. Low., X. Ng., J. Soh., K. Tang., and K. Toh., "Visualizing environments of modern scripting languages", in *Proceedings of the 15th International Conference on Computer Supported Education - Volume 1: CSEDU*, INSTICC, SciTePress, 2023, pp. 146–153, ISBN: 978-989-758-641-5.

[2] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation", in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04, Venice, Italy: Association for Computing Machinery, 2004, pp. 111–122, ISBN: 158113729X. [Online]. Available: `https://doi.org/10.1145/964001.964011`.

[3] "GNU Data Display Debugger", `https://www.gnu.org/software/ddd/`.

[4] P. Guo, "Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia", in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1235–1251, ISBN: 9781450386357. [Online]. Available: `https://doi.org/10.1145/3472749.3474819`.

[5] "ISO/IEC 9899:1999 - n1256", `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.

[6] "JSCPP", `https://github.com/felixhao28/JSCPP`.

[7] "Node Package Manager", `https://www.npmjs.com/`.

[8] "React", `https://react.dev/`.

[9] "Source Academy", `https://sourceacademy.org/`.

[10] "Valgrind memcheck", `https://valgrind.org/docs/manual/mc-manual.html`.

c-viz — Specification for v0.1.2*

Cai Kai'An†

26/04/2024

c-viz is a project that aims to develop a web-based explicit-control evaluator[1] for a suitably chosen sublanguage of C. The goal is to create an environment for running C programs suitable for first-year computer science students. The mental model encouraged by the visualizer avoids knowledge of a C compiler or intimate details of the underlying hardware, which tend to be obstacles for many learners. At the same time, students will still be exposed to important concepts such as undefined behaviours and C's memory model.

This document specifies the subset of C supported.

Contents

*Source code made available at this public repository

†E-mail: kaian531@gmail.com

[1]As described in SICP - JS Edition Section 5.4

Introduction

This document follows the structure of (and frequently refers to[2]) the ISO/IEC 9899:1999 - n1256 version of the C99 standard[3]. Where applicable, we express grammars using Extended Backus–Naur form[4], adopting the ISO/IEC 14977 standard proposed by R. S. Scowen.

Deviations from the `C` standard will be on their own paragraph and highlighted as such.

Section 1 – *Concepts* describes various language concepts, such as scopes of identifiers and types. Section 2 – *Lexical Grammar* to section 6 – *External Definitions* detail the syntax, type checking constraints and semantics of various language constructs. In section 7 – *Memory Model* we look at the the various memory segments and memory related runtime data structures implemented in the interpreter. Lastly, an informal description of the big step operational semantics of the language is provided in section 8 – *Operational Semantics*.

[2] certain definitions and wordings may be lifted *verbatim* from the standard

[3] https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[4] we use ::= for definition, , for concatenation, | for alternation, [x] for optional x, { x } for zero or more repetitions of x, (...) for grouping, '...' for terminals, – for negation and ?...? for special sequences.

1 Concepts

1.1 Numerical limits

The following are the constant values defined for the numerical limits of integer types. A future version of `c-viz` may allow for configuration of these values.

CHAR_BIT	8	number of bits for smallest object (byte)
SCHAR_MIN	$-(2^7-1)$	minimum value for an object of type `signed char`
SCHAR_MAX	2^7-1	maximum value for an object of type `signed char`
UCHAR_MAX	2^8-1	maximum value for an object of type `unsigned char`
CHAR_MIN	$-(2^7-1)$	minimum value for an object of type `char`
CHAR_MAX	2^7-1	maximum value for an object of type `char`
MB_LEN_MAX	1	maximum number of bytes in a multibyte character
SHRT_MIN	$-(2^{15}-1)$	minimum value for an object of type `short int`
SHRT_MAX	$2^{15}-1$	maximum value for an object of type `short int`
USHRT_MAX	$2^{16}-1$	maximum value for an object of type `unsigned short int`
INT_MIN	$-(2^{31}-1)$	minimum value for an object of type `int`
INT_MAX	$2^{31}-1$	maximum value for an object of type `int`
UINT_MAX	$2^{32}-1$	maximum value for an object of type `unsigned int`
LONG_MIN	$-(2^{31}-1)$	minimum value for an object of type `long int`
LONG_MAX	$2^{31}-1$	maximum value for an object of type `long int`
ULONG_MAX	$2^{32}-1$	maximum value for an object of type `unsigned long int`
LLONG_MIN	$-(2^{63}-1)$	minimum value for an object of type `long long int`
LLONG_MAX	$2^{63}-1$	maximum value for an object of type `long long int`
ULLONG_MAX	$2^{64}-1$	maximum value for an object of type `unsigned long long int`

1.2 Scopes of identifiers

An identifier can denote an object; a function; a tag or a member of a structure; or a typedef name. The same identifier can denote different entities at different points in the program.

The *scope* of an identifier is a region of program code in which the identifier may be used to refer to a particular entity. An identifier can designate different entities only if they are in different scopes or in different name spaces.

There are two kinds of scopes: file and block. Function prototype scope and function scope[5] are not supported.

The scope of an identifier is determined by where its declaration appears. If the declaration appears outside of any block, the identifier has *file scope*, which terminates at the end of the translation unit. If it appears inside a block, the identifier has *block scope*, which terminates at the end of the block.

If two different entities existing in the same name space have the same identifier, then the scope of one of them (the *inner scope*) must be a strict subset of the other (the *outer scope*). Within the inner scope, the identifier will designate the entity declared within and the entity declared in the outer scope is *hidden*. Two identifiers is said to have the *same scope* if their scopes terminate at the same point.

Structure tags have scopes beginning just after the appearance of the tag in the type specifier portion of its declaration. All other identifiers have scope beginning just after the completion of its declarator.

[5] used only by label names for `goto` constructs

1.3 Name spaces of identifiers

If more than one declaration of an identifier is visible at any point, the context in which the identifier is used disambiguates what entity it refers to. Thus, there are separate *name spaces* for identifiers:

- the *tags* of structures (appears right after the keyword `struct`)
- the *members* of structures; each structure has a separate name space for its members (disambiguated by the type of the expression used to access the member)
- all other identifiers

1.4 Storage duration of objects

An object has a *storage duration* that determines its lifetime. There are three storage durations: static, automatic, and allocated.

The *lifetime* of an object is the portion of program execution during which memory is allocated to it. Hence, an object has a constant address and retains its last-stored value throughout its lifetime.

If an object is referred to outside of its lifetime, the interpreter throws the error `"no object allocated at ..."`. Future versions of `c-viz` should allow for such runtime checks to be disabled.

Pointers retain their values when objects they point to reaches the end of their lifetime.

An object designated by an identifier with file scope has *static storage duration*, with its lifetime spanning the entire execution of the program[6].

