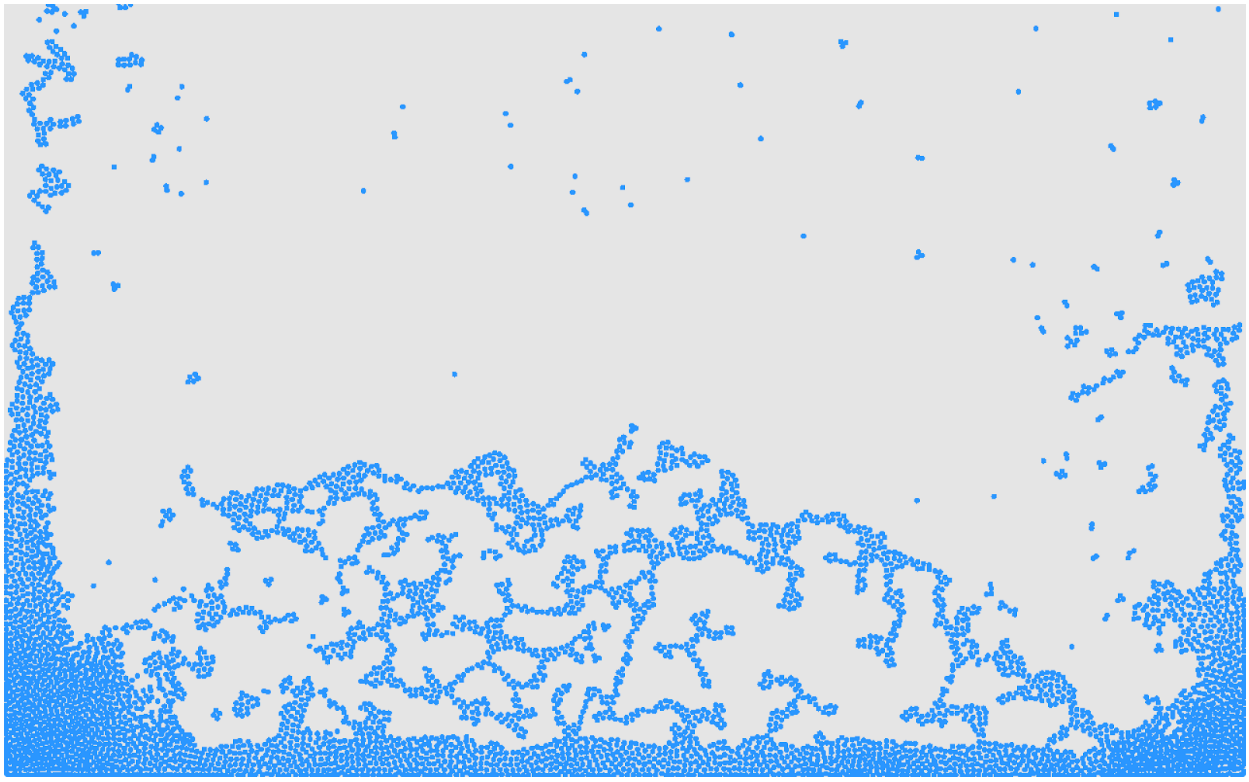


shux

Project Report

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

Table of Contents

Introduction

shux is a language optimized for particle-based fluid simulation. The syntax facilitates writing concurrently-safe, easily extendible, and simply expressed mathematical rules by which particles are updated each frame. Standard library functions allow for the display of such simulations in real time to the user.

Programming in shux revolves around the concepts of kernels and generators. A kernel is a pure function, whereas a generator is a stateful function intended to be expressed over the set of particles and called many times to move a simulation through forward time. Generators and kernels are the building blocks of our simulations because they help programmers to easily express a separation between stateful computations and pure computations, maintain centralized local statefulness such that parallelism could be extended in the future, and easily represent complex operations using operators such as for, do, map, and filter on generating expressions with our lambda system.

With this rich feature set, while originally designed as a domain-specific language for particle-based fluid simulation, shux is a reasonably robust general-purpose modern language that provides a toolset familiar to programmers experienced with languages like the C family, while incorporating a number of functional syntax features such as the aforementioned maps, filters, and lambdas, in addition to performant lookback for easy implementation of integrators, and much more.

Related Work

The development of shux was heavily influenced by the team's (Luke's) past work in the area of particle-based fluid simulation. Particle-based methods have become increasingly popular in recent years, due in part to the rise of computer graphics and numerical methods for physics. While mathematically elegant, implementations in the public and private domain of particle-based simulation algorithms such as Smoothed Particle Hydrodynamics, Discrete Element Method, and more have generally been written in C++ with extensive use of external linear algebra and mathematics libraries or, nowadays, in Nvidia CUDA, allowing for efficient parallel computation on the GPU.

While GPU computing will always serve a purpose of allowing for significant performance gains in this area, many of these solvers, especially those that are CPU bound and used during the research process, are frequently overly verbose, poorly organized and poorly optimized, and cluttered with helper code for rendering, spatial gridding, and multiprocessing. Little progress has been made on this problem specifically in the field of Lagrangian physics simulation, though other domain-specific languages have been introduced to great fanfare (and publication in the

esteemed ACM SIGGRAPH conference journal), such as simit-lang, a language for easy expression of algorithms on mesh-like data structures.

Driven by the success of these languages, we looked to understand what made them popular in their specific domains, noting that specific syntactic sugar, robust standard libraries, and language designs which minimized programmer error. Keeping these success in mind, we worked closely together to design a language based on past experience in the area of Lagrangian physics simulation.

Tutorial

We will provide a brief introduction to the syntax of shux, common motifs, and define how to use some notable constructs. For more information on specifics, please read the LRM in a following section.

Basic shux program

```
kn main () int {
     0
}
```

A less basic program

```
let global int  = 9; /* global type. these are immutables.

kn foo(int[] arr, int x) int {
     int y = x - 1; /* declare immutable type */
     var int z = y + 1; /* declare mutable type */
     z - 1; /* change mutable type */
     arr[0] + arr[1] + x /* return expression simply has no semicolon */
}
kn main() int [
     int[5] array = [1,2,3,4,5];
     int result = foo(array, 7);
     result
}
```

Structs

```
struct stuff { scalar x; scalar y; bool z; }

kn main() int {
     scalar x;
     scalar y;
     x = y = 1.0; /* if not initialized, can assign to immutable types */

     struct stuff foo = stuff { .x=x; .y=y; .z=false}
     if foo.x < 1.5 then 1 else 0 /* if's are simply expressions, just like
                                     everything */
}
```

Filters

```
kn lessThanThree(int x) bool {
      x < 3
}

kn main() int  {
      int[5] values = [2,3,4,5,6];
      int[] filtered = values :: lessThanThree; /* filtered = [2] */
}
```

Maps

```
kn aboveGVelocity(vector a) bool {
     a > 9.81 || a<-9.81
}

kn main() int {
    vector<7> accelerations = (1.0, 17.2, -2.7, 3.6, 2.99, 0.128, 12.1);
    bool[] aboveg = accelerations @ aboveGVelocity;
    if bool[3] -1 else 0
}
```

Calling Generators: For and Do's

```
kn nothing() { /* this is what a void function looks like in shux */

}
gn foo(int x)  {
    x+1
}

kn main() int {
    int[5] x = for 5 do foo(1); /* Returns array [2,2,2,2,2] */
    int y = do 10 do foo(1); /* returns 2, the final element /*
    x[0]                        after the 10th call */
}
```

But what's the point of calling the same function several times?

Lookback Values

```
int x = x..2 : 0 + 1; /* update the value of x by
                         taking the value two iterations ago
                         and adding one to it. if can't go
                         back two iterations, take 0 */

Exampl:

gn fib() int {
     int x = x..1 : 0 + x..2 : 1
}


kn main() int {
     int[5] fibs = for 5 fib(); /* [1,1,2,3,5] becomes fibs */
}
```

Lambdas

```
kn main() int {
     int[6] numbers = [1,3,5,7,9,11];
     int[6] double = numbers @ (int x) -> { x*2 };
     int[2] /*returns 6 */
}
```

Complex Example

Euler approximation on an array of particles in shux. By doing OpenGL pushes where specified within the comments, we can simulate some basic particle simulation

```
  gn euler(struct particle[] init) struct particle[] {
        struct particle[] pbuf = (pbuf..1 : init) @ (struct particle p) -> {
        particle {
           .v = p.v + g * dt;
           .x = p.x + p.v * dt
```

```
      }
   };
  pbuf
}

kn main() int {
  var struct particle[1] p_init = [ particle { .v=1.0; .x=0.0}];
  struct particle[1000] euler = for 1000 euler(p_init);
  euler @ (struct particle state) -> {
          state /* push to OpenGL--bindings not finalized */
     };
   0
}
```

Language Reference Manual

Lexical Conventions

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’]*
```

Keywords

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

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.

Types

shux uses the following keywords to refer to types. shux's typing system is discussed in detail in the following sections.

primitives:
bool
int
scalar / float
string

collections:
array
struct
vector

Literals

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

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']+
```

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

Boolean

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

Escape Sequences

The following sequences within a string have their special semantics:

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

Strings

String literals are any ASCII characters in double quotes.

```
"I am a string literal"
```

Strings can be defined by the following regular expression:

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

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.

Mutable variables

var precedes a type declaration that is mutable.

Sample Usage:

```
var scalar y = 5.0;
y = 4.2;
```

Primitives

Booleans

Type: bool.
Can be true or false

Integers

Type: int
32-bit signed integer literal.

Scalar

Type: scalar
64-bit signed floating point number.

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.

Collections

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 immutable array */
var int[5] y;           /* declare mutable array*/
x[3] = 4;               /* set index 3 to 4 */
```

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};
```

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<size> name = (lit1, lit2, lit3, …)*
Access: *name[index]*

```
vector<2> g = (0.0, -9.81);
scalar y_accel = g[1];
```

Comments

```
/* Luke was here*/
var struct particle[5] p_init;  /* Andy is a potato */

/*
 * test comment
 * hi mom
 * test line 3
 */
var struct particle[5] p_init;
```

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

Operations

Value Binding

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

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

Syntax

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.

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 *extern fname(args)* effectively adds a pre-defined global namespace to the program. The shux standard libraries are further discussed in detail a following section.

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.

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.

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. */
```

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.

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.

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

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.

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.

Functional Expressions

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

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.

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_lt_5 = original @ (int i) -> { i + 1 } :: (int i) -> { i < 5 };
```

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

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.

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

Standard Library

Graphics calls are handled as follows:

```
graphics_init()
graphics_loop(_ptr render_func, _ptr update_func)
graphics_do_render(scalar[] pts_buffer)
graphics_do_update()
```

Graphics init is used to create a window and set up the rendering system. After calling graphics_init in main, the user is able to call graphics_loop, which enters into a blocking loop, returning only upon the termination of the window. render_func and update_func must be shux kernel functions, themselves calling graphics_do_render(`pts_buffer`) and graphics_do_update() respectively. render_func is called upon the need of a new frame for the window (an OpenGL redraw event), and the corresponding points buffer is read and rendered by calling graphics_do_render(pts_buffer), for some 1d array of particle coordinates (flattened). update_func is called when there is “idle” time for the main loop, i.e. as fast as possible to process when not rendering. This should be used for simulation stepping/logic and graphics_do_update() should be called to post a redisplay (keep framebuffer constant).

LRM Appendix

Toolchain

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

`-a` prints the AST

Where the output filename is the .ll file corresponding to the input.

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

Original Prototype Grammar (AST)

*Please see end code listings for final implementation. This is a reduced early version meant for illustration purposes, included with the LRM.

