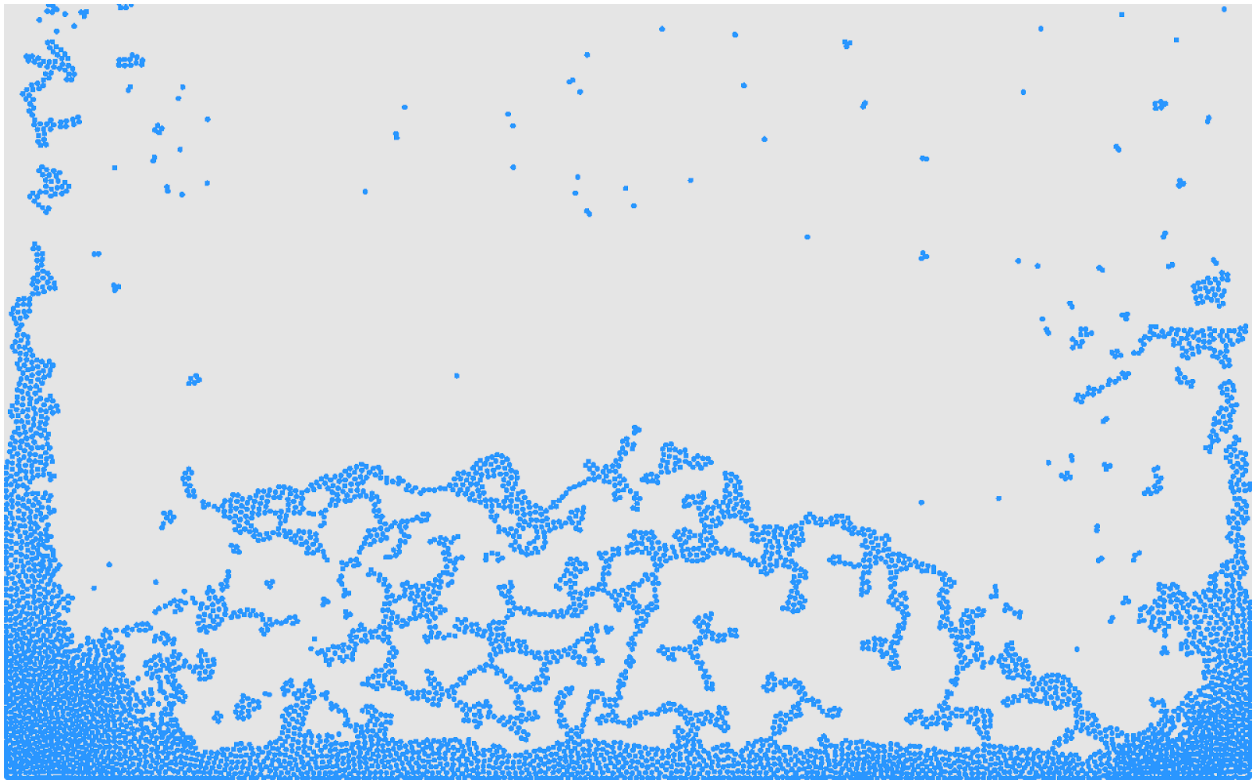

shux

Language Reference Manual

Lucas Schuermann (lvs2124)
John Hui (jzh2106)
Mert Ussakli (mu2228)
Andy Xu (lx2180)

Table of Contents

(1) Introduction

Though physics simulation is a massive field with many areas of active research, much emphasis is given to so-called “Lagrangian” problems, or particle-based discretization schemes, commonly encountered when modeling phenomena such as granular materials, fluid dynamics, cloth, and many others.

shux is a language optimized for expressing, simulating, and rendering particle-based (Lagrangian) physics problems. Currently, though many implementations of solvers for such problems exist, they are frequently overly verbose, poorly organized and poorly optimized, and cluttered with helper code for rendering, spatial gridding, and multiprocessing. By introducing a revised syntax, better-suited semantics and adequate abstraction, shux aims to function as a general-purpose language better catered towards the needs of particle-based physics simulations.

shux is fundamentally based upon **generators** and pure functions (**kernels**), which can be processed asynchronously over the set of all simulated particles. Other features, such as variable state lookback, strong typing, and built-in functional mathematical expression representation and optimization features help to facilitate a maximally concise and minimally error-prone user experience when solving domain-specific problems, largely in physical phenomena simulation, as modeled by particle-discretized partial differential equations.