An object designated by an identifier with block scope has *automatic storage duration*. Its lifetime extends from entry into the block until execution of that block ends[7]. The initial value of the object is indeterminate.

Allocated storage duration is described in subsection 7.3 – *Heap allocations*.

1.5 Types

Types give meaning to values stored in objects. They are partitioned into:

- *object types* which fully describe objects
- *function types* that describe functions
- *incomplete types* that describe objects (or `void`) but lack information to determine their sizes

The types supported by `c-viz` are: `_Bool`, `char`, `signed char`, `unsigned char`, `short int`, `unsigned short int`, `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`, `void`, arrays, structures, functions and pointers. Other than functions (function type) and `void` (incomplete type), the other types are object types.

The *signed integer types* consists of `signed char`, `short int`, `int`, `long int` and `long long int`.

For each of the signed integer types, there is a corresponding unsigned integer type designated with the keyword `unsigned`. They use the same amount of storage and have the same alignment requirements. These types, combined with the type `_Bool`, make up the *unsigned integer types*.

[6]program execution begins on entry to the `main` function and ends when the `main` function returns
[7]the corresponding `EXIT_BLOCK` instruction is popped from the control stack

If a signed integer type can represent a range of nonnegative values S and its corresponding unsigned integer type can represent a range of values U, then $S \subseteq U$. Furthermore, the representation of any $x \in S$ is the same for both types.

Computations involving unsigned operands can never overflow. A result x that cannot be represented by the resulting unsigned integer type with maximum value M will be reduced to $x \mod (M + 1)$.

The interpreter throws an error on signed integer overflows.

Real and complex floating types are currently not supported.

The type `char` along with the signed and unsigned integer types make up the *integer types* and *arithmetic types*. The three types `char`, `signed char` and `unsigned char` make up the *character types*.

The type `char` is defined to have the same range, representation and behaviour as `signed char`.

The `void` type denotes an empty set of values and is an incomplete type that cannot be completed.

Any number of *derived types* may be constructed from the object, function and incomplete types:

- An *array* type describes a contiguously allocated nonempty set of objects with a particular *element type*. Arrays are characterized by their element type and the number of elements.
- A *structure* type describes a sequentially allocated nonempty[8] set of member objects, each of which has an optionally specified name.
- A *function* type describes a function with some return type. It is characterized by its return type and the number and types of its parameters.
- A *pointer* type may be derived from any type, called the *referenced type*. It describes an object whose value is the memory address of an entity of the referenced type.

These derived types may be constructed recursively.

Union types are currently not supported.

The *scalar types* consists of arithmetic and pointer types. The *aggregate types* consist of array and structure types.

Array types of unknown sizes[9] or variable length are not supported.

A structure type of unknown content is an incomplete type. It can be completed by declaring the same structure tag with its defining content later in the same scope[10].

A type has known constant size if the type is not incomplete.

Only *unqualified* types are supported.

Pointers to any type have the same representation and alignment requirements.

[8] empty structures are supported by some `C` compilers and is also a valid `C++` construct

[9] arrays of unknown sizes `T[]` in function parameter lists can be replaced by corresponding pointer types instead; their only other use in `C` is within `extern` declarations, which are not supported

[10] see example in section 6 – *External Definitions*

1.5.1 Size and alignment

Every complete object type has two properties, *alignment requirement* and *size*. The size of an object is the number of bytes in memory that it occupies. The alignment requirement of an object is a non-negative integral power of two representing the number of bytes between successive addresses at which objects of this type can be allocated.

The weakest (smallest) alignment is the alignment of the types `char`, `signed char` and `unsigned char`, which equals 1.

An object of type `int` has large enough size to contain any value in the range **`INT_MIN`** to **`INT_MAX`** as defined in subsection 1.1 – *Numerical limits*.

The values for the alignment requirements and sizes of the scalar types used by `c-viz` are typical for 32-bit x86 architectures. Future versions of `c-viz` may allow configuration of these values.

CHAR_SIZE	1	**CHAR_ALIGN**	2^0
SCHAR_SIZE	CHAR_SIZE	**SCHAR_ALIGN**	2^0
UCHAR_SIZE	CHAR_SIZE	**UCHAR_ALIGN**	CHAR_ALIGN
SHRT_SIZE	2	**SHRT_ALIGN**	2^1
USHRT_SIZE	SHRT_SIZE	**USHRT_ALIGN**	SHRT_ALIGN
INT_SIZE	4	**INT_ALIGN**	2^2
UINT_SIZE	INT_SIZE	**UINT_ALIGN**	INT_ALIGN
LONG_SIZE	4	**LONG_ALIGN**	2^2
ULONG_SIZE	LONG_SIZE	**ULONG_ALIGN**	LONG_ALIGN
LLONG_SIZE	8	**LLONG_ALIGN**	2^3
ULLONG_SIZE	LLONG_SIZE	**ULLONG_ALIGN**	LLONG_ALIGN
PTR_SIZE	INT_SIZE	**PTR_ALIGN**	INT_ALIGN
		MAX_ALIGN	LLONG_ALIGN

1.5.2 Representation

Values stored in objects consist of $n \times$ **`CHAR_BIT`** bits, where n is the size of the object in bytes. The value may be copied into an object of type `unsigned char [n]` and the resulting set of bytes is called the *object representation* of the value. Two values with the same object representation compare equal, but the converse does not necessarily hold.

Padding bytes in structure types take on unspecified values.

Signed integer types use two's complement representation for negative values[11].

1.5.3 Compatible types

Two types have *compatible type* if any of the following hold:

- the types are the same
- both are structure types such that
  - both either have no tag or the same tag
  - there is a one-to-one correspondence between their members such that each pair of members are declared with compatible types and the same name

[11]this is a consequence of the underlying implementation using JavaScript's `BigInt` type, which "act as two's complement binary strings" for binary operations

  - the corresponding members shall be declared in the same order
- both are array types such that their element types are compatible and their lengths are equal
- both are pointer types such that their referenced types are compatible
- both are function types such that
  - both specify compatible return types
  - both have the same arity
  - corresponding parameters shall have compatible types

1.6 Conversions

Where an expression is used in a context where a value of different type is expected, *conversion* may occur.

```
int n = 1L;  // expression 1L has type long, but int is expected
char *p = malloc(10);  // expression malloc(10) has type void*, but char* is expected
```

1.6.1 Conversions as if by assignment

Provided that the simple assignment constraints hold, the following conversions happen:

- In the assignment operator, the value of the right hand operand is converted to the type of the left hand operand.
- When initializing an object of scalar type, the value of the initializer expression is converted to the type of the object being initialized.
- In a function call expression, the value of each argument expression is converted to the type of the corresponding parameter.
- In a return statement, the value of the `return` expression is converted to an object having the return type of the function.