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
        ( expr )
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
```

Project Plan

Our team met once per week for a number of hours on Saturday, with a time on Friday reserved for any necessary catchup/planning/extra work time. Meetings are generally used to sync up on progress, debate and solve problems, and assign specific tasks for the next milestone(s). All team members remain active on slack every day for tasks such as code review, quick question answering, and general discussion.

Our team worked in a horizontally collaborative way on the proposal and LRM, editing shared documents as necessary to produce as large of a volume of ideas as possible, and then made multiple revision passes per person in order to remove duplication, improve wording, and overall increase quality while decreasing verbosity. The same technique was used to push out a high quality final presentation and report in a relatively short amount of time, allowing for maximal time to be spent on language development and testing.

Development Goals/Timeline

After the proposal and LRM, our team set out with the ideal development timeline as follows. Note that while this was demonstrably not exactly adhered to, having a well-organized set of ideas by means of which we could easily point to the next steps we needed to focus on at any given time helped immensely with product management, organization, and maintenance of momentum.

1. Developing the scanner
2. Developing the AST and scanner
3. Polishing core syntax and resolving issues in grammar
4. Testing AST, scanner, parser, and template code generation in hello world
5. Bundling above into regularized development environment with tests
6. Implementing logging module and adding necessary hooks for code generation
7. Begin implementing type checker and translator (AST and code generation for LLVM)
8. Implementing namespacing and global constants with tests
9. Implementing basic generators, implement basic output streams
10. Testing more complex coroutines
11. Implementing basic graphics library for testing
12. Implementing lookback
13. Implementing maps and filters with tests
14. Building full programs and tests for above
18. Implementing full graphics library and begin work on demonstrations
19. Fixing tests, polishing demos, and finding bugs via blind user testing
20. Writing final report, open sourcing materials, and documenting
21. Engaging in post-finals debauchery

For each milestone, a set of tasks were delineated equally amongst team members based upon interest and relevance to different roles/pipeline stage specialization (more rigorously defined later in the report). Any rolling bugs or improvements that were applicable to multiple or any milestone were filed as issues on Github's tracker, and were resolved as time allowed or as project progress necessitated.

Roles and Responsibilities

Formally, Luke was the project manager/tester, Andy was the system architect, John was the language guru, and Mert was the frontend guy/language guru. Because of our integration process and the relative complexity of fixing errors, connecting together stages of the pipeline, testing, and making feature decisions, we all touched and/or read most of the code throughout the development cycle. With that said, our specialization allowed experts to write the majority of the code in their respective parts, meaning that we could optimize for familiarity and ease of implementation within individual stages.

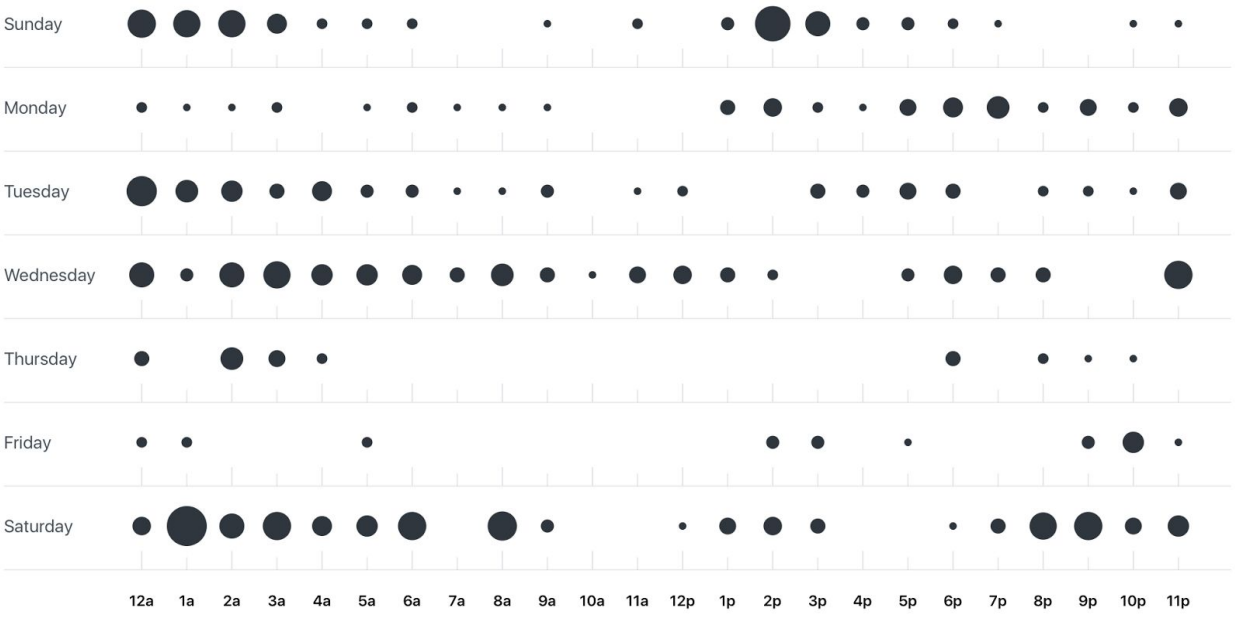
Development Environment

We utilized a provisioned virtual machine on Google cloud for all of our development. By creating an automated environment setup script and all working on the same machine, we were able to reduce incongruencies in build environment. The machine ran command-line Ubuntu 16.04, with all relevant packages installed. We used git for version control (as shown below), tracked many many many issues with github issues tracker, and reviewed pull requests using their tools. We would like to thank emacs for being better than vim (Luke for writing these sections and being able to take poetic license), grubhub for many dinners, and John's suitemate for putting up with many many late night coding sessions.

Version Control Statistics

Graphs were produced by GitHub based upon commit `2263848`.

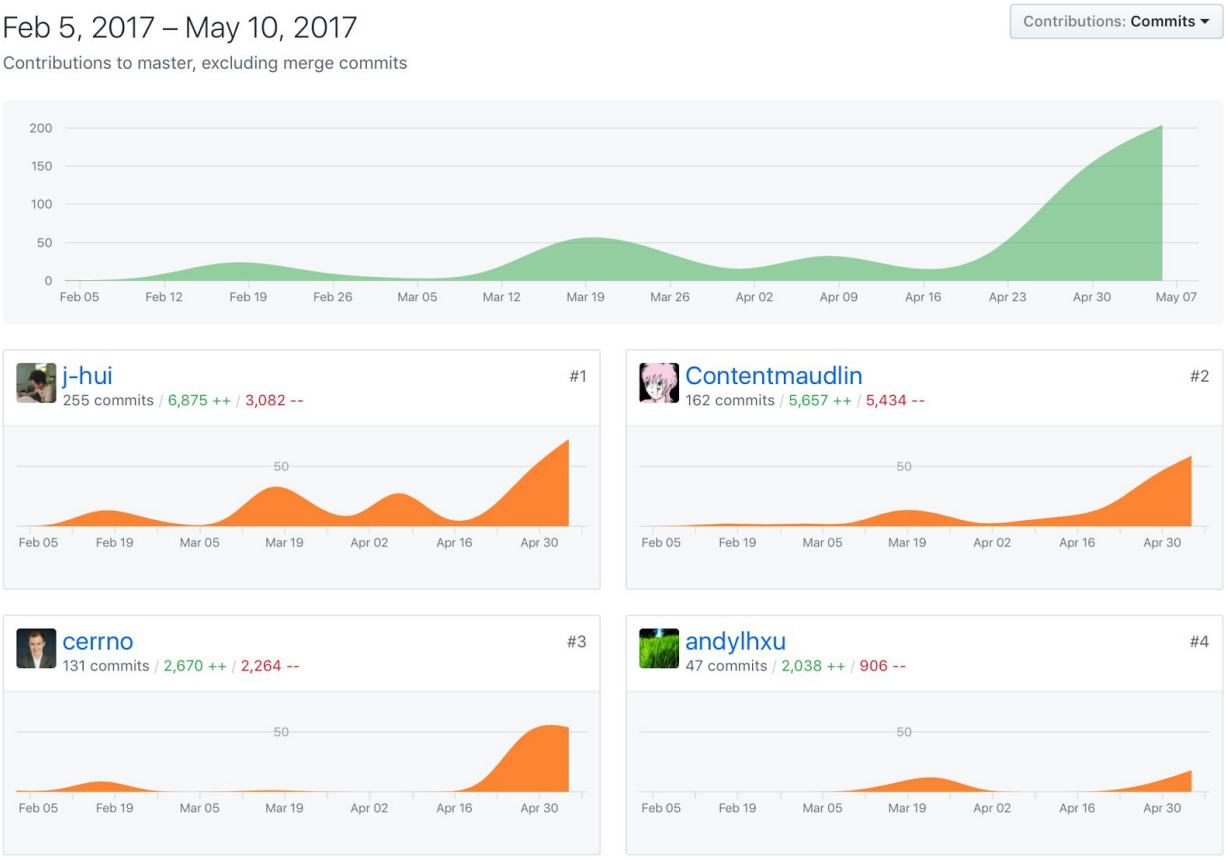
Punch Card



Code Frequency

Contributors

Please note that contributions are disproportionate given the frequency of commits, the nature of the complexity/verbosity of the work, and so on. This is meant to give an impression of our work over time.



Commit Log

Note that all commits are provided, including merge commits, in order to give a sense of development continuity and timeline. Commits are sorted by commit date, not authoring date, so the dates may appear out of order.

- `2017-05-10 Andy Lianghua Xu aeab17c Cll for (#183)`
- `2017-05-10 Andy Lianghua Xu 03d5aed Merge pull request #181 from j-hui/sast-cast`
- `2017-05-10 j-hui f04a480 resolved`
- `2017-05-10 j-hui 49d5c60 fixed function calls`
- `2017-05-10 Andy Lianghua Xu 557ea98 Cll (#180)`
- `2017-05-10 j-hui 39db2af bug fix, binary primitive operators can experience change of type after alol`

- 2017-05-10 j-hui 4e2a32d prefixing
- 2017-05-10 j-hui 4937141 now fixed finally
- 2017-05-10 j-hui 77e591b mrproper
- 2017-05-10 Andy Lianghua Xu 26a5f85 works
- 2017-05-10 Andy Lianghua Xu 97ca821 Merge branch 'cll' of github.com:shux-lang/shux into cll
- 2017-05-10 j-hui a4e2c2c latest
- 2017-05-10 j-hui 18d44fe implemented ccall and cexcall
- 2017-05-10 j-hui e71396d stubbed out other cases for CAST
- 2017-05-10 j-hui 96ff4ea bug fixed cast_llast
- 2017-05-10 Lucas Schuermann aeefb9e merging in graphics code
- 2017-05-10 j-hui a03cb4f Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 John Hui 2263848 something comiples no warnings cuz all disabled (#179)
- 2017-05-10 j-hui 609b2cd something comiples no warnings cuz all disabled
- 2017-05-10 Mert Ussakli 7f2e514 Merge pull request #178 from j-hui/sast-cast
- 2017-05-10 Andy Lianghua Xu 6d9dae5 Merge branch 'master' into cll
- 2017-05-10 j-hui d3784da removed print statements
- 2017-05-10 j-hui f0ddb85 stpid stupid bug
- 2017-05-10 j-hui a60a135 Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 Lucas Schuermann a4ea809 Merge pull request #174 from shux-lang/test_suite
- 2017-05-10 Andy Lianghua Xu 10d6b56 Merge branch 'master' into cll
- 2017-05-10 Andy Lianghua Xu e0790ad prettyprint
- 2017-05-10 Contentmaudlin ed4c2c7 fib
- 2017-05-10 Andy Lianghua Xu 2a9a06e Merge branch 'master' into cll
- 2017-05-10 Andy Lianghua Xu 5ff9259 solve the issue
- 2017-05-10 Andy Lianghua Xu 24c69cd good
- 2017-05-10 Mert Ussakli c546071 Merge pull request #176 from j-hui/sast-cast
- 2017-05-10 j-hui c4dbf62 Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 Contentmaudlin 2f2a322 pushin
- 2017-05-10 j-hui 7ba4c37 print removed
- 2017-05-10 Andy Lianghua Xu 4c1037b Merge branch 'master' into cll
- 2017-05-10 Andy Lianghua Xu 1de7a72 fixing multiple errors
- 2017-05-10 Contentmaudlin 5289a21 bla back
- 2017-05-10 Contentmaudlin 49c7a29 Merge branch 'master' of github.com:shux-lang/shux
- 2017-05-10 Contentmaudlin 2ae5227 bla
- 2017-05-10 Contentmaudlin 78d9027 building euler
- 2017-05-10 Andy Lianghua Xu 06941ad Merge branch 'master' into cll
- 2017-05-10 Andy Lianghua Xu aef6f17 Merge pull request #175 from j-hui/sast-cast

- 2017-05-10 j-hui 16c6b6e Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 j-hui 2531c24 fixed somes
- 2017-05-10 Contentmaudlin 0c75e66 ugh why are there so many ~~B~~ :
- 2017-05-10 Contentmaudlin dfa022c booleans :))
- 2017-05-10 Andy Lianghua Xu a5a94fd adding branch supports
- 2017-05-10 Contentmaudlin 5ec79fe struct access typ bugfix
- 2017-05-10 j-hui 5e8f8cc Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 Mert Ussakli 8f1bf91 Merge pull request #173 from j-hui/master
- 2017-05-10 j-hui a8875bf Merge branch 'master', remote-tracking branch 'origin' into sast-cast
- 2017-05-10 Mert Ussakli ff9ef51 Merge pull request #172 from j-hui/sast-cast
- 2017-05-10 j-hui a76c6af ay
- 2017-05-10 j-hui 9cd3716 Merge branch 'master' of github.com:shux-lang/shux
- 2017-05-10 j-hui 5ff2a7c eager my nuts
- 2017-05-10 Contentmaudlin 0b15565 inherits
- 2017-05-10 j-hui 002b6be updated for gti
- 2017-05-10 Mert Ussakli 8f80f09 Merge pull request #171 from j-hui/sast-cast
- 2017-05-10 j-hui 1801286 update
- 2017-05-10 j-hui 174c050 Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 Mert Ussakli bbb6e0e Merge pull request #170 from j-hui/sast-cast
- 2017-05-10 j-hui 04502ba okay
- 2017-05-10 Andy Lianghua Xu e10b019 added all operators and their implementations
- 2017-05-10 j-hui c500cc8 tail recursion whoop whoop
- 2017-05-10 j-hui 40b8aee not lets either
- 2017-05-10 j-hui 4882861 just don't prefix things for now
- 2017-05-10 Andy Lianghua Xu cf2ac4e Merge branch 'master' into cll
- 2017-05-10 Mert Ussakli a7673af Merge pull request #156 from j-hui/sast-cast
- 2017-05-10 Contentmaudlin a2ffc6a greater than man, greater than earth
- 2017-05-10 j-hui e094954 SEX SEX SEX SEX
- 2017-05-10 Lucas Schuermann e252b5d Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-10 Lucas Schuermann 3e27401 Merge pull request #169 from shux-lang/graphics
- 2017-05-10 j-hui b4da48b Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 j-hui 1d54260 added some comments
- 2017-05-10 Contentmaudlin 17e2934 .
- 2017-05-10 Contentmaudlin 73af30c will you follow where the functional pointers go in your dreams or will you perish

- 2017-05-10 j-hui 6d2f64c this is absolutely wat
- 2017-05-10 Lucas Schuermann 2bf3161 working on extern tests for graphics library
- 2017-05-10 Lucas Schuermann 0842506 Merge pull request #168 from shux-lang/graphics
- 2017-05-10 Contentmaudlin 3cc715a hey doug it's sextern calling
- 2017-05-10 Lucas Schuermann b2ecddf add extern tests
- 2017-05-10 Lucas Schuermann bf27f4a Merge pull request #167 from shux-lang/test_suite
- 2017-05-10 Lucas Schuermann d1ac252 fix vector test
- 2017-05-10 Contentmaudlin 9135d74 temporarily allow no size arrays
- 2017-05-10 Contentmaudlin 346e155 new print case
- 2017-05-10 Lucas Schuermann 03f2ad3 Merge pull request #166 from shux-lang/semant
- 2017-05-10 Contentmaudlin c4d96ce noops
- 2017-05-10 Contentmaudlin f538ace Merge branch 'master' of github.com:shux-lang/shux into semant
- 2017-05-10 Contentmaudlin e0aea16 type checking proper for gn lookback
- 2017-05-10 Andy Lianghua Xu 9427715 closing the LLLitDud
- 2017-05-10 Lucas Schuermann 7c45898 Merge pull request #165 from shux-lang/cll
- 2017-05-10 Andy Lianghua Xu f65c12c crazy hell the program now compiles
- 2017-05-10 Andy Lianghua Xu 9d59fa7 working on translating blocks
- 2017-05-10 Lucas Schuermann c4399eb Merge pull request #163 from shux-lang/test_suite
- 2017-05-10 Lucas Schuermann c8c7ce4 final set of map tests
- 2017-05-10 Lucas Schuermann 2e4bfa6 map typecheck tests
- 2017-05-10 Lucas Schuermann 98eec65 the script should track warnings now
- 2017-05-10 j-hui f05e1f1 Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-10 j-hui eade081 ocaml is thirsty and so are them gurlz
- 2017-05-10 Lucas Schuermann 01e9ddc syntax fixes
- 2017-05-10 Contentmaudlin a95d582 Merge branch 'master' of github.com:shux-lang/shux into semant
- 2017-05-10 Contentmaudlin 4d41d57 building that lookback feature
- 2017-05-10 Lucas Schuermann 6bd1951 Merge pull request #162 from shux-lang/noop
- 2017-05-10 MertUssakli 4e2ac18 noop
- 2017-05-10 j-hui ebe0c32 fixed struct type prefix bug
- 2017-05-10 Lucas Schuermann 7083df1 Merge pull request #158 from shux-lang/test_suite
- 2017-05-10 Lucas Schuermann 1aad6da more complex block tests
- 2017-05-10 Lucas Schuermann 40c108a more tests for conditionals and fixes
- 2017-05-09 j-hui 8702fbb added type check thing instead ay
- 2017-05-10 Lucas Schuermann 28dc8e8 change nomenclature in makefile
- 2017-05-10 Lucas Schuermann 5db1c71 fix syntax
- 2017-05-10 Lucas Schuermann 4994a06 update lookback tests to disallow lookback on formals

- 2017-05-10 Lucas Schuermann 21d05ba fix syntax
- 2017-05-10 Lucas Schuermann 4f07c2f screw scalar literals
- 2017-05-10 Lucas Schuermann 9f2e34b fix vector test
- 2017-05-10 Lucas Schuermann 092f59e fix more test syntax
- 2017-05-10 Lucas Schuermann f1fb76c fix some struct tests
- 2017-05-10 Lucas Schuermann 7c19e75 testing maps
- 2017-05-09 j-hui 8b60d4e map my fap
- 2017-05-10 Lucas Schuermann 1eb7456 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-10 Contentmaudlin 5160f30 map type checking
- 2017-05-10 Lucas Schuermann 30df9db Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-10 Contentmaudlin a96ca76 fixed val initialization
- 2017-05-10 Lucas Schuermann bcc2e3a test filter type checking
- 2017-05-10 Lucas Schuermann ac4f697 some more null cases for filter
- 2017-05-09 Mert Ussakli 84ea7b9 Merge pull request #151 from shux-lang/test_suite
- 2017-05-10 Lucas Schuermann 465e9d5 Frontend tests
- 2017-05-10 Lucas Schuermann 0652798 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-09 Lucas Schuermann 19d6694 Merge pull request #152 from shux-lang/semant
- 2017-05-10 Contentmaudlin 47bbbb5 filter/map Ú
- 2017-05-10 Lucas Schuermann 8364549 more robust test script
- 2017-05-10 Lucas Schuermann 6d5635b Update with syntactic sugar
- 2017-05-10 Lucas Schuermann 63ea981 update lookback syntax
- 2017-05-10 Lucas Schuermann 44c733e update euler demo and some syntax
- 2017-05-09 j-hui f6bb1cf Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-09 j-hui dad5fbb down with the and and adn ad nad adn
- 2017-05-09 j-hui b5fffa0 changed the wat in cast to wat
- 2017-05-10 Contentmaudlin 5900698 merge
- 2017-05-10 Contentmaudlin 605a128 remove print
- 2017-05-10 Lucas Schuermann fa6a218 fixing typos/syntax errors
- 2017-05-10 Contentmaudlin 4e160ae Merge branch 'master' of github.com:shux-lang/shux into semant
- 2017-05-09 Mert Ussakli 0839ea6 Merge pull request #155 from j-hui/sast-cast
- 2017-05-10 Lucas Schuermann b4e8e11 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-10 Contentmaudlin 8f455b4 uhwfiuashof
- 2017-05-10 Contentmaudlin 2555602 merge
- 2017-05-09 j-hui 8fc2c0d changed sccope of SKnCall to SKnLambda
- 2017-05-10 Lucas Schuermann 43be4ec Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-10 Lucas Schuermann 57a31c8 more complete gn tests
- 2017-05-09 Andy Lianghua Xu 42636d3 Merge pull request #154 from j-hui/sast-cast

- 2017-05-09 j-hui 879c4cb changed const decl ay
- 2017-05-10 Lucas Schuermann 531e629 add bad lookback cases
- 2017-05-10 Lucas Schuermann 81c6e69 starting some block tests
- 2017-05-10 Lucas Schuermann e2f1959 add gn bad syntax test
- 2017-05-10 Lucas Schuermann 1e62433 add bad for example
- 2017-05-10 Lucas Schuermann e81100c add gn immut test
- 2017-05-10 Lucas Schuermann 51fa866 add gn for call test, change example
- 2017-05-09 Mert Ussakli 8d58f1c Merge branch 'master' into semant
- 2017-05-10 Contentmaudlin 843673e fixed slitarray
- 2017-05-10 Lucas Schuermann 790b071 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-09 Lucas Schuermann 68bc165 Merge pull request #150 from j-hui/sast-cast
- 2017-05-09 Mert Ussakli c6b6c92 Merge pull request #120 from shux-lang/test_suite
- 2017-05-09 Lucas Schuermann 30b3f32 working on larger example
- 2017-05-09 Lucas Schuermann 370a7dd removed duplicate examples
- 2017-05-09 Lucas Schuermann 4717596 small changes
- 2017-05-09 Lucas Schuermann 08904c0 merge in new code from master
- 2017-05-09 Lucas Schuermann 3f63f96 first round of test fixes for new syntax
- 2017-05-09 j-hui be42a6c ay squashed some bugz
- 2017-05-09 j-hui 92a952b ocaml eagerness is killing me
- 2017-05-09 j-hui c2d346d now actually compiles and doesn't throw around dumbass warnings everywhere
- 2017-05-09 j-hui 899e1cc now just prints a tonneee of warnings.
- 2017-05-09 j-hui 319e0d4 added something that doesn't die every every line of code, kinda dumb
- 2017-05-09 j-hui 40207dc because tags make everthing better
- 2017-05-09 j-hui ea642e2 added debug statemetns to SAST->CAST
- 2017-05-09 Contentmaudlin 57249d5 remove pattrn matching warning
- 2017-05-09 Contentmaudlin 0260c5e Merge branch 'master' of github.com:shux-lang/shux
- 2017-05-09 Contentmaudlin 619e556 remove do
- 2017-05-09 Mert Ussakli d4c78fe Merge pull request #149 from j-hui/sast-cast
- 2017-05-09 j-hui 76c62ff added some bug fixes due to recursion
- 2017-05-09 Mert Ussakli 17b14e2 Inherited bindings identified in lambdas and hoisted out (#148)
- 2017-05-09 Andy Lianghua Xu 151df1a Merge pull request #145 from j-hui/sast-cast
- 2017-05-09 Mert Ussakli 8b86d3d Merge pull request #146 from shux-lang/semant
- 2017-05-09 Andy Lianghua Xu 8605136 Merge pull request #147 from shux-lang/castllast
- 2017-05-09 Andy Lianghua Xu e3d0525 fully function pipeline, func decls, function formal and locals, struct defs done
- 2017-05-09 Andy Lianghua Xu 877eaa7 updated Makefile

- 2017-05-09 Contentmaudlin dd68741 sugared up do expressions ƛ
- 2017-05-09 j-hui cf74599 Shux 'do's considered harmful
- 2017-05-09 j-hui 368a61c fixed change with void expressions
- 2017-05-09 j-hui 570bd36 merged splurged
- 2017-05-09 j-hui 11ba3fb ay ay ay ay progressvim sast_cast.ml
- 2017-05-09 Andy Lianghua Xu eb905df added merlin config and Makefile update
- 2017-05-09 Contentmaudlin df6a1b1 extend pipelyne ʒ
- 2017-05-09 Contentmaudlin 90fc889 Merge branch 'master' of github.com:shux-lang/shux
- 2017-05-09 Contentmaudlin cd69023 fixed some tests
- 2017-05-09 Andy Lianghua Xu 7678f2f Merge pull request #128 from shux-lang/llast
- 2017-05-09 Andy Lianghua Xu 6ae939d Merge branch 'master' into llast
- 2017-05-09 Andy Lianghua Xu 950d243 added global variables, but only for constant int or doubles
- 2017-05-09 j-hui 48cdbb1 completed return by reference
- 2017-05-09 Contentmaudlin a02cdcc accessing immutability
- 2017-05-09 Contentmaudlin ad96d05 indexing fix on ast_sast
- 2017-05-09 Contentmaudlin 5bf88c3 arays A
- 2017-05-09 Contentmaudlin 7173e1c Merge branch 'master' of github.com:shux-lang/shux
- 2017-05-09 Mert Ussakli 33622dd AST->SAST enhancements (#143)
- 2017-05-09 Mert Ussakli 72d2aee Merge pull request #144 from j-hui/sast-cast
- 2017-05-09 j-hui f2997de generators implemented
- 2017-05-08 j-hui 4dae5b4 halfway through implementing generators
- 2017-05-09 Contentmaudlin aeb87e1 array return types, generators and kernels.. I have fixed so many bugs they can call me guilty of insect genocide at this point
- 2017-05-09 Contentmaudlin eca6fe0 extracting out max lookback value
- 2017-05-08 j-hui 8f81c24 array conditionals implemented, not all that different from primtives
- 2017-05-08 j-hui 84d462a finished array literals
- 2017-05-08 Mert Ussakli 58ecd2f Merge pull request #142 from j-hui/sast-cast
- 2017-05-08 Mert Ussakli 098b19a Merge pull request #141 from shux-lang/semant