(2) Lexical Conventions

(2.a) Identifiers

An identifier in shux, also known as a programmer-defined name, can be any ASCII string that begins with an alphabetic character or an underscore, followed by any number of lowercase/uppercase letters, underscores, and numbers.

```
[‘a’-’z’ ‘A’-’Z’ ‘_’][‘a’-’z’ ‘A’-’Z’ ‘_’ ‘0’-’9’]*
```

(2.b) Keywords

(2.b.i) Declarations

shux uses the following modifiers to indicate the type of variable bindings in a global scope (outside of function definitions):
ns
let
kn
gn

Within functions, the following keyword is used to indicate a mutable value (rather than immutable defaults):
var

(2.b.ii) Control Flow

shux attempts to limit imperative programming by introducing conditional and looping constructs as expressions rather than statements, as follows:
if … then … [elif] … else
for
do...while

shux does support C-style while-loops, which can be used to implement C-style for-loops and do-while-loops, but as doing so requires mutable values, it is highly discouraged.

(2.b.iii) Types

shux uses the following keywords to refer to types. shux's typing system is discussed in detail in Section **(2.d)**.

primitives:
bool
int
scalar
string

collections:
array
struct
vector

(2.c) Literals

shux supports all of its primitive types as literals in code, defined as follows.

(2.c.i) Integer

ints are 32-bit signed integers. They consist of at least one digit. Integer literals are defined using the following regular expression:

```
['0'-'9']+
```

(2.c.ii) Scalar

scalars are 64-bit signed floating point numbers. They are structured just like doubles in C.

The decimal point character . defines scalars. The part before the decimal point is the integer part of the scalar, and the part after the decimal is point is the fraction part of the scalar.

To be maximally compatible with scientific calculations, shux scalars support scientific notation, which indicates the decimal exponential of the scalar. It is the character e, followed by a +/- and then an integer.

These can be defined by the following regular expression:

```
((['0'-'9']+ '.' ['0'-'9']* | ['0'-'9']* '.' ['0'-'9']+) ('e' ['+' '-'] ['0'-'9']+)?)
```

Example of a scalar literal: 12.37e-17

(2.c.iii) Boolean

shux's basic boolean type. shux reserves the keywords *true* and *false* to refer to boolean literals.

(2.c.iv) Escape Sequences

The following sequences within a string have their special semantics:

\n	new line
\r	carriage return
\t	tab
\"	double quotes
\\	backslash

(2.c.v) Strings

String literals are any ASCII characters in double quotes.

"I am a string literal"

Strings can be defined by the following regular expression:

\" (\\. | [^"])* \"

(2.d) Data Types

shux is strongly typed. Operators are only allowed in between same types unless specified otherwise.

Implicit type casting is not available in our language syntax – instead, we will provide library functions that perform these conversions.

(2.d.i) Mutable variables

var precedes a type declaration that is mutable.

Sample Usage:
var scalar y = 5.0;
y = 4.2;

(2.d.ii) Primitives

(2.d.ii.1) booleans

Type: bool.
Can be true or false

(2.d.ii.2) integers

Type: int

32-bit signed integer literal.

(2.d.ii.3) scalar

Type: scalar
64-bit signed floating point number.

(2.d.ii.4) string

Type: string
Sequence of ASCII literals. Enclosed in double quotes for string literals. Escape character is backslash '\'

\n for new line, \t for tab sequences.

(2.d.iii) Collections

(2.d.iii.1) Arrays

A sequence of names in contiguous memory. The [] operator is used for array access and declaration. Arrays are immutable by default.