1.6.2 Usual arithmetic conversions

The operands of the following operators undergo implicit conversions to obtain a common type, in which the calculation is performed:

- multiplicative operators: `*, /, %`
- additive operators: `+, -`
- relational operators: `<, >, <=, >=`
- equality operators: `==, !=`
- bitwise AND operator: `&`
- bitwise exclusive OR operator: `^`
- bitwise inclusive OR operator: `|`
- conditional operator: `?:`

The conversion proceeds as follows:

- Both operands undergo integer promotions.
- If the types are the same, we are done.

- Else the types are different:
  - If the types have the same signedness, the operand whose type has the smaller *conversion rank*[12] is implicitly converted[13] to the other type.
  - Else the operands have different signedness:
    - If the unsigned type has *conversion rank* greater than or equal to the rank of the signed type, then the operand with the signed type is implicitly converted to the unsigned type.
    - Else the unsigned type has *conversion rank* less than the signed type:
      - If the signed type can represent all values of the unsigned type, then the operand with the unsigned type is implicitly converted to the signed type.
      - Else both operands undergo implicit conversion to the unsigned type counterpart of the signed operand's type.

1.6.3 Value categories

Every expression in C is characterized by two independent properties: a type and a value category.

Every expression belongs to one of two value categories: lvalue and rvalue.

Lvalue expressions

A *lvalue* expression is any expression with object type which designates an object.

Lvalue expressions can be used in the following contexts:

- as the operand of the address-of operator
- as the operand of the pre/post increment and decrement operators
- as the left-hand operand of the member access (dot) operator
- as the left-hand operand of the assignment operators

If a lvalue expression is used in any context other than `sizeof` or the operators listed above, non-array lvalues undergo lvalue conversion, which loads the value of the object from memory.

The following expressions are lvalues:

- identifiers that do not designate a function
- string literals
- parenthesized expressions (x) if the unparanthesized expression x is a lvalue
- the result of a member access (dot) operator if the left-hand operand is a lvalue
- the result of a member access through pointer `->` operator
- the result of the indirection `*` operator applied to a pointer to object
- the result of the subscript `[]` operator

Modifiable lvalue expressions

A *modifiable lvalue* is any lvalue expression of non-array type.

Only modifiable lvalue expressions may be used as the left-hand operand of assignment operators and as operands of increment/decrement operators.

---

[12] see Integer promotions
[13] see Integer conversions

Rvalue expressions

A *rvalue* expression is defined by exclusion: an expression is a *rvalue* if it is not a lvalue. The address of a rvalue expression cannot be taken.

The following expressions are rvalues:

— integer and character constants

— all operators not specified to return lvalues, such as

  - function call expressions
  - cast expressions
  - member access (dot) operator applied to a non-lvalue structure
  - results of all arithmetic, relational, logical and bitwise operators
  - results of increment/decrement operators
  - results of assignment operators[14]
  - conditional operator
  - comma operator
  - address-of operator

1.6.4 Value transformations

Lvalue conversion

Any lvalue expression of non-array type when used in any context other than

— as the operand of the address-of operator

— as the operand of the pre/post increment and decrement operators

— as the left-hand operand of the member access (dot) operator

— as the left-hand operand of the assignment operators

— as the operand of `sizeof`

undergoes *lvalue conversion*: the type and value remains the same but it loses its lvalue properties. Specifically, the address may no longer be taken.

This conversion models the memory load of the object.

```
int x = n;  // lvalue conversion on n, loads object from memory
int *p = &n;  // no lvalue conversion, does not read from memory
```

Array to pointer conversion

Any lvalue expression of array type when used in any context other than

— as the operand of the address-of operator

— as the operand of the `sizeof`

undergoes a conversion to the non-lvalue pointer to its first element.

```
int a[3], b[3][4];
int *p = a;  // conversion to &a[0]
int (*q)[4] = b;  // conversion to &b[0]
```

[14]lvalue in C++

Function to pointer conversion

Any function designator expression when used in any context other than

- as the operand of the address-of operator
- as the operand of the `sizeof`

undergoes a conversion to the non-lvalue pointer to the function designated by the expression.

```
void f() { return; }
void (*p)() = f; // conversion to &f
(***p)(); // repeated conversion to &f and dereference to f
```

1.6.5 Implicit conversions

Implicit conversion, whether as if by assignment or a usual arithmetic conversion, consists of:

1. value transformation, if applicable
2. one of the conversions listed below, if it produces the target type

Compatible types

Conversion of a value of any type to any compatible type is a no-op.

Integer promotions

Integer promotion is the implicit conversion of a value of any integer type with *rank* less than or equal to *rank* of int to the value of type `int` or `unsigned int`.

If `int` can represent the entire range of values of the original type, the value is converted to type `int`. Else, the value is converted to `unsigned int`.

Integer promotions preserve the value, including the sign.

Rank is a property defined for all integer types:

- The ranks of all signed integer types are different and increases with their precision.
- The ranks of all signed integer types equal the ranks of their unsigned counterparts.
- The rank of `char` is equal to the rank of `signed char` and `unsigned char`.
- The rank of `_Bool` is less than the rank of any other standard integer type.
- Ranking is transitive.

The specific values for ranks used by `c-viz` are given in the following table:

Integer Type	Rank
`_Bool`	0
`Char`	1
`SignedChar`	1
`UnsignedChar`	1
`ShortInt`	2
`UnsignedShortInt`	2

Integer Type	Rank
`Int`	3
`UnsignedInt`	3
`LongInt`	4
`UnsignedLongInt`	4
`LongLongInt`	5
`UnsignedLongLongInt`	5

Integer promotions are applied in the following contexts:

- as part of *usual arithmetic conversions*
- to the operand of the unary arithmetic operators + and -
- to the operand of the unary bitwise operator ~
- to both operands of the shift operators << and >>

Boolean conversions

A value of any scalar type can be implicitly converted to `_Bool`. Values that compare equal to zero are converted to `0` and all other values are converted to `1`.