- 2017-05-08 Contentmaudlin 11d78a9 fixed kernel and generator translation crashing
- 2017-05-08 j-hui 7ec7f2c done conditionals
- 2017-05-08 Contentmaudlin 8140d89 Merge branch 'master' of github.com:shux-lang/shux into semant
- 2017-05-08 Mert Ussakli e935cae More Semant and completed translation template (#140)
- 2017-05-08 Contentmaudlin 5d11f42 removed swap files
- 2017-05-08 Contentmaudlin 8a4f803 removed swap file

- 2017-05-08 Contentmaudlin ae36c2b completed ast_sast translation, semant fixes with lookback, integrated translation into entry point
- 2017-05-08 j-hui 20fca4a finished nested assign of primitives
- 2017-05-08 Contentmaudlin f3cbd1a disallowed Lookback on mutable types
- 2017-05-08 j-hui 193e85f finished access of struct primitives
- 2017-05-08 Contentmaudlin 434871c translating generators. hoisting done
- 2017-05-08 Contentmaudlin 17dea77 indentation machine broke
- 2017-05-08 j-hui 2e6448b finished binops?
- 2017-05-08 Contentmaudlin 58a2472 formatting kn function in ast_sast
- 2017-05-08 Mert Ussakli a7d18ee Merge pull request #139 from shux-lang/semant
- 2017-05-08 Contentmaudlin 91b0f45 bugfix
- 2017-05-08 Contentmaudlin f1beb4e merge conflict
- 2017-05-08 Contentmaudlin c226089 implemented type checking for lambda literals
- 2017-05-08 Contentmaudlin 7986489 lambda type bug with arrays fixed
- 2017-05-08 j-hui cf9b23b going to attempt weird refactor..
- 2017-05-08 Mert Ussakli b917e73 New Semant Changes and Bugfixes (#137)
- 2017-05-08 Contentmaudlin 31a75df initialization fix for assignments
- 2017-05-08 Contentmaudlin e0e8d49 implemented new array declaration policty
- 2017-05-08 Contentmaudlin 065f4a3 struct namespaces declarations
- 2017-05-08 Contentmaudlin 728928e fixed struct literal typings
- 2017-05-08 j-hui f8d2bfe ready to convert to DFS
- 2017-05-08 Lucas Schuermann 6e89bcb Merge remote-tracking branch 'origin' into test_suite
- 2017-05-08 Lucas Schuermann 5664553 Merge pull request #132 from j-hui/sast-cast
- 2017-05-08 Lucas Schuermann c114ded Merge remote-tracking branch 'origin' into test_suite
- 2017-05-08 j-hui ab51a61 tidied up code a little bit
- 2017-05-08 j-hui f2b3724 l-value walk (assignments) complete
- 2017-05-07 j-hui 6d93539 refactored to prepare for recursive lvalue translation
- 2017-05-07 j-hui 2342643 completed loop assignments
- 2017-05-07 j-hui d4e0325 hokay.
- 2017-05-07 j-hui 343b326 fixed conflict ayy
- 2017-05-07 j-hui 32d20b5 added comment about what it all means
- 2017-05-07 j-hui 68a112c completed unit walk_left
- 2017-05-07 Contentmaudlin e380d48 conflicts
- 2017-05-07 Mert Ussakli 9c3bc0d Namespaces implemented (#130)
- 2017-05-07 Andy Lianghua Xu 1cec465 done with passing structs and arrays into function calls
- 2017-05-07 j-hui 77ca118 added change to SAST to support lambda 'scope'
- 2017-05-07 j-hui f2845ca finished hoisting these bebes
- 2017-05-07 Andy Lianghua Xu 1e17508 implemented arrays and struct
- 2017-05-07 Andy Lianghua Xu 6013030 Merge pull request #127 from j-hui/sast-cast

- 2017-05-07 j-hui f330d44 kn_to_fn can now return multiple defns
- 2017-05-07 j-hui 0f4176f now compiles with updated CAST
- 2017-05-07 j-hui ae39145 pick an operator, any operator
- 2017-05-07 j-hui d54786d changed AST to werk werk werk werk werk
- 2017-05-07 j-hui b6d17d0 okay -> it is
- 2017-05-07 j-hui e07fab5 fixed astprint.ml, made lit struct also have string list
- 2017-05-07 j-hui e46d911 modified ast to match this bebe
- 2017-05-07 j-hui 6ec313f modified parser to take in this bebe
- 2017-05-07 j-hui de6e212 Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-07 John Hui 4510788 Hopefully CAST is stablised by now (#125)
- 2017-05-07 j-hui 4654948 added some annotations to how this guy is gonna werk
- 2017-05-07 j-hui 5aa6198 CAST complete?
- 2017-05-07 Mert Ussakli d87d233 Merge pull request #123 from shux-lang/semant
- 2017-05-07 Contentmaudlin afa0107 this commit kills fascists
- 2017-05-06 MertUssakli 51ab6e3 Merge branch 'semant' of github.com:shux-lang/shux into semant
- 2017-05-06 MertUssakli a371865 testing if those sexy namespace / struct conflicts are handled
- 2017-05-07 Contentmaudlin 9dd4ac6 fixed namespaces
- 2017-05-07 Lucas Schuermann 33f7c58 basic lookback testing
- 2017-05-07 Contentmaudlin fdbbcc4 working on namespaces.. full circle dude
- 2017-05-07 Lucas Schuermann 0d0b6ba cond fail test
- 2017-05-07 Lucas Schuermann 8bebf22 testing ternary operator ?
- 2017-05-07 Lucas Schuermann d88aeb1 gn basic
- 2017-05-06 Lucas Schuermann 0420bde Merge pull request #110 from shux-lang/test_suite
- 2017-05-06 MertUssakli cf89384 struct def arrays now have fixed sizes
- 2017-05-07 Lucas Schuermann 20d11a0 kn vector arg type check
- 2017-05-07 Contentmaudlin d56702d fix ast_sast for John's change in sast
- 2017-05-07 Contentmaudlin f94dd10 merge conflict
- 2017-05-06 Mert Ussakli b5ab840 Merge pull request #117 from j-hui/sast-cast
- 2017-05-07 Lucas Schuermann cacc581 similar tests for vectors
- 2017-05-07 Lucas Schuermann b62771c as usual, one for gn
- 2017-05-07 Lucas Schuermann f48c9e5 kn array arg without length test
- 2017-05-07 Lucas Schuermann d6738a2 update, add one for gn
- 2017-05-07 Lucas Schuermann 62edf7c kn array pass test
- 2017-05-06 MertUssakli f4e1c42 make sast translator compile with new sast
- 2017-05-07 Lucas Schuermann f851c8f for gn as well
- 2017-05-07 Lucas Schuermann 416c56d arg name conflict test
- 2017-05-06 MertUssakli cb53422 fix on vector literals to ensure float type
- 2017-05-07 Lucas Schuermann 890a1dc fix tests

- 2017-05-07 Lucas Schuermann 22e8044 add one for gn
- 2017-05-07 Lucas Schuermann de11aa8 more tests with globals
- 2017-05-07 Lucas Schuermann 12208e2 final batch of test fixes!
- 2017-05-06 Lucas Schuermann 279997c more test fixes
- 2017-05-06 Lucas Schuermann 24ec75f add test for struct immutability
- 2017-05-06 Lucas Schuermann 7144a91 fixing more tests based upon issues
- 2017-05-06 Lucas Schuermann 4d4527a fixed typo in vector test
- 2017-05-06 Lucas Schuermann ef4d739 update nested struct tests to follow convention
- 2017-05-06 Andy Lianghua Xu 8a3658d Merge pull request #92 from shux-lang/llast
- 2017-05-06 j-hui a0e03ed used CLoop
- 2017-05-06 j-hui f5cdddb stubbed out struct blocks
- 2017-05-06 j-hui 128b6db translated return
- 2017-05-06 j-hui bf915e1 Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-06 j-hui f9be017 finished l-value traversale for loops
- 2017-05-06 Mert Ussakli 6aee075 Translated Kernels to SAST (#112)
- 2017-05-06 MertUssakli 237b304 return expr option for gn
- 2017-05-06 MertUssakli 422893c fixed merge conflict
- 2017-05-06 Andy Lianghua Xu 192c95f Merge branch 'master' into llast
- 2017-05-06 Andy Lianghua Xu f99bd59 control flow works
- 2017-05-06 MertUssakli 0680d2a kernel translation to sast
- 2017-05-06 Lucas Schuermann 99ad52b add generator no return fail test
- 2017-05-06 Lucas Schuermann 3956efb add arg conflit test for kn
- 2017-05-06 Lucas Schuermann 3f1c851 add global used in main test
- 2017-05-06 Lucas Schuermann 45fdbbc add let used in gn test
- 2017-05-06 Lucas Schuermann 6c9535c add let used in kn test
- 2017-05-06 Lucas Schuermann c639bfa fixing dumb errors because it is too late
- 2017-05-06 Lucas Schuermann 9a8805a added let immutability test
- 2017-05-06 Lucas Schuermann 5bc9aae another let assign type fail test
- 2017-05-06 Lucas Schuermann bca7dd8 add let assign type check test
- 2017-05-06 Lucas Schuermann 2f35527 add let string test
- 2017-05-06 Lucas Schuermann 585d572 add let bool test
- 2017-05-06 Lucas Schuermann 9cf5bf2 add let scalar test
- 2017-05-06 Lucas Schuermann 6736db2 add let array test
- 2017-05-06 Lucas Schuermann d8cd422 aded global flat ns nested redefine test
- 2017-05-06 Lucas Schuermann 5f8b5ae add ns let check
- 2017-05-06 Lucas Schuermann 47f4add Merge pull request #102 from shux-lang/test_suite
- 2017-05-06 Lucas Schuermann 7df90f2 more tests
- 2017-05-06 Andy Lianghua Xu 2ecdffd Merge pull request #98 from j-hui/sast-cast
- 2017-05-06 j-hui a8bca7b updated SAST to have typed call actuals
- 2017-05-06 Lucas Schuermann 26db690 testing ns fanciness

- 2017-05-06 j-hui b94c8db Merge remote-tracking branch 'origin' into sast-cast
- 2017-05-06 Mert Ussakli 8131f6d Continued AST to SAST Translations (#108)
- 2017-05-06 MertUssakli f57c2b9 removed swap
- 2017-05-06 MertUssakli 254353b finished let decl translation
- 2017-05-06 Lucas Schuermann 0c37c99 really drilling into arrays
- 2017-05-06 Lucas Schuermann 6c81fad struct definition tests
- 2017-05-06 Lucas Schuermann 8b576ea more array tests with structs
- 2017-05-06 MertUssakli 4f3a025 translating letdecls
- 2017-05-06 Lucas Schuermann b4f1cf0 main in ns test
- 2017-05-06 Lucas Schuermann 9b0b771 more robust junk testing etc
- 2017-05-06 Lucas Schuermann 5fe3e20 more vector tests
- 2017-05-06 Lucas Schuermann 8e5c5bc debugging indexing
- 2017-05-06 Lucas Schuermann 9484048 add fail test for vector init
- 2017-05-06 Lucas Schuermann 7b9c29d add basic vector test
- 2017-05-06 Lucas Schuermann 431b66f more struct test assignment tests
- 2017-05-06 Lucas Schuermann 034444e more struct tests
- 2017-05-06 Lucas Schuermann c827cdb more complex struct tests
- 2017-05-06 Lucas Schuermann 1199e40 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-06 Lucas Schuermann 6d6fd5e ns array tests begin
- 2017-05-06 Lucas Schuermann dc9c1d2 Merge pull request #99 from shux-lang/structfix
- 2017-05-06 Contentmaudlin a8052aa fixed non-unique struct fields
- 2017-05-06 MertUssakli 6c68b6f finished check_sexpr
- 2017-05-06 Lucas Schuermann cbec5d4 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-06 Contentmaudlin 2696a64 temp fix for indexing
- 2017-05-06 Lucas Schuermann dec9065 more rigorous struct tests
- 2017-05-06 Andy Lianghua Xu 4baa8f6 compiling
- 2017-05-06 Lucas Schuermann c88ced0 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-06 Lucas Schuermann 7dc09cc more array tests, starting on struct tests
- 2017-05-06 Andy Lianghua Xu a0162ab update Makefile
- 2017-05-06 Contentmaudlin 98007c1 fixed assign to array type checking
- 2017-05-06 Andy Lianghua Xu 7741792 Merge branch 'master' into llast
- 2017-05-06 Andy Lianghua Xu ea57f99 update llast tree and translator
- 2017-05-06 Lucas Schuermann d16115e working with array tests
- 2017-05-05 Lucas Schuermann a8fde6d Merge pull request #93 from shux-lang/semant
- 2017-05-05 MertUssakli a8a2d0a ast to sast ready for first Pr
- 2017-05-05 MertUssakli 5c0e4a3 ast sast translation progress
- 2017-05-05 Contentmaudlin 4f765e4 laggy server i am sad
- 2017-05-05 j-hui db06bae updated CAST
- 2017-05-05 Andy Lianghua Xu c7e7cfd update makefile
- 2017-05-05 Andy Lianghua Xu ac8c84f added skeleton llast
- 2017-05-04 Contentmaudlin 5b81158 finished translating globals

- 2017-05-04 Contentmaudlin f98c17e started sast translation -- translated extern declarations
- 2017-05-04 Mert Ussakli d8341e0 Merge pull request #91 from shux-lang/semant
- 2017-05-04 Contentmaudlin 1a7efd8 made semant check for main
- 2017-05-04 Contentmaudlin 616551e merge conflict fix
- 2017-05-04 Contentmaudlin cdabd5c quick bugfix
- 2017-05-04 Contentmaudlin b0d67ca fixed formals and vardecl name conflicts not being detected
- 2017-05-03 Lucas Schuermann 643852b Merge pull request #78 from shux-lang/test_suite
- 2017-05-04 Lucas Schuermann c1191ab some basic array tests
- 2017-05-04 Lucas Schuermann 27b8991 Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-04 Lucas Schuermann 6afb551 small changes
- 2017-05-04 Lucas Schuermann d0006ea fix typo
- 2017-05-04 Lucas Schuermann a8d2122 updating test validity
- 2017-05-04 Lucas Schuermann 28016f7 more kn tests, some flat ns tests
- 2017-05-04 Lucas Schuermann 4eb66e3 kn test suite
- 2017-05-03 Lucas Schuermann 58a7c1f finished basic test suite, moving on to kn definitions
- 2017-05-03 Lucas Schuermann 894f775 MOAR TESTS
- 2017-05-03 Mert Ussakli 86a2ff9 Merge pull request #76 from j-hui/exceptions
- 2017-05-03 Mert Ussakli a87a456 Compile master (#77)
- 2017-05-03 Lucas Schuermann c288822 tester working properly now for null compile cases
- 2017-05-03 Contentmaudlin 185158b moved functions from ast to semant
- 2017-05-03 j-hui 0faee15 'self-documenting' code
- 2017-05-03 Contentmaudlin ca27729 integrated translations into shuxc and Makefile, updated semant with refactoring
- 2017-05-03 j-hui 9c31d95 okay i really don't know how to write exceptions in OCaml
- 2017-05-03 Contentmaudlin c63ca92 small fix to bind with new sast
- 2017-05-03 Contentmaudlin cecc17e Merge branch 'master' of github.com:shux-lang/shux into semant
- 2017-05-03 Lucas Schuermann 8745108 update pretty output of tester
- 2017-05-03 Mert Ussakli e86f613 Merge pull request #75 from j-hui/sast-cast
- 2017-05-03 Contentmaudlin df9c5a4 re-tractor
- 2017-05-03 Lucas Schuermann 35006a3 more ns tests
- 2017-05-03 Lucas Schuermann 61fe059 getting started with more ns/let testing
- 2017-05-03 j-hui 0f00869 added exceptions.mli
- 2017-05-03 MertUssakli 78f31a9 making assign operator better
- 2017-05-03 MertUssakli 9439c5f merge conflict
- 2017-05-03 MertUssakli 1a91c00 making assign operator better
- 2017-05-03 j-hui d943b6d made SStructs typ contain list of their bindings

- 2017-05-03 Contentmaudlin 31f4a21 working function type checker
- 2017-05-03 MertUssakli c39112c extra ast functions
- 2017-05-03 MertUssakli b31476e semantic checking of function blocks
- 2017-05-03 j-hui 050ea00 Merge branch 'master' of github.com:shux-lang/shux into sast-cast
- 2017-05-03 j-hui 791f036 translates non functional code now
- 2017-05-03 j-hui d03b223 this actually compiles without warnings now ay
- 2017-05-02 MertUssakli 7c0b831 Merge branch 'semant' of github.com:cerrno/shux into semant
- 2017-05-02 MertUssakli e2d54d6 remove compare_ast_typ
- 2017-05-02 Mert Ussakli 993610f Merge pull request #74 from shux-lang/semant
- 2017-05-02 Contentmaudlin ec0e6d3 this should actually live in ast
- 2017-05-02 MertUssakli 9d67d37 comparing ast typ func
- 2017-05-02 Mert Ussakli 3af972a Merge pull request #71 from shux-lang/test_suite
- 2017-05-02 Mert Ussakli 72e72e9 Merge pull request #72 from shux-lang/semant
- 2017-05-02 j-hui 53ec884 okay now this compiles
- 2017-05-02 j-hui 13fd79b still todos
- 2017-05-02 j-hui 5222565 stubbed out walk_kn
- 2017-05-02 j-hui fad006c fixed namespacing issue with vars
- 2017-05-02 j-hui 91121a9 refactored for single argument constructors
- 2017-05-02 j-hui 7f08160 gn -> kn first draft complete
- 2017-05-02 j-hui f50113a real maths for lookbacking
- 2017-05-02 j-hui d81b2c9 implemented lookback
- 2017-05-02 j-hui 6ad6593 implemented lookback
- 2017-05-02 j-hui 357f7f7 implemented SLookbackDefault
- 2017-05-02 j-hui 6fa737f access all gucci too
- 2017-05-02 j-hui 7fd40ae recursion complete {
- 2017-05-02 j-hui ecbb6e8 pattern matching across all exprs
- 2017-05-02 j-hui 9f2d5d2 function pattern matching ay
- 2017-05-02 j-hui cc150f3 function pattern matching ay
- 2017-05-02 j-hui 7414c60 all expressions become 42 holy shit
- 2017-05-02 j-hui 2b40044 gn_to_kn now takes in a struct formal
- 2017-05-02 Contentmaudlin 2c61030 added extern decl checking
- 2017-05-02 j-hui be6fb33 stubbed out Gn -> Kn types
- 2017-05-01 j-hui 3ad1366 kerchoo
- 2017-05-01 j-hui 245e0b9 prefixing in place
- 2017-05-02 Lucas Schuermann 80cd634 revert test code
- 2017-05-02 Lucas Schuermann 8a46afb robust failure testing
- 2017-05-02 Lucas Schuermann 3562474 new testing structure
- 2017-05-01 MertUssakli 8bfd100 remove .ml from build
- 2017-05-01 MertUssakli 3288099 basic global type checking implemented
- 2017-05-02 Lucas Schuermann d37bc57 Merge branch 'test_suite' of github.com:shux-lang/shux into test_suite
- 2017-05-02 Lucas Schuermann b8f4c1d fix typo
- 2017-05-02 Lucas Schuermann 805edd9 fix typo

- 2017-05-02 Lucas Schuermann da9386c Merge branch 'master' of github.com:shux-lang/shux into test_suite
- 2017-05-02 Lucas Schuermann e1ae3be now tracking errors in compilation for tests
- 2017-05-02 Lucas Schuermann 82bb0ce moar tests
- 2017-05-01 MertUssakli 8b772f1 done with check_expr yoyoyo
- 2017-05-02 Lucas Schuermann 5e502f4 starting to expand basic tests
- 2017-05-01 MertUssakli a907b3a function decl in trans_env, call in check_expr
- 2017-05-01 Lucas Schuermann 34ad417 more scripting fun
- 2017-05-01 Lucas Schuermann 25b8177 basic working test framework
- 2017-04-29 Lucas Schuermann 7c43415 update gitignore to ignore build dir
- 2017-05-01 MertUssakli 7129310 Merge branch 'master' of github.com:cerrno/shux into semant
- 2017-04-30 j-hui 1d0747b updated sast_cast.ml to bind properly
- 2017-04-30 j-hui 438b990 Merge remote-tracking branch 'origin' into sast-cast
- 2017-04-30 John Hui 0049361 couple of typos (#70)
- 2017-04-30 j-hui 44bd00a merged with ltest tree changes
- 2017-04-30 Mert Ussakli 462c025 Merge pull request #69 from j-hui/trees
- 2017-04-30 j-hui d452e54 finallly this compiles too
- 2017-04-30 j-hui 80a0fe0 Ptr and Void now prefixed with S
- 2017-04-30 j-hui 3054391 lookback and access are no longer binops, max iter added to gn
- 2017-04-30 Mert Ussakli 6c4cc82 Merge pull request #68 from j-hui/trees
- 2017-04-30 j-hui f26c90a updated astprint for array size
- 2017-04-30 j-hui 58c11ea arrays now have length
- 2017-04-30 j-hui 7f6248b added void type
- 2017-04-30 j-hui fa28df9 completed changes fro lookback and access
- 2017-04-30 j-hui b93ebb0 made changes to parser
- 2017-04-30 j-hui aee45f4 lookback and access are no longer binops
- 2017-04-30 j-hui e591bff Merge branch 'trees' of github.com:shux-lang/shux into trees
- 2017-04-30 j-hui 990a922 Merge branch 'master' into trees
- 2017-04-30 j-hui 02ec5f9 working on gn prefixes..
- 2017-04-30 MertUssakli 9d015d1 Merge branch 'master' of github.com:cerrno/shux into semant
- 2017-04-30 j-hui 5ce15e1 starting gn->kn conversion
- 2017-04-30 j-hui b055d79 refactoring from scratch
- 2017-04-30 j-hui 722182b added changes to sast
- 2017-04-30 MertUssakli 157d753 write code for Access operator -- note that theres a bug with how the access field is defined as an Id type
- 2017-04-29 MertUssakli 8bf9fc3 FINALLY coded up struct literal type checking
- 2017-04-28 j-hui 8b3c514 Merge branch 'master' into sast-cast
- 2017-04-28 Mert Ussakli dbd40a4 Merge pull request #67 from j-hui/fn-ptr
- 2017-04-28 j-hui ae6e7f8 i am harry potter i speak parser tongue

- 2017-04-28 j-hui d9ef23e ASTprint now no longer sad because unmatched connections
- 2017-04-28 j-hui eb4b01b Ptr type now added parser and AST i just dug my own grave
- 2017-04-28 Lucas Schuermann ab7d02f Merge pull request #66 from shux-lang/shuxcml
- 2017-04-28 j-hui d6ff19e added new ptr type
- 2017-04-28 j-hui 7c7ffb1 ugh where was i
- 2017-04-28 MertUssakli 09eb6a2 shuxc print spec if no args
- 2017-04-28 MertUssakli 1967825 struuuct
- 2017-04-28 MertUssakli a45aec5 Merge branch 'master' of github.com:cerrno/shux into semant
- 2017-04-28 Mert Ussakli f9ea4d0 Merge pull request #64 from shux-lang/structs
- 2017-04-28 MertUssakli baf07fd changed struct literals
