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' |        keyword
              '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'
```

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 |     integer constant
                         OctalConstant ) , [ IntegerSuffix ]
     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`	`int`
	`long int`	`unsigned int`
	`long long int`	`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`
u or U	`unsigned int`	`unsigned int`
	`unsigned long int`	`unsigned long int`
	`unsigned long long int`	`unsigned long long int`
l or L	`long int`	`long int`
	`long 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 int`
	`unsigned long 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

character constant

```
         CharacterConstant  ::=  ''' , CChar , '''
                     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

string literal

```
  StringLiteral  ::=  '"' , [ SCharSequence ] , '"'
 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 ,                          bitwise OR operator
                            { '|' , ExclusiveORExpression }
```

3.13 Logical AND operator

Syntax

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

3.14 Logical OR operator

Syntax

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

3.15 Conditional operator

Syntax

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

3.16 Assignment operators

Syntax

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

3.16.1 Simple assignment

3.16.2 Compound assignment

3.17 Comma operator

Syntax

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

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 ] ,                      structure specifier
                             '{' , StructDeclarationList , '}'
                         |   '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 }
ExternalDeclaration  ::=  FunctionDefinition
                       |  Declaration
```

external definitions

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 ,
                         CompoundStatement
   DeclarationList  ::=  Declaration , { Declaration }
```

function definitions

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

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] ,	structure specifier
		'{' , StructDeclarationList , '}'	
	|	'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 }
```