Integer conversions

A value of any integer type can be implicitly converted to any other integer type. Except where covered by promotions and boolean conversions above, the following rules apply:

- if the target type can represent the value, the value is unchanged
- if the target type is unsigned, modulo arithmetic is applied such that the result fits in the target type
- if the target type is signed, the interpreter throws an error

```
char x = 1;  // int converted to char, result unchanged
unsigned char n = -123456;  // target is unsigned, result is 192
signed char m = 123456;  // target is signed, error is thrown
sizeof(int) > -1;  // false: operator > applies usual arithmetic conversion
                   // target type is unsigned, -1 becomes UINT_MAX
```

Pointer conversions

A pointer to `void` can be implicitly converted to and from any pointer to object type. If a pointer to object is converted to a pointer to `void` and back, its value compares equal to the original pointer.

```
int *p = malloc(10 * sizeof(int));  // malloc returns void*
```

The integer constant expression `0` can be implicitly converted to any pointer type. The result is a null pointer value of its type, guaranteed to compare unequal to any non-null pointer value of that type.

```
int *p = 0;
```

2 Lexical Grammar

Syntax

Token	::=	Keyword | Identifier | Constant	token
	|	StringLiteral | Punctuator	
_	::=	{ WhiteSpace | LongComment | LineComment }	token separator
WhiteSpace	::=	'␣' | '\n' | '\r' | '\t' | '\u000b' | '\f'	

Semantics

A *token* is the minimal lexical element of the language.

Each token can be separated by *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), comments (subsection 2.7 – *Comments*), or both.

2.1 Keywords

Syntax

Keyword	::=	'auto' | 'break' | 'case' | 'char' | 'const' | 'continue' | 'default' | 'do' | 'double' | 'else' | 'enum' | 'extern' | 'float' | 'for' | 'goto' | 'if' | 'inline' | 'int' | 'long' | 'register' | 'restrict' | 'return' | 'short' | 'signed' | 'sizeof' | 'static' | 'struct' | 'switch' | 'typedef' | 'union' | 'unsigned' | 'void' | 'volatile' | 'while' | '_Alignas' | '_Alignof' | '_Atomic' | '_Bool' | '_Complex' | '_Generic' | '_Imaginary' | '_Noreturn' | '_Static_assert' | '_Thread_local'	keyword

Semantics

The above tokens (case sensitive) are reserved for use as keywords, and shall not be used otherwise (such as for identifiers).

2.2 Identifiers

Syntax

Identifier	::=	IdentifierNondigit , { IdentifierNondigit | Digit }	identifier
IdentifierNondigit	::=	Nondigit | UniversalCharacterName	
Nondigit	::=	*? lowercase or uppercase alphabet ?* | '_'	
Digit	::=	*? digit from 0 to 9 ?*	

Semantics

An identifier is a sequence of nondigit characters (which includes the underscore _, the lower and upper case Latin letters, and other characters) and digits, which designates one or more entities as described in subsection 1.2 – *Scopes of identifiers*.

2.3 Universal character names

Syntax

UniversalCharacterName	::=	`'\u'` , HexQuad	universal character name
	|	`'\U'` , HexQuad , HexQuad	
HexQuad	::=	HexadecimalDigit , HexadecimalDigit , HexadecimalDigit , HexadecimalDigit	

Semantics

Universal character names are used in identifiers, character constants and string literals to designate characters that are not in the basic character set.

2.4 Constants

Syntax

Constant	::=	IntegerConstant | CharacterConstant	constant

Semantics

Each constant has a type, determined by its form and value as described later.

2.4.1 Integer constants

Syntax

IntegerConstant	::=	(DecimalConstant | HexadecimalConstant | OctalConstant) , [IntegerSuffix]	integer constant
DecimalConstant	::=	NonzeroDigit , { Digit }	
OctalConstant	::=	`'0'` , { OctalDigit }	
HexadecimalConstant	::=	HexadecimalPrefix , HexadecimalDigit , { HexadecimalDigit }	
HexadecimalPrefix	::=	`'0x'` | `'0X'`	
NonzeroDigit	::=	*? digit from 1 to 9 ?*	
OctalDigit	::=	*? digit from 0 to 7 ?*	
HexadecimalDigit	::=	*? digit from 0 to 9 ?*	
	|	*? lowercase or uppercase alphabet from a to f ?*	
IntegerSuffix	::=	UnsignedSuffix , [LongLongSuffix | LongSuffix]	
	|	(LongLongSuffix | LongSuffix) , [UnsignedSuffix]	
UnsignedSuffix	::=	`'u'` | `'U'`	
LongSuffix	::=	`'l'` | `'L'`	
LongLongSuffix	::=	`'ll'` | `'LL'`	

Semantics

An integer constant may have a prefix specifying its base and a suffix specifying its type.

The type of an integer constant is the first of the corresponding list in which its value can be represented:

Suffix	Decimal Constant	Octal or Hexadecimal Constant

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

u or U	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

l or L	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both u or U and l or L	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
ll or LL	`long long int`	`long long int` `unsigned long long int`
Both u or U and ll or LL	`unsigned long long int`	`unsigned long long int`

2.4.2 Character constants

Syntax

CharacterConstant	::=	`'''` , CChar , `'''`	character constant
CChar	::=	EscapeSequence	
	|	*? UTF-16 char ?* - (`'''` | `'\n'` | `'\'`)	
EscapeSequence	::=	SimpleEscapeSequence | OctalEscapeSequence | HexadecimalEscapeSequence | UniversalCharacterName	
SimpleEscapeSequence	::=	`'\'` , (`'''` | `'"'` | `'?'` | `'\'` | `'a'` | `'b'` | `'f'` | `'n'` | `'r'` | `'t'` | `'v'`)	
OctalEscapeSequence	::=	`'\'` , OctalDigit , [OctalDigit] , [OctalDigit]	
HexadecimalEscapeSequence	::=	`'\x'` , HexadecimalDigit , { HexadecimalDigit }	

2.5 String literals

Syntax

StringLiteral	::=	`'"'` , [SCharSequence] , `'"'`	string literal
SCharSequence	::=	SChar , { SChar }	
SChar	::=	EscapeSequence	
	|	*? UTF-16 char ?* - (`'"'` | `'\n'` | `'\'`)	

2.6 Punctuators

Syntax

```
Punctuator ::= '[' | ']' | '(' | ')' | '{' | '}' | '.' | '->' |          punctuator
               '++' | '--' | '&' | '*' | '+' | '-' | '~' | '!' |
               '/' | '%' | '<<' | '>>' | '<' | '>' | '<=' | '>=' |
               '==' | '!=' | '^' | '|' | '&&' | '||' | '?' | ':' |
               ';' | '...' | '=' | '*=' | '/=' | '%=' | '+=' | '-='
               | '<<=' | '>>=' | '&=' | '^=' | '|=' | ',' | '#' |
               '##' | '<:' | ':>' | '<%' | '%>' | '%:' | '%:%:'
```

2.7 Comments

Syntax

```
LongComment ::= '/*' , { ? UTF-16 char ? - '*/' } , '*/'     comment
LineComment ::= '//' , { ? UTF-16 char ? - '\n' }            comment
```

3 Expressions

3.1 Primary expressions

Syntax

```
PrimaryExpression ::= Identifier                              primary expr
                    | Constant
                    | StringLiteral
                    | '(' , Expression , ')'                  parenthesized expr
```

3.2 Postfix operators

Syntax

```
PostfixExpression      ::= PrimaryExpression , { PostfixOp }                postfix operators
PostfixOp              ::= '[' , Expression , ']'                           array subscripting
                         | '(' , [ ArgumentExpressionList ] , ')'           function call
                         | '.' , Identifier                                 structure member
                         | '->' , Identifier                                structure member
                         | '++' | '--'                                      postfix incr/decr
ArgumentExpressionList ::= AssignmentExpression ,
                           { ',' , AssignmentExpression }
```

3.2.1 Array subscripting

Constraints

One of the expressions shall have type "pointer to object *type*", the other shall have integer type, and the result has type "*type*".