- 2017-04-28 Mert Ussakli aab7ea7 Merge pull request #62 from j-hui/trees
- 2017-04-28 MertUssakli a0f7bec structing it up
- 2017-04-26 Contentmaudlin 40f6971 yep they are
- 2017-04-26 Contentmaudlin cbab623 tabs are really bad at ocaml arent they
- 2017-04-26 Contentmaudlin ff390e6 first completed draft of check_expr
- 2017-04-26 MertUssakli 809436c quality filter/map semant checking content
- 2017-04-26 MertUssakli 462000d Merge branch 'semant' of github.com:cerrno/shux into semant
- 2017-04-26 Contentmaudlin c3214a8 pushing this shit because google cloud is laggy as fuck and I want to work locally
- 2017-04-25 Contentmaudlin 6fdb2bd making progress on semant, building expr type checker
- 2017-04-23 Contentmaudlin 20ae9ad template for expr checking
- 2017-04-23 Contentmaudlin a09de73 test cases for namespaces--flattening name spaces works
- 2017-04-23 Contentmaudlin 435a03c Merge branch 'master' of github.com:shux-lang/shux into semant
- 2017-04-22 Mert Ussakli 8faa2ee Merge pull request #63 from j-hui/ns-fix
- 2017-04-22 j-hui 824e31f fixed namespace parsing
- 2017-04-18 Contentmaudlin c013639 done flat ns, have to test later
- 2017-04-18 Contentmaudlin 3044e07 resolved type conflicts from last night
- 2017-04-18 Contentmaudlin 9c4e8d5 flattening namespaces..
- 2017-04-18 Contentmaudlin 2962988 restore ast for ns
- 2017-04-18 Contentmaudlin a719c57 lol jk
- 2017-04-18 Contentmaudlin b950e0f a ground plan for semant
- 2017-04-18 Contentmaudlin 755b748 we dont need nested namespaces
- 2017-04-18 Contentmaudlin e9cd741 setting up semant translation
- 2017-04-15 j-hui b13e058 am tired
- 2017-04-15 j-hui 6b7a31a fixed some maddening type errors, one of many
- 2017-04-15 j-hui 657fa29 Merge branch 'trees' into sast-cast
- 2017-04-15 j-hui efa8ebe make for easier reading
- 2017-04-15 j-hui bb3d295 added styp to sexpr lists
- 2017-04-15 j-hui f815353 no more vector types, no more vector lits

- 2017-04-15 j-hui 12612e9 modificates
- 2017-04-15 j-hui a6b20be i done some lisp programming
- 2017-04-15 j-hui 9fb2404 updated code to use SGlobal vs SLocalVal and SLocalVar
- 2017-04-15 j-hui 4735e04 Merge branch 'trees' into sast-cast
- 2017-04-15 j-hui c223e79 scope now includes val vs var
- 2017-04-15 j-hui 3185432 now struct has counter declared too
- 2017-04-15 j-hui ae686a0 ay done with struct def'ns
- 2017-04-15 j-hui ba2b0a3 Merge branch 'trees' into sast-cast
- 2017-04-15 j-hui 277dd40 added fixed length array types
- 2017-04-15 j-hui aa177a6 fixed SLocal typo
- 2017-04-15 j-hui 0d24063 almost done with the declaration of structs for generators
- 2017-04-15 j-hui 2c66141 Merge branch 'trees' into sast-cast
- 2017-04-15 j-hui e49c9f4 took out scope for bind, those are self-evident dadoi
- 2017-04-15 j-hui b976c93 some namespacing progress
- 2017-04-15 j-hui 4ea18d5 need to think about namespacing, yet again
- 2017-04-15 j-hui 86ed4b3 Merge branch 'trees' into sast-cast
- 2017-04-15 j-hui e16fad7 added scope to SAST
- 2017-04-15 j-hui 6a96885 translation manslaition
- 2017-04-15 j-hui 043935b merged
- 2017-04-15 Mert Ussakli 51a68ec Merge pull request #61 from j-hui/trees
- 2017-04-15 j-hui 7681a25 forgot that formals are also vals
- 2017-04-15 j-hui 808fafd Merge branch 'trees' of github.com:shux-lang/shux into trees
- 2017-04-15 j-hui 9ab735d holy fuck, epiphany. pausing work here to push a change on another branch
- 2017-04-14 j-hui aa6790e tidied up code a bit, thought about some things
- 2017-04-14 j-hui 3552205 started implementing this, need deep map
- 2017-04-14 j-hui 4195cdb Merge branch 'trees' of github.com:shux-lang/shux into sast-cast
- 2017-04-14 John Hui f54ad3b First draft of CAST complete (#60)
- 2017-04-14 j-hui 6b3384b cast first draft complete
- 2017-04-14 j-hui eb53887 less clutter
- 2017-04-13 Mert Ussakli a5e10b3 Merge pull request #59 from j-hui/trees
- 2017-04-13 j-hui a6a2535 split function types
- 2017-04-13 Mert Ussakli 266be7e Merge pull request #58 from j-hui/trees
- 2017-04-13 j-hui 8894672 updated SAST to include hoisted variables
- 2017-04-13 Mert Ussakli f03cd7b Merge pull request #57 from j-hui/trees
- 2017-04-13 j-hui cfb95a7 fixed tabs
- 2017-04-13 j-hui e570aef split operators by types
- 2017-04-13 j-hui 4062303 conservative draft of cast
- 2017-04-13 j-hui ff38f6d split operators into categories for better code reuse at lower levels
- 2017-04-13 j-hui 25f83b6 some ocd things
- 2017-04-13 j-hui 9a66651 added some comments to sast
- 2017-04-13 j-hui 65cfde5 removed assignment as an operator from frontend

- 2017-04-13 j-hui 55a5f5e prepended s to struct def
- 2017-04-13 j-hui c5e84cd positive unary opereator is a nop
- 2017-04-13 j-hui 9736861 differentiate between int and float arithmetic operations
- 2017-04-13 j-hui b1e305c Merge branch 'trees' of github.com:shux-lang/shux into trees
- 2017-04-13 Contentmaudlin 70340ca sast also one vertical line in ast
- 2017-04-13 Contentmaudlin 45e1f55 no void kernels in sast
- 2017-04-12 j-hui 1711b03 move trees to backend
- 2017-04-13 Contentmaudlin d939d90 cast draft
- 2017-04-13 Contentmaudlin 6be03e4 lookback + fixes
- 2017-04-13 Contentmaudlin ab6fb75 Merge branch 'semant' of https://github.com/shux-lang/shux into semant
- 2017-04-12 Mert Ussakli 4641aee Merge pull request #56 from shux-lang/lookback-default
- 2017-04-12 John Hui 0f7df94 updated astprint.ml to include lookbackdefault
- 2017-04-12 John Hui 2c2f1c3 added lookbackdefault to AST
- 2017-04-12 MertUssakli f10e53b sast first draft
- 2017-04-02 Mert Ussakli fc9ae05 Merge pull request #53 from j-hui/extern-alias
- 2017-04-02 j-hui b35a529 added alias field to externs for namespace flattening
- 2017-03-28 Mert Ussakli 9b8691c Merge pull request #51 from shux-lang/assign_fix
- 2017-03-27 Andy Lianghua Xu 730765c add Assign op matching
- 2017-03-27 Andy Lianghua Xu 2facf22 Merge pull request #50 from j-hui/ast-sast
- 2017-03-27 j-hui 79c7510 astprint.ml now reflects new assign case
- 2017-03-27 j-hui af595de desugared assignment modification
- 2017-03-27 Andy Lianghua Xu 7c5d25c Merge pull request #49 from andylhxu/codegen
- 2017-03-27 Andy Xu f9c176d add autobuild using ocamlbuild
- 2017-03-27 Andy Xu ad89bf0 Makefile fixed indent
- 2017-03-27 Lucas Schuermann cbeb93a Merge pull request #48 from andylhxu/andy
- 2017-03-27 Andy Xu 07d0f56 default makes
- 2017-03-27 Andy Xu 6b18161 remove compiler warnings of unused func and vars
- 2017-03-27 Andy Xu 5f4427e remove the unused install_deps script
- 2017-03-27 John Hui ae134e1 shitty print to pretty print (#45)
- 2017-03-26 Mert Ussakli afdf175 Merge pull request #44 from shux-lang/andy
- 2017-03-26 Andy Xu 3f70b3a resolve merge conflict in Make
- 2017-03-26 Andy Xu 89cfc82 fixed PHONY targets, add clean and cleanall
- 2017-03-26 Mert Ussakli 7df624a Merge pull request #38 from cerrno/andy
- 2017-03-26 Contentmaudlin d1e8f7b modified shuxc, some codegen stuff, makefile hello hook enhanced to just run

- 2017-03-26 Contentmaudlin 20c9687 |
- 2017-03-26 Contentmaudlin af593f9 Merge branch 'andy' of github.com:cerrno/shux into andy
- 2017-03-26 Andy Xu fc0f575 codegen working!
- 2017-03-26 Contentmaudlin ac9a7b7 Merge branch 'andy' of github.com:cerrno/shux into andy
- 2017-03-26 Andy Xu 4b7b872 not at Blocking
- 2017-03-26 Contentmaudlin 19b6c23 Merge branch 'andy' of github.com:cerrno/shux into andy
- 2017-03-26 Andy Xu 17637f2 working on building statements of a function body
- 2017-03-26 Mert Ussakli 0e652e4 Merge pull request #43 from j-hui/parser
- 2017-03-26 j-hui 72a588a can now have functions without statements
- 2017-03-26 Mert Ussakli d20d46d Merge pull request #41 from cerrno/mert
- 2017-03-26 Contentmaudlin 7df28a6 some test cases, enhanced astprint and parser fixes
- 2017-03-26 Contentmaudlin 59991e6 Merge branch 'master' of github.com:cerrno/shux into mert
- 2017-03-26 Andy Xu 2594e1a done a dummy expr builder
- 2017-03-26 Contentmaudlin b4c71c3 fix vector lit
- 2017-03-26 Contentmaudlin d527b88 remove symlink
- 2017-03-26 Contentmaudlin 6c5ab72 u wot m8
- 2017-03-26 Contentmaudlin 8d0cb92 List.rev fixes
- 2017-03-26 Mert Ussakli b3e894c Merge pull request #39 from j-hui/pretty-print
- 2017-03-26 j-hui 8608c5a more cleanups
- 2017-03-26 j-hui af3d5e9 small cleanup
- 2017-03-26 j-hui 5e81a5e finished all lits except for lambdas
- 2017-03-26 Contentmaudlin d57128e Merge branch 'andy' of github.com:cerrno/shux into andy
- 2017-03-26 Andy Xu 9cb7862 fixed void type type deduction
- 2017-03-26 j-hui 993363e i'm dumb, string_of_list was written stupidly
- 2017-03-26 j-hui 1b94d38 string lits now are wrapped like tuna
- 2017-03-26 j-hui 4062044 factored in nop for easy reading
- 2017-03-26 j-hui 8f01fd8 prints vectors, using helper
- 2017-03-26 j-hui 3d16e7f now prints arrays
- 2017-03-26 j-hui 2847f6e literal cases stubbed out
- 2017-03-26 j-hui 46f2931 make pretty print pritty again
- 2017-03-26 j-hui f4594e1 now pretty prints struct types
- 2017-03-26 John Hui 4a98838 make make make again (#36)
- 2017-03-26 Andy Xu 5d4e129 Merge from master
- 2017-03-26 Andy Xu 8e85906 done upto function decls
- 2017-03-26 Andy Lianghua Xu fa76ac6 Merge pull request #33 from cerrno/mert
- 2017-03-26 Andy Xu 134c883 temporarily revised ast to allow llvm function declarator to work
- 2017-03-26 Andy Lianghua Xu 4f80bb3 Merge pull request #32 from cerrno/mert

- 2017-03-26 Contentmaudlin 73c8a13 Merge branch 'master' of github.com:cerrno/shux
- 2017-03-26 Contentmaudlin dbfc73c Merge branch 'master' of github.com:cerrno/shux
- 2017-03-26 Contentmaudlin 565a5a9 reverse list in func_decl
- 2017-03-26 Contentmaudlin 13161f0 oops, here's the actual thing
- 2017-03-26 Contentmaudlin 9ea70e3 close-to-complete implementation of astprinter.ml
- 2017-03-26 Contentmaudlin 2c7774a Merge branch 'master' of github.com:cerrno/shux
- 2017-03-26 Mert Ussakli d31aea0 Merge pull request #30 from cerrno/andy
- 2017-03-26 Andy Xu a87d4de Merge branch 'master' into andy
- 2017-03-26 Andy Xu b5436c0 fixed dep for ast
- 2017-03-26 Andy Lianghua Xu 2b965a0 Merge pull request #31 from cerrno/mert
- 2017-03-26 Contentmaudlin 27f8f8e fix example
- 2017-03-26 Contentmaudlin fb4d0cd check for main, basic pretty printing
- 2017-03-26 Andy Xu 4ca5a42 include examples
- 2017-03-25 Contentmaudlin b577b86 Merge branch 'master' of github.com:cerrno/shux
- 2017-03-25 Contentmaudlin f225b94 Merge branch 'master' of github.com:cerrno/shux into mert
- 2017-03-25 Mert Ussakli 47caa61 Merge pull request #29 from cerrno/andy
- 2017-03-25 Andy Xu a6f9d5d make twice..
- 2017-03-25 Andy Xu fcfb27d fixed the Makefile dep
- 2017-03-25 Contentmaudlin 0d3c774 merge
- 2017-03-25 Contentmaudlin c7c2969 merge merge
- 2017-03-25 Contentmaudlin c5834f0 semant beginning work
- 2017-03-25 Mert Ussakli cbc87c1 Merge pull request #28 from cerrno/andy
- 2017-03-25 Andy Xu 60b2571 change Makefile
- 2017-03-25 Andy Xu 670ac5b resolved conflict
- 2017-03-25 Andy Xu 70942fb shuxc.ml should be outside of frontend or backend
- 2017-03-25 Andy Xu 07b70c5 changed where shuxc.ml is located
- 2017-03-25 Andy Xu ac36e9b remove all Makefiles in src
- 2017-03-25 Andy Xu a4aba5f create build folder and have the build dir symblink flattened
- 2017-03-25 Lucas Schuermann 276d297 Merge pull request #27 from cerrno/luke_small_fixes
- 2017-03-25 Lucas Schuermann f47a426 Merge branch 'master' of github.com:cerrno/shux
- 2017-03-25 Lucas Schuermann 95d9c3f small setup fixes
- 2017-03-25 John Hui e936e03 Merge pull request #26 from cerrno/backend
- 2017-03-25 Lucas Schuermann 6676d0c Update README.md
- 2017-03-25 Contentmaudlin ae1adcf make hello hook
- 2017-03-25 Contentmaudlin 1df4fc8 makefile make a make make a make a make
- 2017-03-25 Contentmaudlin 4a4fdfb Merge branch 'master' of github.com:cerrno/shux

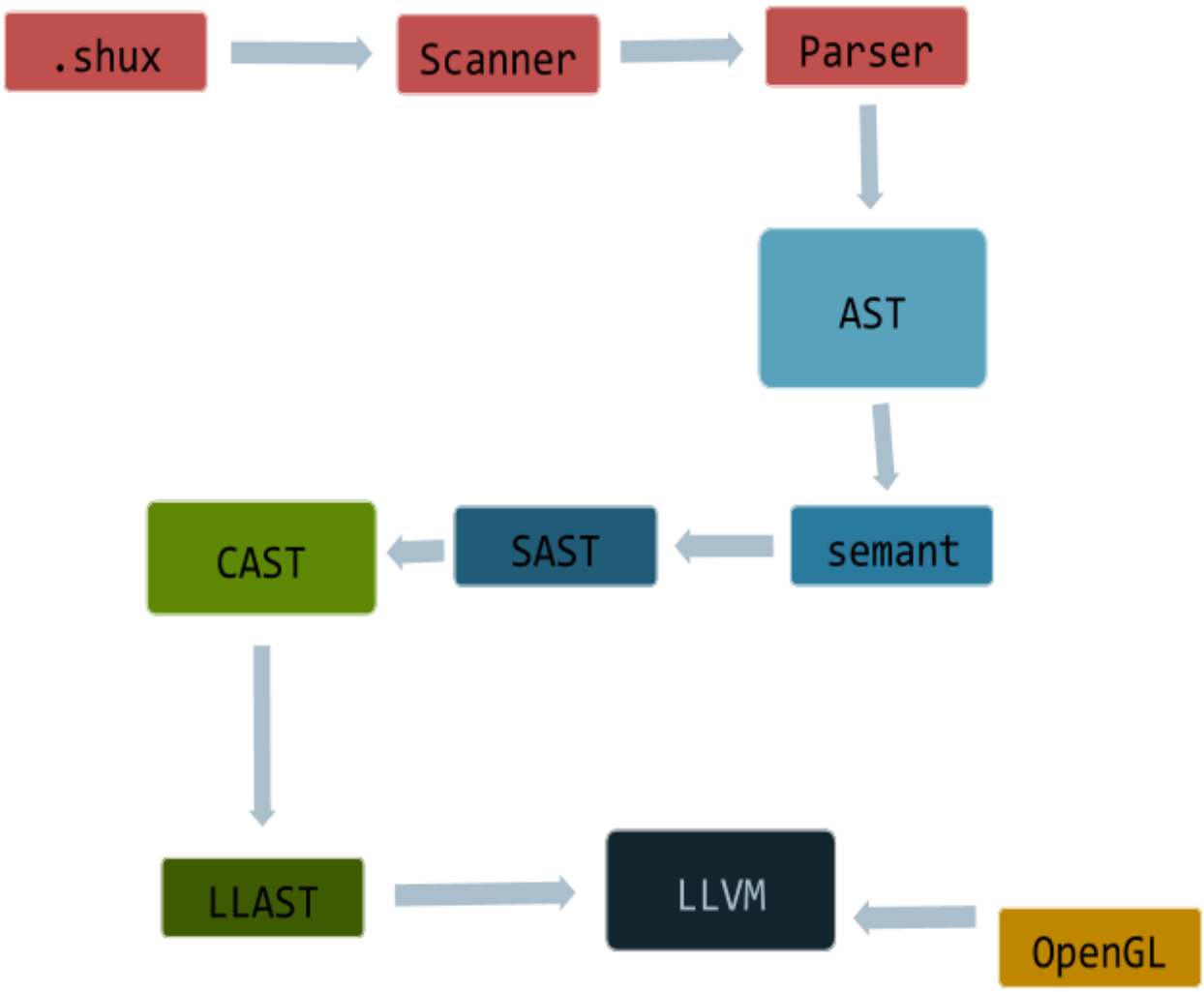
- 2017-03-25 John Hui 29734ae Merge pull request #25 from cerrno/backend
- 2017-03-25 Contentmaudlin 0aa5349 Merge branch 'backend'
- 2017-03-25 Contentmaudlin 9b999b9 extern)lol(
- 2017-03-25 Lucas Schuermann 6854aa9 Merge pull request #24 from cerrno/backend
- 2017-03-25 Contentmaudlin 3f3a6bd Merge branch 'frontend' of github.com:cerrno/shux into backend
- 2017-03-25 Lucas Schuermann 1cb6e6b Merge pull request #23 from cerrno/frontend
- 2017-03-25 Contentmaudlin e0633e8 sorry removed a
- 2017-03-25 John Hui 75f7639 Merge pull request #22 from j-hui/frontend
- 2017-03-25 John Hui 4998ba9 wow fuk trump
- 2017-03-25 Contentmaudlin eb44a72 fixed comments
- 2017-03-25 Contentmaudlin 89114d6 Merge branch 'master' of github.com:cerrno/shux into backend
- 2017-03-25 Mert Ussakli 796357a Merge pull request #20 from cerrno/frontend
- 2017-03-25 Contentmaudlin 7ec5e59 Merge branch 'frontend' of github.com:cerrno/shux into frontend
- 2017-03-25 Contentmaudlin d0a7843 i think we should just bring this over to frontend
- 2017-03-25 Mert Ussakli ce582c4 Merge pull request #18 from cerrno/ussakli-frontend
- 2017-03-25 Contentmaudlin 671f8f7 Merge branch 'ussakli-frontend' of github.com:cerrno/shux into ussakli-frontend
- 2017-03-25 Contentmaudlin 9bf1298 merged with current frontend
- 2017-03-25 John Hui 722c35e Merge branch 'frontend' into ussakli-frontend
- 2017-03-25 Lucas Schuermann a6a65a3 Merge pull request #19 from j-hui/hello-world
- 2017-03-25 John Hui 4a79c93 Merge pull request #17 from j-hui/frontend
- 2017-03-25 John Hui c1092d1 reverted .gitignore, going to do that in global .gitignore
- 2017-03-25 John Hui 692ff19 wrote hello world demo
- 2017-03-25 John Hui cd4b8e5 added scanner.ml to .gitignore
- 2017-03-25 John Hui b2b8e72 because nobody cares about scanner.ml
- 2017-03-25 Contentmaudlin 7b74cc0 injected john's nested comments fix
- 2017-03-25 Contentmaudlin 0f10f81 stdin if no argument
- 2017-03-25 Contentmaudlin 1bfa42b oops, supposed to be committing these
- 2017-03-25 Contentmaudlin 7f97c89 entry point, fixed makefile and dependencies
- 2017-03-25 John Hui 0be825e removed yacc associativity directives #trimthefat
- 2017-03-25 John Hui 9c2221e StructMember -> Access because we also use . for namespacing
- 2017-03-25 John Hui 371c307 now namespaces and let declarations are optional..
- 2017-03-25 John Hui 3f6b5c7 added extern ladoozy shmoozies
- 2017-03-25 John Hui 7faec48 added support for void return type

- 2017-03-25 John Hui adebb89 additional cleanup
- 2017-03-25 John Hui 684eca0 because vector<2> looks better than vector(2)
- 2017-03-25 John Hui 4b860e3 repeat after me: our code will not look like a flaming pile of garbage
- 2017-03-25 John Hui d428878 make sure these indents don't cramp on nobody's style
- 2017-03-25 John Hui 87b19e8 who needs pointers d00d
- 2017-03-25 John Hui 9737b8e manually pulled from Mert's branch
- 2017-03-25 John Hui e53524b updated scanner style and comments
- 2017-03-25 John Hui 6e28e82 why do ocaml derefences look so fuck ugly
- 2017-03-25 John Hui d719c39 updated some comments
- 2017-03-24 Mert 9063586 setting up the pipeline
- 2017-03-24 John Hui d7ec64a use (x,y) vector notation for great goods
- 2017-03-24 John Hui ef045a5 Merge pull request #15 from j-hui/frontend
- 2017-03-24 John Hui ee1d9dd added underscore for empty generator
- 2017-03-24 John Hui 8df79e1 factored out NoExpr
- 2017-03-24 John Hui a8965d2 ast now worksgit st
- 2017-03-21 John Hui b2c2492 typo of keyword
- 2017-03-21 John Hui 38dc7a3 fixed struct
- 2017-03-21 John Hui d2433fb use [[and]] for vectors
- 2017-03-21 John Hui 487fa1c merged
- 2017-03-21 Mert 0e0c33d give bad programmars a chance
- 2017-03-21 Mert 5f3aceb hemidemisemicolons are good for you
- 2017-03-21 Mert aa04e7f fixed SR conflicts
- 2017-03-20 John Hui 910703a took out LDBRACK stuff
- 2017-03-20 Mert 4497551 removed GT from follow set of expr
- 2017-03-20 John Hui 60792c2 moved lookback to end, still has S/R conflicts
- 2017-03-20 John Hui 016b1c3 formating doormatting
- 2017-03-20 John Hui 5566f77 finished parser.mly with 2 S/R conflicts
- 2017-03-20 John Hui bc2222f gone and done the hard bits
- 2017-03-20 John Hui 23fdf65 parser -> ast down to call
- 2017-03-20 John Hui c7521c1 done all the functional shits
- 2017-03-20 John Hui 9a9ae0e fixed formatting, mixing tabs and spaces is one hell of a drug
- 2017-03-20 John Hui c6c0074 we did not have fall-through for conditonal expr
- 2017-03-20 John Hui 3e52b43 parser->ast up until conditional exprs
- 2017-03-20 John Hui 57596ca added note about NoExpr, needs more thought
- 2017-03-20 John Hui f24343b parser->ast up until ret_expr, added NoExpr to AST
- 2017-03-20 John Hui 4c87f02 removed conditional vs iterative statements rule
- 2017-03-20 John Hui bebaaf1 completed parser->ast up to statements
- 2017-03-20 John Hui 959673e implemented parser -> ast up to struct decls
- 2017-03-19 John Hui 572d82d maybe this ast works now
- 2017-03-19 John Hui 9b1cdbc fixed all these compiler errors using and for corecursive types

- 2017-03-19 John Hui b7953e7 should be complete, now need to debug for syntax errors
- 2017-03-19 John Hui 4be6989 renamed to use auto syntax highlighting
- 2017-03-16 John Hui 0b4e596 picking up where we left off...
- 2017-03-05 Lucas Schuermann 3d7be9c Merge pull request #8 from cerrno/j-hui-frontend
- 2017-03-05 Mert 19494b3 added mod to mult_op
- 2017-03-05 Mert eddcbe6 added assignment expressions
- 2017-03-05 Mert 577d0f2 i am the parse man
- 2017-03-04 Mert 9a6b483 got parser to run with 20 sr conflicts
- 2017-03-04 Mert 97c4310 Merge branch 'frontend' of https://github.com/j-hui/shux into j-hui-frontend
- 2017-03-04 John Hui 8df615d added support for static initialisation
- 2017-03-04 John Hui dd9a221 added le decls and types
- 2017-03-04 John Hui 4a4acf6 completed expression sexpressions
- 2017-03-04 John Hui 81f2c66 completed grammar rules up to unit_exprs
- 2017-03-04 John Hui eadc681 did up until func decls
- 2017-03-04 John Hui 37bdb40 added tokens to parser
- 2017-03-04 John Hui d7161b8 added tokens from scanner
- 2017-02-25 Mert c0deb6e basic entry point for shux frontend
- 2017-02-25 Mert 71678ea Merge branch 'master' into frontend
- 2017-02-25 Lucas Schuermann 0cf8f8e Merge pull request #5 from cerrno/luke_frontend
- 2017-02-25 Lucas Schuermann e1f3045 fixed typo in readme
- 2017-02-25 Lucas Schuermann 8e8a4a6 add placeholder for ast work
- 2017-02-25 Lucas Schuermann 8859676 work on ast
- 2017-02-25 Lucas Schuermann d996368 work on operators in AST
- 2017-02-25 Mert 97b91df Merge branch 'luke_frontend' of github.com:cerrno/shux into luke_frontend
- 2017-02-25 Mert 27278a4 no notes in shux
- 2017-02-25 Lucas Schuermann 05ffb1e makefiles working now
- 2017-02-25 Lucas Schuermann 7dbadd7 more small fixes
- 2017-02-25 Lucas Schuermann 27ff651 rename directory because I'm an idiot
- 2017-02-25 Lucas Schuermann 609f0cc script -> +x
- 2017-02-25 Lucas Schuermann 97bdb78 more makefile/automation scripts work
- 2017-02-25 Lucas Schuermann 3295f6f getting started with some code taken from Note Hashtag for dependencies and the ast
- 2017-02-25 Lucas Schuermann 53c46a6 more directory structure/general work