Instantiation: *type[] name*
Access: *name[index]*

```
int[5] x = {0,1,2,3,4};          /* initialize 5 integers in an immutable array */
var int[5] y;                    /* declare mutable array*/
x[3] = 4;                        /* set index 3 to 4 */
```

(2.d.iii.2) Structs

Structs are collections of primitive types in a C-like style.

Definition: *struct struct-name { type1 field1; type2 field2; … }*
Instantiation: *struct struct-name name = {field1 = value1, field2 = value2, … }*
Access: *name.field-name*

Example:

```
struct particle {
        float x;
        float y;
}

struct particle p = {x=1.0, y=2.0};
```

(2.d.iii.3) Vectors

Vectors are collections over which mathematical operations are built-in. Vectors can only contain scalar types and are designed for linear algebra and matrix calculations.

Instantiation: *var vector name = <lit1, lit2, lit3, … >*
Access: *name[index]*

```
vector g = <0.0, -9.81>;
float y_accel = g[1];
```

(2.e) Comments

```
/* Luke was here*/
var particle[10] p_init = {0}; /* Andy is a potato */

/*
 * test comment
 * hi mom
 * test line 3
 */
var particle[10] p_init = {0};
```

C-style syntax for single-line comments is not supported in shux. Subsequent lines of multi-line comments must begin with *.

(2.f) Operations

(2.f.i) Value Binding

A single equals sign '=' indicates assignment, wherein an expression on the right is bound to the identifier on the left.

(2.f.ii) Operators

Precedence	Operator	Description	Associativity
1	() [] <>	Function call Array subscripting Vector subscripting	Left-to-right
2	..	Historical variable access Structure member access	

	.		
3	!	Logical negation	Right-to-left
4	- (unary)	Unary minus	
5	* / %	Multiplication, division and remainder	Left-to-right
6	+ - (comp)	Addition and subtraction	
7	== <= >= > <	Relational operators	
8	&& ||	Logical AND and logical OR	
9	::	Filtering operator	Left-to-right
10	@	Map	
11	= += -= *= /=	Simple assignment Assignment by sum and difference Assignment by product and quotient	Right-to-left
12	,	Separator	Left-to-right

(3) Syntax

(3.a) Program Structure and Global Namespace

A shux program consists of, strictly in the following order, namespace declarations, global constant declarations, and function (kernels or generators) declarations. This is ordering is syntactically enforced for code clarity.

program
    ns-decls let-decls fn-decls

All shux programs **must** implement a program entry point as follows:

```
/* reserved main id */
kn main() -> int {
    /* [...] program logic */
}
```

main() must be a kernel (stateless) function that returns an integer, and has a lookback value of 0.

(3.a.i) Namespacing

In order to organise globally-named function and constant declarations into cohesive units, shux uses namespaces which encapsulate programs within them. A program can begin with any number of namespaces, and though they may be nested, this is highly discouraged.

ns-decls
    ns-decls ns-decl

ns-decl
    `ns` *ns-id* { *program* }

Declarations within namespaces may be later accessed with the dot operator. For example:

Declaration:
```
ns foo {
    let int bar = 4;
}
```

Access:

```
foo.bar
```

Note that after namespaces have been declared, the program continues to define constants and functions in the *global namespace*. This means that these name bindings do not need to be preceded by a namespace identifier and the dot operator.

Including standard libraries using *#include<...>* effectively adds a pre-defined global namespace to the program. The shux standard libraries are extensively discussed in detail in Section **(4)**.

(3.a.ii) Global Constant Declarations

In the global namespace, any number of static values may be declared and bound to identifiers using the `let` keyword, followed by some type identifier, the = (assignment) operator, and the static value that it should be associated with.

let-decls
    let-decls let-decl

    _
let-decl
    let type id = expr ;

The static values bound to those identifiers will be evaluated at compile time and associated with that identifier for the entire following namespace, and so must be unique.