Semantics

The definition of the subscript operator `[]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`. As a consequence of the conversion rules that apply to the `+` operator, if `E1` is an array object (equivalently, a pointer to the first element of an array) and `E2` is an integer, `E1[E2]` designates the `E2`-th element of `E1` (0-indexed).

3.2.2 Function calls

3.2.3 Structure members

3.2.4 Postfix increment and decrement operators

3.3 Unary operators

Syntax

```
UnaryExpression ::= PostfixExpression
                  | '++' , UnaryExpression                        prefix incr
                  | '--' , UnaryExpression                        prefix decr
                  | UnaryOperator , CastExpression                unary operators
                  | 'sizeof' , '(' , TypeName , ')'               sizeof operator
                  | 'sizeof' , UnaryExpression                    sizeof operator
UnaryOperator   ::= '&'                                           address operator
                  | '*'                                           indirection operator
                  | '+' | '-' | '~' | '!'                         unary arithmetic operators
```

3.3.1 Prefix increment and decrement operators

3.3.2 Address and indirection operators

3.3.3 Unary arithmetic operators

3.3.4 The sizeof operator

3.4 Cast operators

Syntax

```
CastExpression ::= UnaryExpression
                 | '(' , TypeName , ')' , CastExpression          cast operator
```

3.5 Multiplicative operators

Syntax

```
MultiplicativeExpression ::= CastExpression ,                     multiplicative operators
                             { ( '*' | '/' | '%' ) , CastExpression }
```

3.6 Additive operators

Syntax

```
AdditiveExpression ::= MultiplicativeExpression ,                 additive operators
                       { ( '+' | '-' ) , MultiplicativeExpression }
```

3.7 Bitwise shift operators

Syntax

```
ShiftExpression ::= AdditiveExpression ,                          bitwise shift operators
                    { ( '<<' | '>>' ) , AdditiveExpression }
```

3.8 Relational operators

Syntax

```
RelationalExpression ::= ShiftExpression ,                        relational operators
                         { ( '<' | '>' | '<=' | '>=' ) , ShiftExpression }
```

3.9 Equality operators

Syntax

```
EqualityExpression ::= RelationalExpression ,                     equality operators
                       { ( '==' | '!=' ) , RelationalExpression }
```

3.10 Bitwise AND operator

Syntax

```
ANDExpression ::= EqualityExpression , { '&' , EqualityExpression }    bitwise AND operator
```

3.11 Bitwise exclusive OR operator

Syntax

```
ExclusiveORExpression ::= ANDExpression , { '^' , ANDExpression }      bitwise XOR operator
```

3.12 Bitwise inclusive OR operator

Syntax

InclusiveORExpression	::=	ExclusiveORExpression , { '|' , ExclusiveORExpression }	bitwise OR operator

3.13 Logical AND operator

Syntax

LogicalANDExpression	::=	InclusiveORExpression , { '&&' , InclusiveORExpression }	logical AND operator

3.14 Logical OR operator

Syntax

LogicalORExpression	::=	LogicalANDExpression , { '||' , LogicalANDExpression }	logical OR operator

3.15 Conditional operator

Syntax

ConditionalExpression	::=	LogicalORExpression , ['?' , Expression , ':' , ConditionalExpression]	conditional operator

3.16 Assignment operators

Syntax

AssignmentExpression	::=	UnaryExpression , AssignmentOperator , AssignmentExpression	assignment operators
	|	ConditionalExpression	
AssignmentOperator	::=	'='	simple assignment
	|	'*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' | '^=' | '|='	compound assignment

3.16.1 Simple assignment

3.16.2 Compound assignment

3.17 Comma operator

Syntax

Expression	::=	AssignmentExpression , { ',' , AssignmentExpression }	comma operator

4 Declarations

Syntax

Declaration	::=	DeclarationSpecifiers , [InitDeclaratorList] , ';'	declaration
DeclarationSpecifiers	::=	(StorageClassSpecifier | TypeSpecifier | TypedefName) , { StorageClassSpecifier | TypeSpecifier | TypedefName }	
InitDeclaratorList	::=	InitDeclarator , { ',' , InitDeclarator }	
InitDeclarator	::=	Declarator , ['=' , Initializer]	

4.1 Storage-class specifiers

Syntax

StorageClassSpecifier	::=	'typedef'	storage class specifier

4.2 Type specifiers

Syntax

TypeSpecifier	::=	'void' | 'char' | 'short' | 'int' | 'long' | 'signed'	type specifiers
	|	'unsigned' | '_Bool' | StructOrUnionSpecifier	

4.2.1 Structure specifiers

Syntax

StructOrUnionSpecifier	::=	'struct' , [Identifier] , '{' , StructDeclarationList , '}'	structure specifier
	|	'struct' , Identifier	tags
StructDeclarationList	::=	StructDeclaration , { StructDeclaration }	
StructDeclaration	::=	SpecifierQualifierList , [StructDeclaratorList] , ';'	
SpecifierQualifierList	::=	(TypeSpecifier | TypedefName) , { TypeSpecifier | TypedefName }	
StructDeclaratorList	::=	StructDeclarator , { ',' , StructDeclarator }	
StructDeclarator	::=	Declarator	

4.2.2 Tags

4.3 Declarators

Syntax

Declarator	::=	[Pointer] , DirectDeclarator	declarator
DirectDeclarator	::=	(Identifier | ('(' , Declarator , ')')) , { DirectDeclaratorPart }	
DirectDeclaratorPart	::=	'[' , [IntegerConstant] , ']'	array declarator
	|	'(' , [ParameterList] , ')'	function declarator
Pointer	::=	'*' , { '*' }	pointer declarator
ParameterList	::=	ParameterDeclaration , { ',' , ParameterDeclaration }	
ParameterDeclaration	::=	DeclarationSpecifiers , [Declarator | AbstractDeclarator]	