- 2017-02-25 Lucas Schuermann a8059ef directory structuring and makefiles
- 2017-02-25 Lucas Schuermann d280437 added exponential operator ^
- 2017-02-25 Mert f3005d0 removed scanner.ml
- 2017-02-25 Lucas Schuermann 0166331 remove intermediate files from repo
- 2017-02-25 Lucas Schuermann 1fb0807 Merge pull request #1 from j-hui/frontend
- 2017-02-25 John Hui 4da59e3 changed algebraic type names
- 2017-02-25 John Hui f1b5118 alias 'float' to 'scalar'
- 2017-02-25 John Hui ad38a32 added some exceptions
- 2017-02-25 John Hui 3122881 added parser.mli for parser

- 2017-02-25 John Hui 8f74224 fixy wixy
- 2017-02-25 John Hui def38a2 added boolean literals
- 2017-02-25 John Hui bd5a06e line numbers in comments
- 2017-02-25 John Hui 71b0101 fixed dumb Makefile
- 2017-02-25 John Hui bc7e386 added dumbass Makefile so AP TAs rest peacefully
- 2017-02-25 John Hui 60b139f put into da diretoryyy
- 2017-02-25 John Hui 831bb3b added literals and ids
- 2017-02-25 John Hui 69978c8 exceptions with line numbers
- 2017-02-25 John Hui 2179c88 add more fancy fancies
- 2017-02-25 John Hui 9bcfca9 added teh fancy fancies
- 2017-02-25 John Hui ecc83ae comments can now nestgit add -u
- 2017-02-25 John Hui b4f610c fixed comments
- 2017-02-25 John Hui 80cf66c added comments no nesting
- 2017-02-25 John Hui 6cf28a5 started discrete scanner
- 2017-02-05 Lucas Schuermann 9148a17 Initial commit

Architecture



Scanner (Team)

Tokenizes the input file and pass the tokenized file to the parser.

Parser (Team)

Generates an abstract syntax tree from the list of tokens passed in by the scanner.

AST (John + Team)

To ensure the validity of our AST and the robustness of our frontend, we also wrote a pretty printer for our AST.

Semantic Checker (Mert)

shux has a strict type system, which necessitates a robust semantic checker. The semantic checker is implemented, largely adapted from some of the past project's like funk, with translation environments tree traversals across the program. The generally routine is pretty standard but complications develop from the features of shux: expressions, maps, filters, lambdas, generator calls and mutable arrays. Lambdas types aren't explicit, and the semantic checker is able to infer them and check them against the rest of the program. These details coupled with the strictness of shux's type system necessitated a robust semantic checker.

The development of the semantic checker was also heavily test driven: after the main functionalities were in place, the type checking system was tightened through assessment based on test cases.

SAST (Mert)

The initial intention was to create the SAST from the semantic checker, but eventually we decided on building a semantic checker the worked entirely statically and then subsequently writing a translator for a SAST. The SAST is mainly responsible for tagging every expression in our language with explicit types, hoisting out declarations to the top of function declarations, demarcating kernel/generator calls from one another, separating binary operators and extracting information from the AST to help with the later stages of the translation.

Particularly of significance when it comes to extracting info from the AST is the hoisting out of lookback values. The AST to SAST translator traverses through all of the generator calls, finds the maximum lookback value and tags all generators with this value. This is later used in the CAST translation stage to actually implement stateful values.

CAST (John)

The CAST is an intermediate form of representation that serves as a platform for implementation. Inspired by C-like semantics, the CAST buffers the translation process from the higher level, richly type-annotated SAST to the lower level, register-aware LLAST.

Before performing the actual translation to the CAST, the SAST to CAST translator first performs several passes through the SAST to perform several conversions, most notably:

1. Convert generator definitions to kernel definitions, and produce its corresponding argument struct definition
2. Hoist lambdas and declare them as kernel functions, and replace all invocations of lambdas with IDs to their generated names

Afterwards, there is a lengthy translation process that is needed to take into account assignment by value and by reference, recursively evaluates array and struct literals and assignments.

Where the CAST differs from C is its simplified scoping policy – instead of needing to explicitly manage anonymous temporary values, the CAST uses a “Push” statement in order to allocate a variable which all enclosed statements may “Peek” at.

LLAST (Andy)

The LLAST is the last layer of our translation pipeline. I designed LLAST to be the syntax tree of llvm programs optimized in a way that it's easy to be built by the ocaml binding. The goal of having this syntax tree is not just to simply add another level of abstraction but to support the upper stages of the translation without their need to ever worrying about 1) details on how ocaml-llvm binding is used and how it generates LLVM IR 2) to translate from CAST and LLAST, only two trees need to be referred and taken care of, without the need to understanding how higher level features of the language or understanding the lower level issues with LLVM IR.

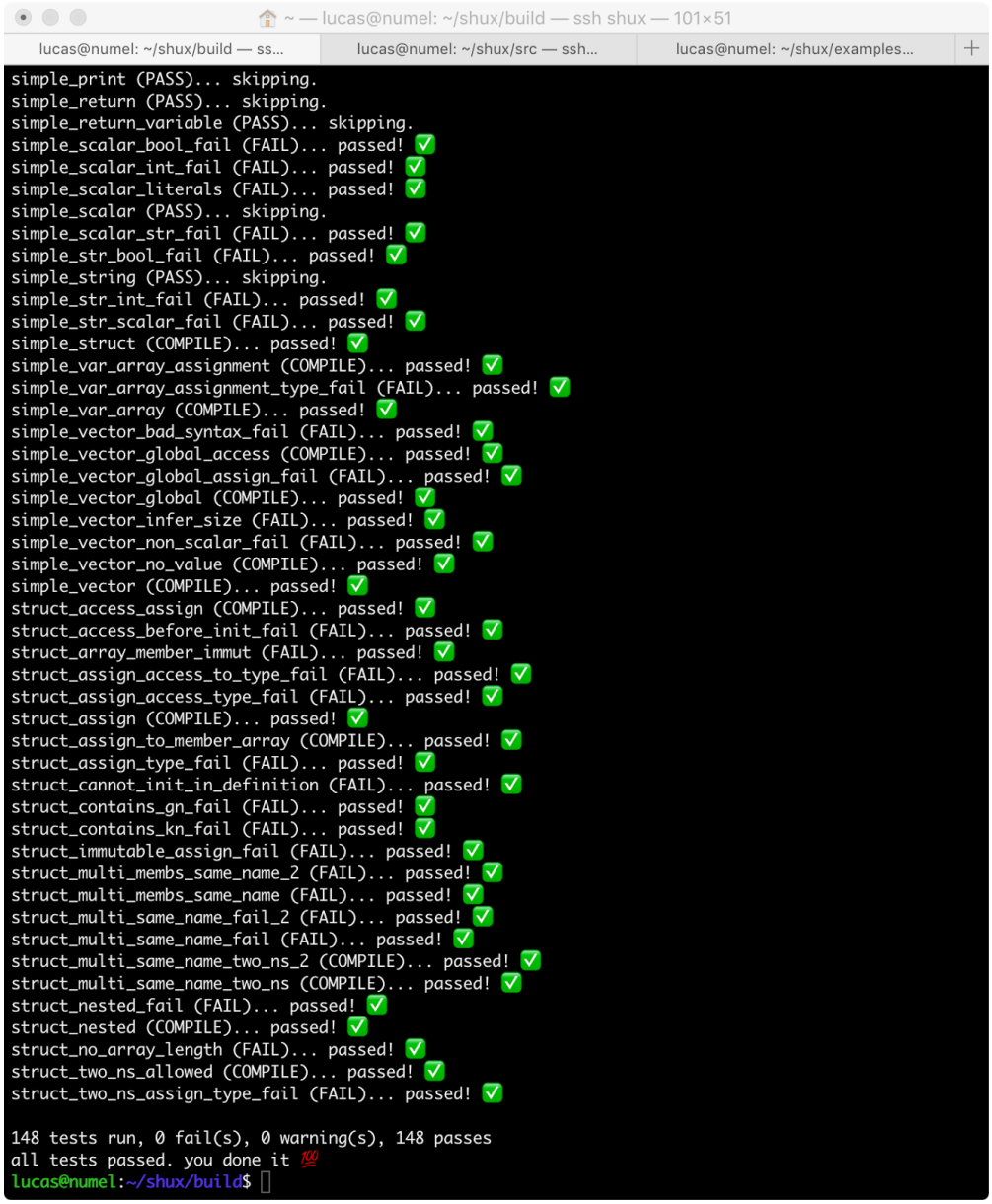
There are several features of LLAST worth noticing from my implementation:

- LLVM registers are "protected" from being touched by higher levels. All variables in LLAST are essentially stack variables referred by their names. For instance, in order to carry out two nested binary operations, the upper layer has to tell the LLAST names of the two variables, and the name of the temp variable to put to, and then the third variable name to add the temp variable to.
- LLVM variable names and branch names are generated and automatically mapped in the CAST->LLAST stage, this allows us to generate annonymous temp variables to be used later.
- LLVM bindings

Standard Library Bindings (Luke and Andy)

A full set of OpenGL bindings in native LLVM were implemented. By means of linking against

Testing (Luke)



Automation

A large suite of automated tests were used to thoroughly verify every semantic aspect of the language. Test cases could check for warnings/errors during compilation, keywords in AST printing, desired crash circumstances or correct outputs, or verified correct semantic checking.

Tests were bundled in the tests directory along with addendums added to the Makefile in the build directory, allowing for the quick testing of new changes before they were pulled into master. New tests were added frequently as more cases came up, more features were added, or new pipeline pieces came online. These were constantly merged into master to facilitate everyone using as complete of a test set as possible.

Coverage/Design

The tests directory contains over 150 test cases. These cases were used extensively to test all parts of the frontend, including assignment, operators, printing, comments, control flow, arrays, blocking, user-defined types, the standard library, generators, kernels, and special syntax for maps and filters. Special attention was given to verifying the type checker and the ability to infer types from lambdas in complex expressions such as maps. Null tests were added in every conceivable opportunity to verify that incorrect syntax or behavior would not be accepted. A non-exhaustive list of some test cases considered follows:

refactor

- var keywords
- namespaces ->
- := do/for

static

- int
- string
- scalar
- bool
- comment
- print
- return
- bad int <- string
- bad int <- scalar
- bad int <- bool
- bad string <- int
- bad string <- scalar
- bad string <- bool
- bad scalar <- int
- bad scalar <- string
- bad scalar <- bool
- bad bool <- int
- bad bool <- scalar
- bad bool <- string
- bad comments
- bad print statement

- bad return (two) ???
- multiple main
- bad no main
- kn single args
- kn mult args
- many kn
- bad many kn same name
- many kn different signature ???
- bad kn same arg name
- bad kn return
- bad kn arg redefine
- bad multi kn return
- bad kn same name in ns
- bad main no return
- array
- bad array type assignment
- bad array length specification
- array length inference
- var array
- var array allows assignment
- bad var array type assignment
- bad not var assignmentvd
- instantiating struct
- creating struct
- bad creating struct wrong types
- bad struct multi members same name
- bad many struct global same names
- allowed struct same name two namespaces
- bad struct no array length
- bad struct init values given in definition
- bad not allowed to assign to struct member immutable array
- allowed to assign to struct member mutable array
- bad not allowed to assign to ns member immutable array
- allowed to assign to ns member mutable array
- bad non-global namespace
- bad struct same name two namespaces assignment type conflict
- nested struct allowed
- bad kn in struct
- bad gn in struct
- bad lookback inside kn
- struct access assignment
- bad struct access before assignment
- bad struct access assign wrong type

- bad struct access assigned to wrong type
- bad struct assign without var
- vector define
- bad vector define with non-scalar
- vector define without assign
- bad vector init syntax
- bad assign to immutable vector
- bad main defined inside of namespace
- global flat ns multiple
- bad global flat ns multiple members same name
- global flat ns
- global flat ns only with kns
- global flat ns multiple with multi members
- global flat ns complex types
- global flat ns arrays
- global flat ns gns
- global flat ns complex composite test 1
- global flat ns complex composite test 2
- global nested ns redefine parameter allowed
- global ns access members in main
- global ns access type checked with let
- global arrays
- global scalars
- global bools
- global strings
- global assignment type checking 1
- global assignment type checking 2
- bad global assign overwrite immut
- global used in kn
- global used in gn
- global used in main
- bad global without let
- global conflict with arg name kn
- global conflict with arg name gn
- bad global conflict with arg name kn in ns (allowed)
- bad global conflict with arg name gn in ns (allowed)
- bad global conflict with field name in ns (allowed)
- bad kn arg name conflict in ns (allowed)
- bad gn arg name conflict in ns (allowed)
- bad kn arg name conflict in body
- bad gn arg name conflict in body
- passing arrays into kn
- passing arrays into gn

- passing arrays without size into kn
- passing arrays without size into gn
- passing vectors into kn
- passing vectors into gn
- passing vectors without size into kn
- passing vectors without size into gn
- gn basic
- gn lookback
- gn calling do
- gn calling for
- bad gn call syntax
- bad gn def syntax
- bad lookback syntax (type)
- bad loopback syntax (variable)
- lookback not allowed in kernels
- gn no return
- gn multi same name
- gn arg redefine
- gn define only once
- conditional if
- nested if
- if else
- cond ? syntax
- dual cond ? syntax
- map syntax
- filter syntax
- bad map syntax (type)
- bad filter syntax (type)
- bad map syntax (keywords)
- bad filter syntax (keywords)

libraries/runtime

- library extern
- graphics library calls with types
- graphics library calls (advanced)
- graphics libraries
- graphics runtime init
- graphics runtime simple
- graphics runtime complex
- graphics runtime real-time render
- print
- print_int

demos

- particle moving on screen
- Euler
- Fibonnaci

Lessons Learned

Luke

Start early. Code often. To quote Jae Woo Lee, “you are a flower.”

Project management on a macro level was solid: a shared virtual machine for development and good guidelines and pull request reviews etc. helped us maintain a solid, relatively monolithic codebase. On a micro level, our biggest advantage, becoming specialized on different parts of the pipeline for which we were responsible, also led to a much larger amount of time spent on resolving differences between implementations, trying to read each other’s code, and so on, even when good specifications were put in place for interaction between the stages.

While this led to endless headaches attempting to resolve bugs that emerged at different layers of the pipeline, it also allowed us to undertake a massively ambitious project, with our final cumulative progress putting us very very close to a working set of language features, with filter being the main one left unimplemented. Working so rigorously on the multistage pipeline was a fantastic learning experience--one which I believe will benefit us far more on an ongoing basis as compared to fully (many times over) implementing a basic language.

John

OCaml doesn’t play well with lazy evaluation – this isn’t even a joke about productivity, portions of dead code were eagerly executing due to name declarations not having arguments.

On a more serious note, designing the intermediate stages of abstraction was really an eye-opener to the kind of internal bookkeeping that compilers and intepreters must perform during translation. The entire translation stage was also a great exercise in thinking about recursion, and how to effectively use algebraic typing and pattern matching for relatively clean and concise code. My experience working with OCaml validated my appreciation with typing, and only in encountering difficulty in constituting our own typing policies did I realise that they’re a lot more than just a convenience.

Finally, in terms of working as part of team, while I saw the benefits of specialisation and parallelising workflow, I also realised the importance of working together and pair programming. Most of my bugs surfaced when other people were testing their code, and vice versa – it turns out that the integrity of a pipeline is only as robust as its weakest link, and errors may silently but fatally propagate throughout the pipeline to cause a lot of trouble.
People should specialise, but at the same time it’s important for everyone to know what’s going on in the project, every step of the way.

Mert

Our frontend isn't only the first stage of our pipeline to be completed but also the most robust one. This is mostly due to the way it has been developed: we wrote a pretty ast printer and wrote tests and tirelessly threw unit tests at them. The later stages of the pipeline were developed in relative isolation and this proved to be more problematic than we anticipated eventually. Write unit tests, make plans and commit to them, be communicative about what you have in mind.

And most importantly, don't be intimidated! A lot of the things about this project seems scary (like OCaml, writing a semantic checker or generating LLVM code) turn out to be not only a lot simpler but also a lot of fun once you dig into it. It is best to just go at it right away and churn out as much code as possible. In the end, you may just find yourself coding OCaml code in your spare time.

Andy

Adding another layer of indirection leads to exponentially more complication. [Sad.]

Don’t rely on the face that adding additional layers will simplify working with the current set.

Future Plans

More than 4000 lines of code in and several stages of the pipeline complete, shux still has yet more work to do. We believe our language is incredibly potent and are looking to completing the full promised set of functionality after this class is over.

Short term plans are to complete functionality that is not implemented: finish the translation of filters, make array literal type inference more robust, complete the CAST -> LLAST -> LLVM pipeline completely and test it super rigorously.

Long term plans focus a lot on refactoring. This was our first time writing a compiler and also first time implementing a major software project in OCaml, and a lot of software design choices were made early on that were perhaps not necessary. The verbosity of our semantic checker or the highly recursive, almost Lisp-like closure focused CAST translator could be rebuilt in simple, more efficient and more readable terms. Some of our trees could be redefined. Further, coding style could be standardised, with a set of well-defined standard internal library functions.

There's more functionality to be added. We need an interpreter to interpret the global namespace in compile time. We need better-integrated OpenGL graphics calls baked into our own language. Our language as a physics simulation language currently lacks a useful library: matrix manipulation, a mathematics library, perhaps even built-in integrators could all be super useful. We also realised too late that a lot more thought needs to be put into typing, specifically with regards to array sizes – on one hand, we need to support dynamically sized arrays for things like filters and fully functional generators. Finally, since generators reprsent a

We hope that our considerations of these suggested features only show the extent to which we aspired shux to become a full-fledged, equipped language. We really enjoyed working on this project and are already deliberating on what to do with our codebase after the end of the semester.

Appendix

Sample programs, code listings, and tests. **Code filenames follow a similar organizational structure to the task delineation above, giving information about who wrote what.**

[LVSTODO]

./shux/README.md
```
0001    # shux
0002    A programming language for particle-based physics simulation
0003
0004    # Introduction
0005
0006    Smoothed Particle Hydrodynamics will henceforth be the bane of
our existence.
0007
0008    # Authors
0009    - [Lucas Schuermann](http://lvs.io/)
0010    - John Hui
0011    - Mert Ussakli
0012    - Andy Xu
0013
0014    # Compiling
0015
0016    Run the `install_deps.sh` script to set up submodules, install
OCaml, and install third-party libraries.
0017
0018    This script is maintained on Ubuntu 15.04, although it will
probably work on Ubuntu 14/16.x,
0019    macOS 10.10+, and Debian as well.
0020
0021    Once you have the dependencies, just `cd` into the directory
you want and `make`.
0022
0023    # Contributing
0024
0025    1. Grab the lastest master: `git checkout master && git pull
origin master`
0026    2. Create your branch: `git checkout -b alice_feature_name`
0027    3. Make some changes: `git commit ...`
0028    4. Squash commits: `git rebase -i master`, then change
everything except the first commit to `squash`
0029    5. Rebase on the latest master: `git checkout master && git
pull origin master && git rebase master alice_feature_name`
0030    6. Run tests: `make symlink && make && make tests`
0031    7. Push for code review: `git push -f origin
alice_feature_name` (only use push -f on feature branches, not master)
0032
0033    # Syntax Highlighting
0034
0035    This would be nice to eventually have...
```

./shux/.merlin