(3.a.iii) Named Function Declarations

The global namespace ends with any number of named function declarations and definitions, and may either be stateless kernels (indicated by the kn keyword) or locally stateful generators (indicated by the gn keyword).

fn-decls
    fn-decls gn-decl
    fn-decls kn-decl

    _
gn-decl
    gn *id* (*_formals*) *_ret-type* { *local* }
kn-decl
    kn *id* (*_formals*) *_ret-type* { *local* }
formals
    formals , formal
    formal
ret-type
    -> *type*

Note that the optional return type follows the parameter list rather than preceding it, and is indicate by the -> token. If it is not present, the return type is assumed to be void. Returned collection types (such as arrays and structs) are returned by reference, but to immutable data structures.

(3.a.iv) Lookback

shux has a special lookback feature defined over variables. The ".." operator is used over variables with historical access to access their values in previous iterations.

```
gn foo(int bar) -> int {
	int i = i..1 + 3 : bar;
	int j = i..2; //set j equal to the value of i two iterations earlier
}
```

shux implements the lookback feature through a circular buffer in memory representing the states of the variables at each past (and current) iteration. For memory efficiency, the size of the buffer is determined statically through the highest backwards access performed on the variable.

While the lookback feature is defined on generator variables, it can also be used on pure functions (kernels). In this case, the backwards access limits of the variables passed into the kernel are limited by the backwards access limit defined by the generator that calls the kernels. This is determined in compile-time.

If the lookback value is not available (for example, if historical access is attempted on the first iteration), the value of the variable on the current iteration will be returned. shux allows you to specify what value to return if lookback value isn't variable through the following syntax:

```
int i = i..2 + 3 : 0; /* add 3 to the value of i 2 iterations ago and return
                       * or just return 0 if not available. */
```

(3.b) Local namespace

Inside named function blocks is the *local namespace*. This is composed of an optional series of statements, separated by semicolons. Further namespaces, global constants, and named functions may no longer be declared.

local
	_block ; *_ret-expr*
block
	block ; *statement*
statement

decl
expr
ret-expr
expr

Statements consist of either declarations or expressions.

Each local namespace ends with an optional return expression that is not semicolon-terminated – this is the return value of function that local namespace defines. If it is absent, then the function returns a void.

(3.b.i) Declarations

By default, shux variables are declared as immutable. In order to make them mutable, one may specify the var keyword.

decl
_var formal = expr
_var formal
formal
type id

Variable declarations may be mixed with their assignment and thus instantiation, or that may be deferred until later. Subsequent expressions in the local namespace may then access declared variables. However, deferring the assignment for immutable variables may be useful for capturing historical values, for example:

```
gn foo(int bar) -> int {
      int ret;
      ...
      int baz = ret..1 : baz;      /* access the value of ret in its
previous iteration */
      ...                     /* else use default value of baz */
      ret = baz * 2                /* save the return value to ret for
future reference */
}
```

Note that while anonymous functions may be declared, they cannot be bound to declared variables, in order to avoid complications with function types or functional type inference.

(3.c) Expressions

In shux, almost everything is an expression. By having representing algorithmic constructs such as loops and conditionals as expressions, control flow becomes more predictable, and while the syntax looks imperative, the expressed algorithms are in fact functional.

expr
    asn-expr

(3.c.i) Assignment

The assignment expression follows the C style, and takes on the value of its right operand:
asn-expr
    unary-expr asn-op asn-expr
    conditional-expr

asn-op one of
    `= += -= *= /= %= <<= >>= &= ^= |=`

Doing so allows the chaining of assignments. For example (assuming all the variables have already been declared):
`x = y = z = 69;`
is parsed as
`x = (y = (z = 69));`
and takes on the value of 69.

(3.c.ii) Conditional Expressions

Conditional expressions are to switch the value of an expression between sub-expressions depending on some predicate.

conditional-expr
    fn-expr `:` *conditional-expr* *// for historical access*
    bool-expr `?` *fn-expr* `:` *conditional-expr*
    `if` *bool-expr* `then` *fn-expr else-expr*
else-expr
    `elif` *bool-expr* `then` *fn-expr else-expr*
    fn-expr

The first type of conditional expression, of the form `a : b`, is for lookback. This expression will take the value of `a` if is available, but if not, will default to the value of expression `b`. Note that `b` can be another such expression, allowing the chaining of conditional expressions. For example:

```
x = x..1 : 0;
```

In this, `x` is assigned its value in the previous iteration, but on the first iteration, where its history is unavailable, it will be assigned a default value of 0.