4.3.1 Pointer declarators

4.3.2 Array declarators

4.3.3 Function declarators

4.4 Type names

Syntax

```
TypeName                 ::= SpecifierQualifierList , [ AbstractDeclarator ]        type name
AbstractDeclarator       ::= [ Pointer ] , DirectAbstractDeclarator
                           | Pointer
DirectAbstractDeclarator ::= '(' , AbstractDeclarator , ')'
                           | [ '(' , AbstractDeclarator , ')' ] ,
                             ( '[' , [ IntegerConstant ] , ']' |
                             '(' , [ ParameterList ] , ')' ) ,
                             { '[' , [ IntegerConstant ] , ']' |
                             '(' , [ ParameterList ] , ')' }
```

4.5 Type definitions

Syntax

```
TypedefName ::= Identifier                                                  type definition
```

4.6 Initialization

Syntax

```
Initializer     ::= AssignmentExpression                                    initialization
                  | '{' , InitializerList , [ ',' ] , '}'
InitializerList ::= [ Designation ] , Initializer ,
                    { ',' , [ Designation ] , Initializer }
Designation     ::= DesignatorList , '='
DesignatorList  ::= Designator , { Designator }
Designator      ::= '[' , IntegerConstant , ']'
                  | '.' , Identifier
```

5 Statements and blocks

Syntax

```
Statement ::= CompoundStatement                      compound stmt
            | ExpressionStatement                    expression stmt
            | SelectionStatement                     selection stmt
            | IterationStatement                     iteration stmt
            | JumpStatement                          jump stmt
```

5.1 Compound statement

Syntax

```
CompoundStatement ::= '{' , [ BlockItemList ] , '}'   compound stmt
    BlockItemList ::= BlockItem , { BlockItem }
        BlockItem ::= Statement
                    | Declaration
```

5.2 Expression and null statements

Syntax

```
ExpressionStatement ::= [ Expression ] , ';'          expression stmt
```

5.3 Selection statements

Syntax

```
SelectionStatement ::= 'if' , '(' , Expression , ')' , Statement ,     if stmt
                       [ 'else' , Statement ]
```

5.3.1 The if statement

5.4 Iteration statements

Syntax

```
IterationStatement ::= 'while' , '(' , Expression , ')' , Statement     while stmt
                     | 'do' , Statement ,                                do stmt
                       'while' , '(' , Expression , ')' , ';'
                     | 'for' , '(' , [ Expression ] , ';' ,              for stmt
                                     [ Expression ] , ';' ,
                                     [ Expression ] , ')' , Statement
```

5.4.1 The while statement

5.4.2 The do statement

5.4.3 The for statement

5.5 Jump statements

Syntax

```
JumpStatement ::= 'continue' , ';'                        continue stmt
                | 'break' , ';'                           break stmt
                | 'return' , [ Expression ] , ';'         return stmt
```

5.5.1 The `continue` statement

5.5.2 The `break` statement

5.5.3 The `return` statement

6 External Definitions

Syntax

```
TranslationUnit      ::= ExternalDeclaration , { ExternalDeclaration }     external definitions
ExternalDeclaration  ::= FunctionDefinition
                       | Declaration
```

Constraints

Every identifier declared in a translation unit can have at most one external definition. Moreover, if such an identifier is used in an expression, then there shall be exactly one external definition for the identifier in the translation unit.

A translation unit must define a `main` function. The `main` function shall be defined with no parameters[15] and a return type of `int`.

Forward declaration of functions are not supported. Functions must be defined prior to any code that references them. On the other hand, forward declaration of `struct`s, such as to construct self-referential types, are supported in file scope.

```
void f(int); // ERROR: forward declaration of a function
void f(int a) {}

struct node {
  int value;
  struct node *next; // OK: forward declaration of struct node
};
```

Semantics

A translation unit is a sequence of external declarations. They are described as “external” as they appear outside any function. An external definition is an external declaration that is also a definition[16] of a function or an object.

6.1 Function definitions

Syntax

```
FunctionDefinition  ::= DeclarationSpecifiers , Declarator ,              function definitions
                        CompoundStatement
DeclarationList     ::= Declaration , { Declaration }
```

Constraints

The identifier declared in a function definition shall have function type.

The return type of a function shall be `void` or an object type other than array type.

The declaration specifiers shall not contain any storage-class specifiers.

The declaration of each parameter shall include an identifier, other than the special case where there is only a single parameter of type `void` to denote a function that has no parameters.

[15] `argc` and `argv` are not supported

[16] a declaration that causes memory to be reserved for an object or function

Semantics

The identifier for each parameter is an lvalue and is effectively declared at the start of the compound statement constituting the function body. Therefore, these identifiers may not be redeclared in the function body except for in an enclosing block.

The layout of the storage for parameters is specified as such: the leftmost argument has address equal to the starting address of the stack frame, and memory allocation proceeds left to right with minimum padding added to ensure alignment.

On entry to the function, the value of each argument expression is converted to the type of the corresponding parameter as if by assignment.

After all parameters have been assigned, the function body is executed.

If the } that terminates a function is reached, the interpreter throws an error[17]. The only exception is in the case of the `main` function, in which case a value of `0` is returned.

6.2 External object definitions

Semantics

An external definition for an identifier is a declaration with initializer for an object with file scope.

Each identifier may only be declared once. *Tentative definitions* are not supported.

A declaration without an initializer for an object with file scope behaves as if an initializer exists and is equal to `0`.

[17] functions returning `void` need to include a `return` statement as well

7 Memory Model

The memory available to a `c-viz` program is a contiguous sequence of bytes[18], each of which has a unique 32-bit address. The lowest address is `0`, and the highest address is **`INT_MAX`**.

7.1 Memory segmentation

The memory is divided into four non-overlapping segments: stack, heap, data, and text.

Attempting to access an address that falls outside of any of the segments causes a *segmentation fault* to be thrown.

Misaligned memory access throws an error.

7.2 Stack allocations

Unlike typical x86 architectures, stack memory grows upwards (addresses are allocated in increasing order).

The size of a stack frame is calculated by recursively scanning out all declarations made in the function body (and any blocks within).

No control-flow analysis is done to reduce the size of stack frames needed. As an example, consider the following code

```
if (...) {
  // block A
} else (...) {
  // block B
}
```

Let S_A and S_B denote the size of blocks A and B respectively. Then the resulting stack frame will be of size $S_F = p_0 + S_A + p_1 + S_B$ where p_i are sizes of padding bytes. A smarter allocation algorithm will only require $S_F = \max\{p_0 + S_A, p_1 + S_B\}$ bytes.

7.3 Heap allocations

The order and contiguity of storage allocated by successive calls to the `malloc` function is specified by a first-fit allocation algorithm.

The pointer returned by a successful allocation is suitably aligned so that it may be assigned to a pointer to any type of object. Specifically, the pointer contains an address with alignment **`MAX_ALIGN`**[19].

The lifetime of an allocated object extends from the allocation until the deallocation.