```
0001    S src/frontend
0002    S src/backend
0003    S src/
0004    B build/
0005    PKG llvm
```

```
./shux/tests/simple_int.shux
0001    kn main() int {
0002        int a = 2;
0003        int b = 5057598000;
0004        0
0005    }
```

```
./shux/tests/simple_main_return_twice_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_multi_main_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_bool.test
0001    PASS
```

```
./shux/tests/simple_array_ns_assign.test
0001    COMPILE
```

./shux/tests/simple_bool_str_fail.shux
```
0001    kn main() int {
0002       var bool x = false;
0003       x = "hi";
0004       0
0005    }
```

./shux/tests/simple_bool.shux
```
0001    kn main() int {
0002        bool a = true;
0003        bool b = false;
0004        0
0005    }
```

./shux/tests/global_flat_ns_member_same_name_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/non_global_ns_fail.shux
```
0001    kn main() int {
0002       ns bad = {
0003          int x = 5;
0004          int y = 6;
0005       }
0006       0
0007    }
```

./shux/tests/simple_array_fail_type.shux
```
0001    kn main() int {
0002       int[2] a = [2,3];
0003       string b = a[0];
0004       0
0005    }
```

./shux/tests/simple_int_bool_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_cond_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_scalar.shux
```
0001    let scalar b = 3.5;
0002
0003    kn main() int {
0004       0
0005    }
```

./shux/tests/simple_array_ns_assign_immut_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_array_2.shux
```
0001    kn main() int {
0002       scalar[3] a = [2.0,4.0,3.0];
0003       scalar b = a[1];
0004       0
0005    }
```