The `if..then..else` construct is semantically equivalent to the ternary ? operator. Each conditional expresson may switch between an arbitrary number of conditions, e.g.:

```
x = x..2 : x..1 : 0
y = if cond1 then val1 elif cond2 then val2 else val3
z = cond1 ? val1 : cond2 ? val2 : val3
```

The expressions assigned to `y` and `z` are semantically identical.

(3.c.iii) Functional Expressions

The following class of expressions are used to denote operations that take place on list-yielding and functional data types.

(3.c.iii.1) Maps and Filters

Maps (@) and filters (::) may be used on list/array types in order to yield new lists from those lists.

fn-expr

fn-expr `@` *kn-expr*

fn-expr `::` *kn-expr*

iter-expr

These operators are used to apply a following kernel function expression onto the left list-yielding operand. Examples for their usage will be shown in the next section, after the syntax for anonymous functions is introduced.

(3.c.iii.2) Function Expressions

Function expressions may take on one of two forms: a reference to a named kernel function previously defined in the global namespace, or an anonymous kernel function (lambda).

kn-expr

id (*exprs*)

_formals -> { *block* }

Anonymous functions, or lambdas can be declared in the local namespace. The syntax for defining lambdas is similar to that of declaring named functions, except without the identifier. In the following example, an incrementing map function is being applied to an array of integers in order to yield another array, and then filtered to only yield anything less than 5:

```
int[] inc_less_than_5 = original @ i -> { i + 1 } :: i -> { i < 5 };
```

Semantically, lambdas are equivalent to kernels, except they also inherit the namespace of the function block they are declared within.

(3.c.iv) Iterative Expressions

Iterative expressions may be used on named generator functions in order to control the number of iterations those generators execute.

iter-expr

    for *unit-expr gn*
    do *unit-expr gn*
    unit-expr

gn

    id (*exprs*)
    ()

The `for` construct will produce an array containing all the values yielded by the generator called. The `do` construct will produce the value yielded by the generator after *unit-expr* number of iterations, and toss away the intermediate values.

This generator may either be a named generator function previously declared in the global namespace, or the unit generator, (), which does nothing – this is useful for constructing iterative loops.

(3.c.v) Unit Expressions

Unit expressions largely follow the same syntactical rules as C. The details are elaborated in the our grammar. The following operators and constructs differ from C:

- shux uses the . and .. postfix operators followed by a numeric literal to indicate lookback
- shux does not use the * and & operators for dereferencing and referencing – it does not support pointer arithmetic

(4) Standard Library

(4.a) Graphics

Visual display of simulated phenomena, both by means of still images and real-time rendering, is an integral part of nearly all computational physics problems, especially in the case of particle-based dynamics. In order to reduce the time-to-live for producing visual feedback from code, shux includes <graphics>, a basic graphics library optimized for creating simple real-time displays of particle states, based upon well-specified parameters, such as position, size, and color.

Basic graphics operations:
<graphics>
init_window
set_vertex_array
set_vertex_position
set_point_size
set_point_color
set_background_color
flush_screen
display_loop

Sample Usage:
```
#include<graphics>

fn main() {
	graphics.init_window(640, 480, “hello”);
	graphics.set_background_color(255,0,0); /* red */

	/* display loop */
}
```

(4.b) Streams

Like C++ iostream standard library, shux uses stream libraries to handle file and standard streams operations.

(4.b.1) Standard Streams

<stdio>
shux uses <stdio> library to provide stream operations on standard input, standard output and standard error. Functions provided in <stdio> can be invoked anywhere in the body part of a shux program. Nevertheless, the use of standard streams is strongly discouraged because it uses slow system calls that might block the execution of the program. In principle, <stdio> library should only be used for either runtime user input, or to debug the program at development stage. For a shux program to run efficiently, <stdio> functions should not be called inside any generator function or kernel function.