Pointers returned are guaranteed to point to disjoint objects and contain the starting address of the allocated memory. If allocation fails, a null pointer is returned.

If the size of memory requested is a non-positive integer, a null pointer is returned.

[18] backed by JavaScript's `ArrayBuffer`
[19] see Size and alignment

7.3.1 The `malloc` function

```
void *malloc(int size);
```

The `malloc` function allocates `size` bytes of memory of indeterminate value.

The `malloc` function returns a pointer to the allocated memory if allocation was successful, and a null pointer otherwise.

7.3.2 The `free` function

```
void free(void *ptr);
```

The `free` function deallocates the memory pointed to by `ptr`.

If the `ptr` argument[20] does not match a pointer earlier returned by the `malloc` function, the interpreter throws a `"free on address not returned by malloc"` error.

7.4 Text and data

The text segment is read-only and is the only segment that can be executed.

7.5 Additional runtime data structures

7.5.1 `EffectiveTypeTable`

7.5.2 `InitializedTable`

[20] including the case where `ptr` is a null pointer

8 Operational Semantics

8.1 Runtime

8.1.1 Control

8.1.2 Stash

8.1.3 Memory

8.1.4 Symbol table

8.2 Instructions

8.3 Semantics

A Language Syntax Summary

Refer to section – *Introduction* for an explanation of the notation used.

A.1 Lexical Grammar

Token ::= Keyword | Identifier | Constant — token
| StringLiteral | Punctuator

_ ::= { WhiteSpace | LongComment | LineComment } — token separator

WhiteSpace ::= '␣' | '\n' | '\r' | '\t' | '\u000b' | '\f'

LongComment ::= '/*' , { *? UTF-16 char ?* - '*/' } , '*/' — comment

LineComment ::= '//' , { *? UTF-16 char ?* - '\n' } — comment

Keyword ::= 'auto' | 'break' | 'case' | 'char' | 'const' | 'continue' | 'default' | 'do' | 'double' | 'else' | 'enum' | 'extern' | 'float' | 'for' | 'goto' | 'if' | 'inline' | 'int' | 'long' | 'register' | 'restrict' | 'return' | 'short' | 'signed' | 'sizeof' | 'static' | 'struct' | 'switch' | 'typedef' | 'union' | 'unsigned' | 'void' | 'volatile' | 'while' | '_Alignas' | '_Alignof' | '_Atomic' | '_Bool' | '_Complex' | '_Generic' | '_Imaginary' | '_Noreturn' | '_Static_assert' | '_Thread_local' — keyword

Identifier ::= IdentifierNondigit , { IdentifierNondigit | Digit } — identifier

IdentifierNondigit ::= Nondigit | UniversalCharacterName

Nondigit ::= *? lowercase or uppercase alphabet ?* | '_'

Digit ::= *? digit from 0 to 9 ?*

UniversalCharacterName ::= '\u' , HexQuad — universal character name
| '\U' , HexQuad , HexQuad

HexQuad ::= HexadecimalDigit , HexadecimalDigit , HexadecimalDigit , HexadecimalDigit

Constant ::= IntegerConstant | CharacterConstant — constant

IntegerConstant ::= (DecimalConstant | HexadecimalConstant | OctalConstant) , [IntegerSuffix] — integer constant

DecimalConstant ::= NonzeroDigit , { Digit }

OctalConstant ::= '0' , { OctalDigit }

HexadecimalConstant ::= HexadecimalPrefix , HexadecimalDigit , { HexadecimalDigit }

HexadecimalPrefix ::= '0x' | '0X'

NonzeroDigit ::= *? digit from 1 to 9 ?*

OctalDigit ::= *? digit from 0 to 7 ?*

HexadecimalDigit ::= *? digit from 0 to 9 ?*
| *? lowercase or uppercase alphabet from a to f ?*

IntegerSuffix ::= UnsignedSuffix , [LongLongSuffix | LongSuffix]
| (LongLongSuffix | LongSuffix) , [UnsignedSuffix]

UnsignedSuffix ::= 'u' | 'U'

LongSuffix ::= 'l' | 'L'

LongLongSuffix ::= 'll' | 'LL'

```
CharacterConstant ::= ''' , CChar , '''                                   character constant
CChar ::= EscapeSequence
        | ? UTF-16 char ? - ( ''' | '\n' | '\' )
EscapeSequence ::= SimpleEscapeSequence | OctalEscapeSequence |
                   HexadecimalEscapeSequence | UniversalCharacterName
SimpleEscapeSequence ::= '\' , ( ''' | '"' | '?' | '\' | 'a' | 'b' | 'f' |
                         'n' | 'r' | 't' | 'v' )
OctalEscapeSequence ::= '\' , OctalDigit , [ OctalDigit ] , [ OctalDigit ]
HexadecimalEscapeSequence ::= '\x' , HexadecimalDigit , { HexadecimalDigit }
StringLiteral ::= '"' , [ SCharSequence ] , '"'                           string literal
SCharSequence ::= SChar , { SChar }
SChar ::= EscapeSequence
        | ? UTF-16 char ? - ( '"' | '\n' | '\' )
Punctuator ::= '[' | ']' | '(' | ')' | '{' | '}' | '.' | '->' |           punctuator
               '++' | '--' | '&' | '*' | '+' | '-' | '~' | '!' |
               '/' | '%' | '<<' | '>>' | '<' | '>' | '<=' | '>=' |
               '==' | '!=' | '^' | '|' | '&&' | '||' | '?' | ':' |
               ';' | '...' | '=' | '*=' | '/=' | '%=' | '+=' | '-='
               | '<<=' | '>>=' | '&=' | '^=' | '|=' | ',' | '#' |
               '##' | '<:' | ':>' | '<%' | '%>' | '%:' | '%:%:'
```

A.2 Expressions

```
PrimaryExpression ::= Identifier                                          primary expr
                    | Constant
                    | StringLiteral
                    | '(' , Expression , ')'                              parenthesized expr
PostfixExpression ::= PrimaryExpression , { PostfixOp }                   postfix operators
PostfixOp ::= '[' , Expression , ']'                                      array subscripting
            | '(' , [ ArgumentExpressionList ] , ')'                      function call
            | '.' , Identifier                                            structure member
            | '->' , Identifier                                           structure member
            | '++' | '--'                                                 postfix incr/decr
ArgumentExpressionList ::= AssignmentExpression ,
                           { ',' , AssignmentExpression }
UnaryExpression ::= PostfixExpression
                  | '++' , UnaryExpression                                prefix incr
                  | '--' , UnaryExpression                                prefix decr
                  | UnaryOperator , CastExpression                        unary operators
                  | 'sizeof' , '(' , TypeName , ')'                       sizeof operator
                  | 'sizeof' , UnaryExpression                            sizeof operator
UnaryOperator ::= '&'                                                     address operator
                | '*'                                                     indirection operator
                | '+' | '-' | '~' | '!'                                   unary arithmetic operators
CastExpression ::= UnaryExpression
                 | '(' , TypeName , ')' , CastExpression                  cast operator
MultiplicativeExpression ::= CastExpression ,                             multiplicative operators
                             { ( '*' | '/' | '%' ) , CastExpression }
```