```
./shux/tests/gn_lookback_fail_2.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_global_without_let_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_bool_int_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/global_flat_ns_advanced_types.test
0001    COMPILE
```

```
./shux/tests/kn_return_multi_fail.shux
0001    kn foo(int a) int {
0002       0
0003       0
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

```
./shux/tests/simple_array.test
0001    COMPILE
```

./shux/tests/struct_array_member_immut.shux
```
0001    struct foo {
0002        int[2] a;
0003    }
0004
0005    kn main() int {
0006       struct foo my_foo = foo { .a=[2,3] };
0007       my_foo.a = [1,2]; /* should be invalid */
0008       0
0009    }
```

./shux/tests/gn_bad_syntax_define_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_array_inferred_length.test
```
0001    COMPILE
```

./shux/tests/simple_vector_no_value.test
```
0001    COMPILE
```

./shux/tests/struct_multi_membs_same_name_2.shux
```
0001    struct particle {
0002       int x;
0003       scalar x;
0004    }
0005
0006    kn main() int {
0007       int x;
0008       int x;
0009       0
```

```
0010    }
```

./shux/tests/kn_many_one_name_global_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_map_type_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/gn_arg_name_conflict.shux
```
0001    gn foo(int a, int b) int {
0002       int a = 5;
0003       0
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/struct_assign_access_to_type_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_cond_fail.shux
```
0001    kn main() int {
0002       int a = (2 == 2) ? 0;
0003       a
0004    }
```

./shux/tests/simple_vector_bad_syntax_fail.shux
```
0001    kn main() int {
0002       vector g = (0.0, -9.81);
0003       0
0004    }
```

./shux/tests/let_assign_type_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_no_return_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/blocks_if_else_2.shux
```
0001    kn main() int {
0002       int a = 5;
0003       int b = 3;
0004       var int c = 0;
0005       if(a == b) then (c = 6) else (c = 2);
0006       c
0007    }
```

./shux/tests/simple_return_variable.shux
```
0001    kn main() int {
0002        int a = 2;
0003        a
0004    }
```

./shux/tests/simple_array_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_map_type_fail.shux
```
0001    kn main() scalar {
0002       int[5] a = [1,2,3,4,5];
0003       int[5] b;
0004
0005       var scalar sum = 0;
0006       a @ (scalar v) -> {
0007          sum = sum + v;
0008       };
0009
0010       sum
0011    }
```

./shux/tests/let_used_kn.shux
```
0001    let int a = 5;
0002
0003    kn foo() int {
0004       a
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/struct_nested_fail.test
```
0001    FAIL
```

./shux/tests/global_flat_ns_member_same_name_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_scalar.shux
```
0001    kn main() int {
0002        scalar a = 2;
0003        scalar b = 2.89248725;
0004        scalar c = 12.37e-17;
0005        scalar d = 1e3;
0006        0
0007    }
```

./shux/tests/global_flat_ns_multiple_members.shux
```
0001    ns foo = {
0002        let int x = 4;
0003        let scalar x = 4.0;
0004        let string x = "potato";
0005    }
0006
0007    kn main() int {
0008        0
0009    }
0010
```

./shux/tests/gn_local_immut_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_vector_non_scalar_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_main_return_invalid_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_cond.shux
```
0001    kn main() int {
0002       int a = 3;
0003       int b = 5;
0004       int z = (a == b) ? 5 : (a > b) ? 3 : 2;
0005       z
0006    }
```

./shux/tests/simple_str_scalar_fail.shux
```
0001    kn main() int {
0002       var string x = "hi";
0003       x = 423.534524;
0004       0
0005    }
```

./shux/tests/simple_no_main.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_int_scalar_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_used_gn.test
```
0001    COMPILE
```

./shux/tests/gn_vector_pass.test
```
0001    COMPILE
```

./shux/tests/kn_arg_redefine_fail_2.shux
```
0001    kn foo(int a, scalar b) int {
0002       string a = "hi";
0003       string b = "potato";
0004       0
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/blocks_if_else.test
```
0001    COMPILE
```

./shux/tests/simple_array_access.test
```
0001    COMPILE
```

./shux/tests/kn_lookback_fail.shux
```
0001    kn foo(int a, int b) int {
0002       int i = i..1 : 0;
0003       i
0004    }
0005
0006    kn main() int {
```

```
0007        0
0008    }
```

./shux/tests/kn_many_different_sig.shux
```
0001    kn foo(int a) int {
0002        0
0003    }
0004
0005    kn foo(string b, scalar c) int {
0006        1
0007    }
0008
0009    kn main() int {
0010        0
0011    }
```

./shux/tests/simple_cond_2.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_let_type_check_fail.shux
```
0001    ns foo = {
0002        int a = 5;
0003    }
0004
0005    let scalar b = foo.a;
0006
0007    kn main() int {
0008        0
0009    }
```

./shux/tests/simple_vector_global.shux
```
0001    let vector<2> g = (0.0, -9.81);
0002
0003    kn main() int {
```

```
0004        0
0005    }
```

./shux/tests/simple_bool_scalar_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/struct_assign_type_fail.shux

```
0001    struct particle {
0002        int x;
0003        int y;
0004    }
0005    let struct particle dummy = particle { .x=1.0; .y="potato" };
0006
0007    kn main() int {
0008        0
0009    }
```

./shux/tests/graphics_valid_call_test.shux

```
0001    extern print(string s);
0002    extern graphics_init(string title, int window_w, int
window_h);
0003    extern graphics_loop(_ptr render_func, _ptr update_func);
0004    extern graphics_set_points(scalar[][] buffer);
0005
0006    kn my_render() {
0007        scalar[2][2] points = [[1,2],[3,4]];
0008        graphics_set_points(points);
0009    }
0010
0011    kn my_update() {
0012        print("in update loop");
0013    }
0014
0015    kn main() int {
0016        graphics_init("shux Demo", 150, 100);
0017        graphics_loop(my_render, my_update);
0018
```

```
0019        0
0020    }
```

./shux/tests/simple_vector.test
```
0001    COMPILE
```

./shux/tests/simple_int.test
```
0001    PASS
```

./shux/tests/gn_call_for_bad.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_assign.test
```
0001    COMPILE
```

./shux/tests/gn_array_pass.shux
```
0001    gn foo(int[2] a) int {
0002        a[0]
0003    }
0004
0005    kn main() int {
0006        0
0007    }
```

./shux/tests/simple_int_str_fail.shux

```
0001    kn main() int {
0002       var int x = 1;
0003       x = "potato";
0004       0
0005    }
```

./shux/tests/simple_map_syntax_fail.shux
```
0001    kn main() int {
0002       int[5] a = [1,2,3,4,5];
0003       int[5] b;
0004
0005       var int sum = 0;
0006       a @ v -> {
0007          sum = sum + v;
0008       };
0009
0010       sum
0011     }
```

./shux/tests/simple_var_array_assignment.test
```
0001    COMPILE
```

./shux/tests/simple_scalar_literals.shux
```
0001    kn main() int {
0002       var scalar x = 2.0;
0003       scalar y = 2.0e5;
0004       x = 2.0e-5;
0005       0
0006    }
```

./shux/tests/simple_comment.shux
```
0001    kn main() int {
0002        /* this is a single line comment */
0003        /* this comment
```

```
0004          * has many lines
0005          */
0006         0
0007     }
```

./shux/tests/kn_multi_arg_one_name_fail.shux
```
0001     kn foo(int a, int a) int {
0002        0
0003     }
0004
0005     kn main() int {
0006        0
0007     }
```

./shux/tests/simple_vector_global_assign_fail.test
```
0001     FAIL
0002     exception
```

./shux/tests/simple_print.shux
```
0001     extern print(string s);
0002
0003     kn main() int {
0004         print("hello world!");
0005         0
0006     }
```

./shux/tests/simple_var_array.test
```
0001     COMPILE
```

./shux/tests/simple_return.test

```
0001    PASS
```

./shux/tests/let_used_main.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_multiple.shux
```
0001    ns foo = {
0002        let int x = 4;
0003    }
0004
0005    ns bar = {
0006        let scalar x = 5.0;
0007    }
0008
0009    kn main() int {
0010        0
0011    }
0012
```

./shux/tests/simple_main_in_ns_fail.shux
```
0001    ns potato = {
0002       /* if this is allowed, it is very bad form... */
0003       kn main() int {
0004          0
0005       }
0006    }
```

./shux/tests/kn_single_arg.test
```
0001    COMPILE
```

./shux/tests/simple_cond.test
```
0001    COMPILE
```

./shux/tests/struct_contains_kn_fail.shux
```
0001    struct bad {
0002       kn foo() int {
0003          0
0004       }
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/kn_return_multi_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_filter_fail.shux
```
0001    kn main() int {
0002       int[5] test = [1,2,3,4,5];
0003       int[5] test_filtered;
0004
0005       test_filtered = test :: p -> {
0006          p > 3
0007       };
0008
0009       test_filtered[0]
0010    }
```

./shux/tests/global_flat_ns_nested_redefine.shux
```
0001    ns foo = {
0002       ns bar = {
0003          let int a = 0;
```

```
0004        }
0005
0006        let int a = 1;
0007    }
0008
0009    kn main() int {
0010        0
0011    }
```

```
./shux/tests/struct_access_assign.test
0001    COMPILE
```

```
./shux/tests/simple_filter.test
0001    COMPILE
```

```
./shux/tests/struct_multi_same_name_fail_2.test
0001    FAIL
0002    exception
```

```
./shux/tests/kn_many_global.test
0001    COMPILE
```

```
./shux/tests/global_flat_ns_gn.shux
0001    ns foo = {
0002        gn foo(int x, int y) int {
0003            x*y
0004        }
0005    }
0006
0007    kn main() int {
```

```
0008        0
0009    }
```

./shux/tests/kn_vector_pass.test
```
0001    COMPILE
```

./shux/tests/simple_global_without_let_fail.shux
```
0001    int a = 5;
0002
0003    kn main() int {
0004        a
0005    }
```

./shux/tests/let_string.shux
```
0001    let string s = "potato";
0002
0003    kn main() int {
0004        0
0005    }
```

./shux/tests/simple_print_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_map.test
```
0001    COMPILE
```

./shux/tests/struct_assign_access_type_fail.shux
```
0001    struct particle {
0002       int x;
0003       int y;
0004    }
0005
0006    kn main() int {
0007       var struct particle dummy = particle { .x=1; .y=2 };
0008       dummy.x = 4.0;
0009       0
0010    }
```

./shux/tests/gn_arg_redefine.shux
```
0001    gn foo(int a) int {
0002       int a = 5;
0003       0
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/global_flat_ns_let_type_check_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_contains_gn_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_contains_kn_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/non_global_ns_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_bool_int_fail.shux
```
0001    kn main() int {
0002       var bool x = false;
0003       x = true;
0004       x = 1;
0005       0
0006    }
```

./shux/tests/global_flat_ns.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_nested_redefine.test
```
0001    COMPILE
```

./shux/tests/kn_no_return_fail_2.shux
```
0001    kn foo(int a) int {
0002       int b = 3;
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/graphics_extern_tests.test
```
0001    COMPILE
```

./shux/tests/simple_map_type_fail_3.shux
```
0001    kn main() int {
0002       int[5] a = [1,2,3,4,5];
0003       scalar b;
0004
0005       b = a @ (int v) -> {
0006          v + 1
0007       };
0008
0009       0
0010    }
```

./shux/tests/simple_array.shux
```
0001    kn main() int {
0002       int[5] x = [0,1,2,3,4];
0003       0
0004    }
```

./shux/tests/kn_multi_arg_one_name_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_single_arg.shux
```
0001    kn foo(int x) int {
0002       x+1
0003    }
0004
0005    kn main() int {
0006       int b = foo(2);
```

```
0007        b
0008    }
```

./shux/tests/gn_arg_conflict_global.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_multiple_members.test
```
0001    COMPILE
```

./shux/tests/simple_array_fail_type.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_two_ns_allowed.test
```
0001    COMPILE
```

./shux/tests/kn_vector_pass.shux
```
0001    kn foo(vector<2> a) scalar {
0002        a[0]
0003    }
0004
0005    kn main() int {
0006        0
0007    }
```

./shux/tests/simple_str_int_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/simple_array_ns_assign.shux
```
0001    ns foo = {
0002       let int[2] a = [2,4];
0003    }
0004
0005    kn main() int {
0006       int b = foo->a[0];
0007       0
0008    }
```

./shux/tests/let_bool.shux
```
0001    let bool a = false;
0002
0003    kn main() int {
0004       0
0005    }
```

./shux/tests/simple_junk_fail.shux
```
0001    kn main() int {
0002       $$#$;
0003       0
0004     }
```

./shux/tests/struct_access_assign.shux
```
0001    struct particle {
0002       int x;
0003       int y;
0004    }
0005
0006    kn main() int {
0007       var struct particle dummy = particle { .x=1; .y=2 };
```

```
0008        dummy.x = 5;
0009        0
0010    }
```

./shux/tests/simple_comment.test
```
0001    COMPILE
```

./shux/tests/simple_main_return_twice_fail.shux
```
0001    kn main() int {
0002        0
0003        0
0004    }
```

./shux/tests/simple_filter_fail_3.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_assign_immut_fail.shux
```
0001    let int a = 5;
0002
0003    kn main() int {
0004        int b = 3;
0005        a = b; /* should fail */
0006        0
0007    }
0008
```

./shux/tests/simple_int_bool_fail.shux
```
0001    kn main() int {
0002        var int x = 1;
```

```
0003        x = false;
0004        0
0005    }
```

./shux/tests/simple_junk_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_arg_redefine_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/blocks_if_else_nested.shux
```
0001    kn main() int {
0002        int a = 5;
0003        int b = if (a == 0) then
0004            5
0005        else
0006            3;
0007
0008        var int c = 5;
0009
0010        if (a == 5) then
0011            (if (b > 2) then
0012                (c = 3)
0013             else
0014                (c = -1))
0015        else
0016            (c = 42);
0017
0018        0
0019    }
```

./shux/tests/simple_print.test

```
0001    PASS
0002    hello world!
```

./shux/tests/struct_nested.shux

```
0001    struct bar {
0002       scalar y;
0003       scalar n;
0004    }
0005
0006    struct foo {
0007       int x;
0008       struct bar b;
0009    }
0010
0011    kn main() int {
0012       0
0013    }
```

./shux/tests/kn_arg_conflict_global.test

```
0001    COMPILE
```

./shux/tests/simple_main_return_invalid_fail.shux

```
0001    kn main() int {
0002       5.04
0003    }
```

./shux/tests/simple_var_array_assignment_type_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/simple_comment_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_array.test
```
0001    COMPILE
```

./shux/tests/simple_array_fail_length.shux
```
0001    kn main() int {
0002       int[2.0] a = [2,3];
0003       0
0004    }
0005
```

./shux/tests/kn_vector_arg_type_check.test
```
0001    FAIL
0002    exception
```

./shux/tests/gn_array_pass.test
```
0001    COMPILE
```

./shux/tests/gn_multi_same_name_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_array.shux

```
0001    let int[] a = [2, 5];
0002
0003    kn main() int {
0004       0
0005    }
```

./shux/tests/simple_struct.shux
```
0001    struct particle {
0002       scalar x;
0003       scalar y;
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/simple_map.shux
```
0001    kn main() int {
0002       int[5] a = [1,2,3,4,5];
0003       int[5] b;
0004
0005       var int sum = 0;
0006       a @ (int v) -> {
0007          sum = sum + v;
0008       };
0009
0010       sum
0011     }
```

./shux/tests/simple_int_str_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_contains_gn_fail.shux

```
0001    struct bad {
0002        gn foo(int x, int y) int {
0003           x*y
0004        }
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/simple_filter_type_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/simple_vector_global_assign_fail.shux

```
0001    let vector<2> g = (0.0, -9.81);
0002
0003    kn main() int {
0004       g[0] = 5.0;
0005       0
0006    }
```

./shux/tests/gn_lookback_formals_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/global_flat_ns_multiple_multiple.shux

```
0001    ns foo = {
0002       let int x = 4;
0003       let int y = 3;
0004       let scalar z = 2.0;
0005    }
0006
0007    ns bar = {
```

```
0008        let int a = 2;
0009        let int b = 3;
0010        let scalar p = 42.0;
0011        let string hi = "hello there!";
0012    }
0013
0014    ns baz = {
0015        let int qq = 5555;
0016        let scalar d = 5.0;
0017        let int x = 1;
0018        let int y = 2;
0019        let int z = 5;
0020    }
0021
0022    kn main() int {
0023        0
0024    }
0025
```

./shux/tests/struct_nested_fail.shux

```
0001    struct foo {
0002        struct bar {
0003            int x;
0004            scalar y;
0005        }
0006
0007        struct baz {
0008            struct foo {
0009                scalar x;
0010                      scalar y;
0011            }
0012
0013            string str;
0014        }
0015    }
0016
0017    kn main() int {
0018        0
0019    }
```

./shux/tests/kn_arg_name_conflict.test

```
0001    FAIL
```

```
0002    exception
```

./shux/tests/graphics_extern_tests.shux

```
0001    extern print(string s);
0002    extern graphics_init(string title, int window_w, int
window_h);
0003    extern graphics_loop(_ptr render_func, _ptr update_func);
0004    extern graphics_set_points(scalar[] buffer);
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/simple_array_assignment_fail.shux

```
0001    kn main() int {
0002       var int[2] a = [2,3];
0003       a[0] = 2.0;
0004       0
0005    }
```

./shux/tests/kn_array_pass.test