Sample Usage:
```
#include <stdio>
int x = 99;
stdio.print_error("%d bottles of beer.\n", x);
```

(4.b.2) File Operations

To provide local file read and write operations, shux provides a <fileio> library, which can be used inside the program's main() function, but not inside any kernel function or generator functions due to concurrency concerns.

Sample Usage:
```
#include <fileio>

file = fileio.open("output.log");
string result = "{x: 1, y: 2, vx: 10, vy: 20}";
file.write(result);
```

(4.c) Algorithms

shux includes an algorithm library that provides useful algorithms for physics simulations. The algorithm library is implemented in shux language itself and is loadable in the header. Most of them

Basic data structures and containers:
<stack>
<queue>
<linked_list>
<heap>
<grids>

Sample Usage (stack):
```
#include<stack>

stack my_stack = stack.create();
my_stack.push(3);
my_stack.push(4);
var result = my_stack.pop(); /* gives 4 */
result = my_stack.pop(); /* gives 3 */
```

Basic algorithms:
<qsort>
<binary_search>
<reverse>
<grid_helper>

Sample Usage: Grid

```
#include<grid>

struct particle p1 = { … }; /* a particle struct */
struct particle p2 = { … };
var grid main_grid = grid.create(grid_width, grid_height, cell_size);
grid.insert(0, 5, p1); /* a struct and its position is inserted */
grid.insert(5, 5, p2);
```

hmm this syntax is not compatible with grids unless we sugar coat them.
let's sugar coat them hahaha

```
/* grids are collections of cells, which contain x y coordinates in grid
 * and a reference to the struct that is inserted in the grid.
main_grid = main_grid @ cell -> {
        particle = cell.particle;
        cell <- {
                .particle = /* perform operation on particle return new particle */
        }
}
```

Sample Usage Algorithms:

```
#include<algorithms>

int[5] arr = {2, 5, 7, 1, 0};
result = algorithms.qsort(arr);
/* arr == {2,5,7,1,0}; result = {0, 1, 2, 5, 7}; */

var int[5] arr2 = {2, 5, 7, 1, 0};
algorithms.qsort(arr2);
/* arr2 == {0, 1, 2, 5, 7} */
```

(4.d) Linear Algebra

Matrix, Vector Operations	Addition, subtraction Scalar multiplication and division Transposition and conjugation Matrix-matrix and matrix-vector multiplication Dot product and cross product
Block operations	Initializations Block extraction and manipulations Column, row operations

	Cornor-related operations
Reductions	Normalization Linear solving and Reduced echelon form
Decompositions	Inverse Determinant Least squares Rank revealing Eigenvalue (power and jacobi)
Numerical Method Algorithms	<newtons> <runge_kutta> <relaxation> <euler> <verlet> <leapfrog>

(4.d) Pervasives

Since shux is a strongly typed language without the support of implicit type casting, all type casting operations have to be done explicitly. Similarly to how it's done in Ocaml, shux provides <pervasives> in its standard library that allows programmers to convert between compatible types.

Following type casting operations are supported by <pervasives>:

scalar_of_int
int_of_scalar
string_of_scalar
scalar_of_string
string_of_scalar
string_of_format

Sample Usage:

```
#include<pervasives>
scalar e = 2.718281828
int e_int = pervasives.int_of_scalar(e) /* e_int == 2 */
string explanation = pervasives.string_of_format("%.8f is rounded down to %d\n", e, e_int);
```

(5) Appendix

(5.a) Toolchain

Our compiler will be named shucc, producing an executable from *exactly* one shux source file. It will accept the following arguments

-c file to compile
-o <file> output filename
-s <file> output intermediate LLVM
-v print debugging information

(5.b) References

Prof. Edwards' slides
Dennis M. Ritchie's "C Reference Manual" https://www.bell-labs.com/usr/dmr/www/cman.pdf
http://www1.cs.columbia.edu/~sedwards/classes/2015/4115-fall/lrms/note-hashtag.pdf
http://www1.cs.columbia.edu/~sedwards/classes/2012/w4115-fall/lrms/Funk.pdf