AdditiveExpression	::=	MultiplicativeExpression , { ('+' | '-') , MultiplicativeExpression }	additive operators
ShiftExpression	::=	AdditiveExpression , { ('<<' | '>>') , AdditiveExpression }	bitwise shift operators
RelationalExpression	::=	ShiftExpression , { ('<' | '>' | '<=' | '>=') , ShiftExpression }	relational operators
EqualityExpression	::=	RelationalExpression , { ('==' | '!=') , RelationalExpression }	equality operators
ANDExpression	::=	EqualityExpression , { '&' , EqualityExpression }	bitwise AND operator
ExclusiveORExpression	::=	ANDExpression , { '^' , ANDExpression }	bitwise XOR operator
InclusiveORExpression	::=	ExclusiveORExpression , { '|' , ExclusiveORExpression }	bitwise OR operator
LogicalANDExpression	::=	InclusiveORExpression , { '&&' , InclusiveORExpression }	logical AND operator
LogicalORExpression	::=	LogicalANDExpression , { '||' , LogicalANDExpression }	logical OR operator
ConditionalExpression	::=	LogicalORExpression , ['?' , Expression , ':' , ConditionalExpression]	conditional operator
AssignmentExpression	::=	UnaryExpression , AssignmentOperator , AssignmentExpression	assignment operators
	|	ConditionalExpression	
AssignmentOperator	::=	'='	simple assignment
	|	'*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' | '^=' | '|='	compound assignment
Expression	::=	AssignmentExpression , { ',' , AssignmentExpression }	comma operator

A.3 Declarations

Declaration	::=	DeclarationSpecifiers , [InitDeclaratorList] , ';'	declaration
DeclarationSpecifiers	::=	(StorageClassSpecifier | TypeSpecifier | TypedefName) , { StorageClassSpecifier | TypeSpecifier | TypedefName }	
InitDeclaratorList	::=	InitDeclarator , { ',' , InitDeclarator }	
InitDeclarator	::=	Declarator , ['=' , Initializer]	
StorageClassSpecifier	::=	'typedef'	storage class specifier
TypeSpecifier	::=	'void' | 'char' | 'short' | 'int' | 'long' | 'signed'	type specifiers
	|	'unsigned' | '_Bool' | StructOrUnionSpecifier	
StructOrUnionSpecifier	::=	'struct' , [Identifier] , '{' , StructDeclarationList , '}'	structure specifier
	|	'struct' , Identifier	tags
StructDeclarationList	::=	StructDeclaration , { StructDeclaration }	
StructDeclaration	::=	SpecifierQualifierList , [StructDeclaratorList] , ';'	
SpecifierQualifierList	::=	(TypeSpecifier | TypedefName) , { TypeSpecifier | TypedefName }	
StructDeclaratorList	::=	StructDeclarator , { ',' , StructDeclarator }	
StructDeclarator	::=	Declarator	

```
               Declarator  ::= [ Pointer ] , DirectDeclarator                                declarator
         DirectDeclarator  ::= ( Identifier | ( '(' , Declarator , ')' ) ) ,
                               { DirectDeclaratorPart }
     DirectDeclaratorPart  ::= '[' , [ IntegerConstant ] , ']'                               array declarator
                            |  '(' , [ ParameterList ] , ')'                                 function declarator
                  Pointer  ::= '*' , { '*' }                                                 pointer declarator
            ParameterList  ::= ParameterDeclaration ,
                               { ',' , ParameterDeclaration }
     ParameterDeclaration  ::= DeclarationSpecifiers ,
                               [ Declarator | AbstractDeclarator ]
                 TypeName  ::= SpecifierQualifierList , [ AbstractDeclarator ]               type name
       AbstractDeclarator  ::= [ Pointer ] , DirectAbstractDeclarator
                            |  Pointer
 DirectAbstractDeclarator  ::= '(' , AbstractDeclarator , ')'
                            |  [ '(' , AbstractDeclarator , ')' ] ,
                               ( '[' , [ IntegerConstant ] , ']' |
                               '(' , [ ParameterList ] , ')' ) ,
                               { '[' , [ IntegerConstant ] , ']' |
                               '(' , [ ParameterList ] , ')' }
              TypedefName  ::= Identifier                                                    type definition
              Initializer  ::= AssignmentExpression                                          initialization
                            |  '{' , InitializerList , [ ',' ] , '}'
          InitializerList  ::= [ Designation ] , Initializer ,
                               { ',' , [ Designation ] , Initializer }
              Designation  ::= DesignatorList , '='
           DesignatorList  ::= Designator , { Designator }
               Designator  ::= '[' , IntegerConstant , ']'
                            |  '.' , Identifier
```

A.4 Statements

```
                Statement  ::= CompoundStatement                                             compound stmt
                            |  ExpressionStatement                                           expression stmt
                            |  SelectionStatement                                            selection stmt
                            |  IterationStatement                                            iteration stmt
                            |  JumpStatement                                                 jump stmt
        CompoundStatement  ::= '{' , [ BlockItemList ] , '}'                                 compound stmt
            BlockItemList  ::= BlockItem , { BlockItem }
                BlockItem  ::= Statement
                            |  Declaration
      ExpressionStatement  ::= [ Expression ] , ';'                                          expression stmt
       SelectionStatement  ::= 'if' , '(' , Expression , ')' , Statement ,                   if stmt
                               [ 'else' , Statement ]
       IterationStatement  ::= 'while' , '(' , Expression , ')' , Statement                  while stmt
                            |  'do' , Statement ,                                            do stmt
                               'while' , '(' , Expression , ')' , ';'
                            |  'for' , '(' , [ Expression ] , ';' ,                          for stmt
                                             [ Expression ] , ';' ,
                                             [ Expression ] , ')' , Statement
```

```
JumpStatement  ::= 'continue' , ';'                         continue stmt
                 | 'break' , ';'                            break stmt
                 | 'return' , [ Expression ] , ';'          return stmt
```

A.5 External Definitions

```
TranslationUnit     ::= ExternalDeclaration , { ExternalDeclaration }   external definitions
ExternalDeclaration ::= FunctionDefinition
                      | Declaration
FunctionDefinition  ::= DeclarationSpecifiers , Declarator ,            function definitions
                        CompoundStatement
DeclarationList     ::= Declaration , { Declaration }
```