(6) Grammar (AST)

NB: _ before a symbol indicates that it is optional
_ on its own means null string

program
    ns-decls let-decls fn-decls
ns-decls
    ns-decls ns-decl
    _
let-decls
    let-decls let-decl
    _
fn-decls
    fn-decls gn-decl
    fn-decls kn-decl
    _
ns-decl
    ns id = { program }
let-decl
    let type id = rvalue ;
    let struct id = { struct-def }
gn-decl
    gn id (_formals) _ret-type { block _ret-expr }
kn-decl
    kn id (_formals) _ret-type { block _ret-expr }
struct-def
    struct-def; formal
    formal
block
    block ; statement
    statement
    conditional-statement
    iteration-statement
conditional-statement
    if (expr) then { block } else { block }
    if (expr) then { block }

iteration-statement
    for (_expr ; _expr ; _expr) { block }

```
statement
        decl
        expr

ret-expr
        expr
expr
        asn-expr

asn-expr
        unary-expr asn-op asn-expr
        unary-expr asn-op conditional-expr

conditional-expr
        conditional-expr : fn-expr
        bool-expr ? fn-expr : conditional-expr
        if bool-expr then fn-expr else conditional-expr
        fn-expr

fn-expr
        fn-expr @ kn
        fn-expr :: kn
        iter-expr
kn
        id ( exprs)
_formals -> { block }
iter-expr
        for unit-expr gn
        do unit-expr gn
        unit-expr
gn
        id ( exprs )
        ()

unit-expr
        bool-expr
bool-expr
        bool-or-expr
bool-or-expr
        bool-or-expr || bool-and-expr
        bool-and-expr
bool-and-expr
        bool-and-expr && bit-expr
```

```
        bit-expr

bit-expr
        bit-or-expr
bit-or-expr
        bit-or-expr | bit-xor-expr
        bit-xor-expr
bit-xor-expr
        bit-xor-expr ^ bit-and-expr
        bit-and-expr
bit-and-expr
        bit-and-expr & cmp-expr
        cmp-expr

cmp-expr
        eq-expr
eq-expr
        eq-expr == relat-expr
        eq-expr != relat-expr
        relat-expr
relat-expr
        relat-expr < shift-expr
        relat-expr > shift-expr
        relat-expr <= bit-shift-expr
        relat-expr >= bit-shift-expr

bit-shift-expr
        bit-shift-expr << arithmetic-expr
        bit-shift-expr >> arithmetic-expr
        arithmetic-expr

arithmetic-expr
        add-expr
add-expr
        add-expr + mult-expr
        add-expr - mult-expr
        mult-expr
mult-expr
        mult-expr * unary-expr
        mult-expr / unary-expr
        unary-expr

unary-expr
```

```
        _unary-op postfix-expr
unary-op
        +
        -
        ~
        !

postfix-expr
        postfix-expr [ expr ]
        postfix-expr ( exprs )
        postfix-expr . id
        postfix-expr . num
        postfix-expr .. num
        postfix-expr < exprs >
        primary-expr

exprs
        exprs expr
        expr
primary-expr
        id
        lit
        ( rvalue )
decl
        var formal              // mutable
        formal                  // immutable

formals
        formals , formal
formal
        type id

ret-type
        -> type
type
        primitive-t _array-t
        id _array-t             // for structs; our scanner + parser won't really know this

primitive-t
        int
        float
        string
        bool
```

```
        vector-t
vector-t
        vec< num >
array-t
        [] array-t

lit
        struct-lit
        array-lit
        vector-lit
        string-lit
        num
        id
struct-lit
        { struct-fields }
struct-lit-fields
        struct-fields , struct-field
struct-lit-field
        .id = expr

array-lit
        [ array-elements ]
array-elements
        array-elements , expr

vector-lit
        < vector-elements >
vector-elements
        vector-elements , expr
```