```
0001    COMPILE
```

./shux/tests/gn_no_return_fail.shux

```
0001    gn foo(int a, int b) int {
0002       int c = a + b;
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/struct_assign_access_type_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_int_scalar_fail.shux
```
0001    kn main() int {
0002       var int x = 1;
0003       x = 4.3513;
0004       0
0005    }
```

./shux/tests/gn_lookback_basic.test
```
0001    COMPILE
```

./shux/tests/simple_main_in_ns_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/global_flat_ns_member_same_name_fail.shux
```
0001    ns foo = {
0002       int a = 5;
0003       int a = 10;
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/global_flat_ns_array.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_kn_same_name.shux
```
0001    ns foo {
0002       kn bar(int a) int {
0003          0
0004       }
0005       kn bar(int b) int {
0006          0
0007       }
0008       int b = 5;
0009    }
0010
0011    kn main() int {
0012       0
0013    }
```

./shux/tests/global_flat_ns_gn.test
```
0001    COMPILE
```

./shux/tests/kn_many_one_name_global_fail.shux
```
0001    kn foo(int a) int {
0002       0
0003    }
0004
0005    kn foo(int c) int {
0006       0
0007    }
0008
0009    kn main() int {
0010       0
0011    }
```

./shux/tests/simple_vector_bad_syntax_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/gn_lookback_fail_2.shux
```
0001    let int a = 1;
0002
0003    gn foo(int x, int y) int {
0004       int b = 2;
0005       int z = x..a + y..b;
0006       z*y
0007    }
0008
0009    kn main() int {
0010       int result = do 100 foo(5, 7);
0011       0
0012    }
```

./shux/tests/kn_array_pass_no_len.shux
```
0001    kn foo(int[] a) int {
0002       a[0]
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/simple_junk_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_multi_arg.shux
```
0001    kn foo(int a, int b) int {
0002       a+b
```

```
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/simple_array_fail_length.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_multi_membs_same_name.shux
```
0001    struct particle {
0002       int x;
0003       int x;
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/struct_immutable_assign_fail.shux
```
0001    struct foo {
0002        int x;
0003    }
0004
0005    kn main() int {
0006        struct foo my_foo = foo {.x = 1};
0007        my_foo.x = 2;
0008        0
0009    }
```

./shux/tests/struct_assign.shux
```
0001    struct particle {
0002       int x;
```

```
0003       int y;
0004    }
0005    let struct particle dummy = particle { .x=1; .y=2 };
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/struct_assign_to_member_array.shux

```
0001    struct foo {
0002       int[2] a;
0003    }
0004
0005    kn main() int {
0006       var struct foo my_foo = foo { .a=[2,3] };
0007       my_foo.a = [1,2]; /* this should be allowed */
0008       0
0009    }
```

./shux/tests/global_flat_ns_multiple_multiple.test

```
0001    COMPILE
```

./shux/tests/simple_print_fail.shux

```
0001    kn main() int {
0002        int a = 5;
0003        print("hello world!", "this shouldnt work");
0004        0
0005    }
```

./shux/tests/struct_no_array_length.shux

```
0001    struct foo {
0002       int[] arr;
0003    }
0004
```

```
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/struct_two_ns_assign_type_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_many_different_sig.test
```
0001    FAIL
0002    exception
```

./shux/tests/global_flat_ns_kn_overload.shux
```
0001    ns {
0002       kn foo(int a) int {
0003          0
0004       }
0005       kn foo(int b, scalar c, string d) int {
0006          1
0007       }
0008       int a = 5;
0009    }
0010
0011    kn main() int {
0012       0
0013    }
```

./shux/tests/struct_array_member_immut.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_var_array_assignment_type_fail.shux
```
0001    kn main() int {
0002       var int[2] a = [2,3];
0003       a[0] = "potato";
0004       0
0005    }
```

./shux/tests/let_used_gn.shux
```
0001    let int a = 5;
0002
0003    gn foo(int b, int c) int {
0004       a+b+c
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/blocks_if_else.shux
```
0001    kn main() int {
0002       int a = 5;
0003       int b = if a == 0 then 5 else 3;
0004       0
0005    }
```

./shux/tests/kn_array_pass_no_len.test
```
0001    COMPILE
```

./shux/tests/simple_cond_2.shux
```
0001    kn main() int {
0002       int a = 3;
0003       int b = 5;
0004       int z = (a == b) ? 0 : 7;
```

```
0005        z
0006    }
```

```
./shux/tests/struct_access_before_init_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_string.test
0001    PASS
```

```
./shux/tests/struct_multi_same_name_two_ns_2.test
0001    COMPILE
```

```
./shux/tests/struct_multi_same_name_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/gn_basic.test
0001    COMPILE
```

```
./shux/tests/simple_scalar_str_fail.shux
0001    kn main() int {
0002        var scalar x = 1.0;
0003        x = "hi";
0004        0
0005    }
```

./shux/tests/simple_return_variable.test
```
0001    PASS
```

./shux/tests/simple_print_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_scalar_int_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_string.shux
```
0001    kn main() int {
0002        string s = "hi there this is an ASCII string\n\n";
0003        0
0004    }
```

./shux/tests/kn_vector_arg_type_check.shux
```
0001    kn foo(vector<2> a) int {
0002       a[0]
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/gn_arg_redefine.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_arg_redefine_fail.shux
```
0001    kn foo(int a, scalar b) int {
0002       int a = 5;
0003       0
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/simple_vector_global_access.shux
```
0001    let vector<2> g = (0.0, -9.81);
0002
0003    kn main() int {
0004       scalar g_val = g[1];
0005       0
0006    }
```

./shux/tests/struct_multi_same_name_fail.shux
```
0001    struct particle {
0002       scalar x;
0003       scalar y;
0004    }
0005
0006    struct particle {
0007       scalar x;
0008       scalar y;
0009    }
0010
0011    kn main() int {
0012       0
0013    }
```

./shux/tests/struct_assign_type_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_bool_str_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_vector_non_scalar_fail.shux
```
0001    kn main() int {
0002       vector<2> g = (1, 2);
0003       0
0004    }
```

./shux/tests/let_scalar.test
```
0001    COMPILE
```

./shux/tests/simple_str_bool_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_no_main.shux
```
0001    let int a = 5;
```

./shux/tests/global_flat_ns_array.shux
```
0001    ns foo = {
0002       let string[4] bar = ["b", "c", "e", "d"];
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/kn_arg_redefine_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_scalar_int_fail.shux
```
0001    kn main() int {
0002       var scalar x = 1.0;
0003       x = 1;
0004       0
0005    }
```

./shux/tests/struct_multi_same_name_two_ns.shux
```
0001    ns foo = {
0002       struct p {
0003          scalar x;
0004          scalar y;
0005       }
0006    }
0007
0008    ns bar = {
0009       struct p {
0010          int x;
0011          int y;
0012       }
0013    }
0014
0015    kn main() int {
```

```
0016        0
0017    }
```

./shux/tests/struct_multi_same_name_two_ns_2.shux
```
0001    ns foo = {
0002        struct p {
0003            int x;
0004            int y;
0005        }
0006    }
0007
0008    ns bar = {
0009        struct p {
0010            int x;
0011            int y;
0012        }
0013    }
0014
0015    kn main() int {
0016        0
0017    }
```

./shux/tests/global_flat_ns_only_kn.test
```
0001    COMPILE
```

./shux/tests/kn_multi_arg_one_name_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_used_main.shux
```
0001    let int a = 5;
0002
0003    kn main() int {
0004        a
```

```
0005    }
```

./shux/tests/simple_vector_infer_size.shux

```
0001    kn main() int {
0002       vector<> g = (0.0, -9.81);
0003       0
0004    }
```

./shux/tests/simple_array_access.shux

```
0001    kn main() int {
0002       int[5] x = [0,1,2,3,4];
0003       x[2]
0004    }
```

./shux/tests/kn_no_return_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/struct_multi_same_name_fail_2.shux

```
0001    struct particle {
0002       scalar x;
0003       scalar y;
0004    }
0005
0006    struct particle {
0007       int x;
0008       string y;
0009    }
0010
0011    kn main() int {
0012       0
0013    }
```

./shux/tests/simple_var_array_assignment.shux
```
0001    kn main() int {
0002       var int[2] a = [2,3];
0003       a[0] = 3;
0004       0
0005    }
```

./shux/tests/simple_map_type_fail_3.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_array_2.test
```
0001    COMPILE
```

./shux/tests/gn_vector_pass.shux
```
0001    gn foo(vector<2> a, vector<2> b) scalar {
0002       a[0] + b[0]
0003    }
0004
0005    kn main() int {
0006       0
0007    }
```

./shux/tests/simple_array_fail.shux
```
0001    kn main() int {
0002       int[2] x = [2,3];
0003       x[0] = 5; /* assignment not allowed, not var */
0004       0
0005    }
```

./shux/tests/struct_immutable_assign_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_print_fail_2.shux
```
0001    kn main() int {
0002       int a = 2;
0003       print("hello world!", a);
0004       0
0005    }
```

./shux/tests/simple_vector_global_access.test
```
0001    COMPILE
```

./shux/tests/struct_multi_membs_same_name.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_struct.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_kn_overload.test
```
0001    FAIL
0002    exception
```

./shux/tests/let_used_kn.test
```
0001    COMPILE
```

./shux/tests/kn_lookback_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/gn_lookback_fail.shux
```
0001    gn foo(int x, int y) int {
0002       int a = x..1.0 + y..2.0;
0003       a*y
0004    }
0005
0006    kn main() int {
0007       int result = do 100 foo(5, 7);
0008       0
0009    }
```

./shux/tests/struct_nested.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_multiple.test
```
0001    COMPILE
```

./shux/tests/let_assign_type_fail_2.shux
```
0001    let scalar p = 24.5445;
0002
0003    kn main() int {
```

```
0004         var string s = "potato";
0005         s = p;
0006         0
0007     }
```

./shux/tests/let_bool.test
```
0001     COMPILE
```

./shux/tests/struct_cannot_init_in_definition.shux
```
0001     struct foo {
0002        int a;
0003        int b = 5;
0004     }
0005
0006     kn main() int {
0007        0
0008     }
```

./shux/tests/gn_bad_syntax_define_fail.shux
```
0001     gn test(int x) {
0002        0
0003     }
0004
0005     kn main() int {
0006        0
0007     }
```

./shux/tests/kn_array_pass.shux
```
0001     kn foo(int[2] a) int {
0002        a[0]
0003     }
0004
0005     kn main() int {
0006        0
```

```
0007    }
```

./shux/tests/simple_filter_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_return.shux
```
0001    kn main() int {
0002        0
0003    }
```

./shux/tests/blocks_if.test
```
0001    COMPILE
```

./shux/tests/global_flat_ns_member_same_name_fail_2.shux
```
0001    ns foo = {
0002       int a = 5;
0003       scalar a = 2.0;
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/struct_two_ns_assign_type_fail.shux
```
0001    ns foo = {
0002       struct p {
0003          int x;
0004          int y;
0005       }
```

```
0006    }
0007
0008    ns bar = {
0009       struct p {
0010          int x;
0011          int y;
0012       }
0013    }
0014
0015    kn main() int {
0016       var struct foo->p foo_p = struct foo->p { .x=0, .y=0 };
0017       var struct bar->p bar_p = struct bar->p { .x=0, .y=0 };
0018       foo_p = bar_p; /* this line should fail */
0019       0
0020    }
```

./shux/tests/gn_arg_name_conflict.test

```
0001    FAIL
0002    exception
```

./shux/tests/struct_access_before_init_fail.shux

```
0001    struct particle {
0002       int x;
0003       int y;
0004    }
0005
0006    kn main() int {
0007       struct particle dummy;
0008       int y = dummy.x;
0009       0
0010    }
```

./shux/tests/gn_local_immut_fail.shux

```
0001    gn test(int x) int {
0002        int a = x;
0003        a = 2;
0004        a
0005    }
```

```
0006
0007    kn main() int {
0008       0
0009    }
0010
```

./shux/tests/gn_arg_conflict_global.shux

```
0001    let int a = 5;
0002
0003    gn foo(int a, int b) int {
0004       b
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/global_flat_ns_advanced_types.shux

```
0001    ns foo = {
0002       let int x = 1;
0003       let scalar y = 2.0;
0004       let string z = "zzz";
0005       kn bar(int x) int {
0006          x+1
0007       }
0008    }
0009
0010    kn main() int {
0011       0
0012    }
```

./shux/tests/simple_array_assignment_fail.test

```
0001    FAIL
0002    exception
```

./shux/tests/simple_str_bool_fail.shux
```
0001    kn main() int {
0002       var string x = "hi";
0003       x = true;
0004       0
0005    }
```

./shux/tests/struct_no_array_length.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_scalar_str_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/global_flat_ns.shux
```
0001    ns foo = {
0002        let int bar  = 4;
0003    }
0004
0005    kn main() int {
0006        0
0007    }
0008
```

./shux/tests/simple_filter_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/struct_assign_access_to_type_fail.shux
```
0001    struct particle {
0002       int x;
0003       int y;
0004    }
0005
0006    kn main() int {
0007       struct particle dummy = particle { .x=1; .y=2 };
0008       string p = dummy.x;
0009       0
0010    }
```

./shux/tests/simple_filter.shux
```
0001    kn main() int {
0002       int[5] test = [1,2,3,4,5];
0003       int[5] test_filtered;
0004
0005       test_filtered = test :: (int p) -> {
0006          p > 3
0007       };
0008
0009       test_filtered[0]
0010    }
```

./shux/tests/blocks_if_else_nested.test
```
0001    COMPILE
```

./shux/tests/simple_filter_fail_2.shux
```
0001    kn main() int {
0002       int[5] test = [1,2,3,4,5];
0003       int[5] test_filtered;
0004
0005       test_filtered = test :: (int p) -> {
0006          p = p + 1;
0007       };
0008
0009       test_filtered[0]
0010    }
```

./shux/tests/let_assign_type_fail.shux
```
0001    let int a = 5;
0002
0003    kn main() int {
0004       int b = a;
0005       scalar c = a; /* should fail */
0006       0
0007    }
```

./shux/tests/simple_array_ns_assign_immut_fail.shux
```
0001    ns foo = {
0002       int[] a = [2,4];
0003    }
0004
0005    kn main() int {
0006       int b = foo->a[0];
0007       foo->a[1] = 5; /* should fail */
0008       0
0009    }
```

./shux/tests/simple_vector.shux
```
0001    kn main() int {
0002       vector<2> g = (0.0, -9.81);
0003       0
0004    }
```

./shux/tests/global_flat_ns_complex_test_1.test
```
0001    COMPILE
```

./shux/tests/gn_lookback_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/blocks_if.shux
```
0001    kn print(string a) {
0002    }
0003
0004    kn main() int {
0005       int a = 5;
0006       if (a > 3) then
0007                                print("bla")
0008       else
0009          noop;
0010       0
0011    }
0012
```

./shux/tests/run_tests.sh
```
0001    #!/bin/sh
0002
0003    compiler="./shuxc"
0004    runtime=lli
0005    test_ext=".shux"
0006    obj_ext=".ll"
0007    out_ext=".out"
0008    expected_ext=".test"
0009    passes=0
0010    warnings=0
0011    fails=0
0012    skip_pass=0
0013
0014    if [ $# -eq 1 ]; then
0015        if [ "$1" = "COMPILE" ]; then
0016            echo "running only COMPILE/FAIL tests\n"
0017            skip_pass=1
0018            test_names=`ls *.shux`
0019        else
0020            echo "running single test\n"
0021            test_names=$1
0022            # this should be extended to just run the below with
test_names=our one file
```

```
0023        fi
0024    else
0025        echo "running all tests sequentially\n"
0026        test_names=`ls *.shux`
0027    fi
0028
0029    for test in $test_names
0030    do
0031            test=`echo "$test" | cut -d'.' -f1`
0032            echo -n $test
0033
0034        # what intermediate objects do we need?
0035            test_obj=$test$obj_ext
0036            test_out=$test$out_ext
0037            expected_out=$test$expected_ext
0038        expected_head=$(head -n 1 $expected_out)
0039            expected_body=/tmp/expected_body
0040        echo -n " ($expected_head)... "
0041
0042        # run compiler
0043        ERROR="$($compiler $test$test_ext 2>&1 > $test_obj)"
0044        if [ -n "$ERROR" ]; then
0045            echo $ERROR > $test_out
0046
0047            # if echo $ERROR | grep -q "Uncaught exception"; then
0048            #     if [ "$expected_head" = "PASS" ]; then
0049            #         echo "passed! ✅"
0050            #         passes=$((passes+1))
0051            #         continue
0052            #     else
0053            #         echo "compilation failed! saving $test_out
❌"
0054            #         fails=$((fails+1))
0055            #         continue
0056            #     fi
0057            # fi
0058        fi
0059
0060        # check for valid test file
0061        if [ ! -f $expected_out ]; then
0062            # expected output file not found
0063            echo "expected output file not found! ❌"
0064            fails=$((fails+1))
0065            continue
0066        fi
0067
0068        # check type of test and invoke runtime if necessary
0069        echo -n $(tail -n +2 $expected_out) > $expected_body
0070        if [ "$expected_head" = "PASS" ]; then
```

```
0071            if [ $skip_pass -eq 1 ]; then
0072                echo "skipping."
0073            else
0074                # diff against expected output
0075                $runtime $test_obj > $test_out 2>&1
0076                if diff -q $test_out $expected_body > /dev/null;
then
0077                    echo "passed! ✅"
0078                    passes=$((passes+1))
0079                else
0080                    echo "failed! please check $test$out_ext for
debug info ❌"
0081                    fails=$((fails+1))
0082                fi
0083            fi
0084        elif [ "$expected_head" = "FAIL" ]; then
0085            # pattern match against execption keywords
0086            exp_holder=$(cat $expected_body)
0087            if grep -q "$exp_holder" $test_out > /dev/null 2>&1;
then
0088                echo "passed! ✅"
0089                passes=$((passes+1))
0090            elif grep -q "WARN" $test_out > /dev/null 2>&1; then
0091                echo "compilation threw warning 🔶"
0092                warnings=$((warnings+1))
0093            else
0094                echo "failed! expected error keyword(s) not found.
please see $test$out_ext ❌"
0095                fails=$((fails+1))
0096            fi
0097        elif [ "$expected_head" = "COMPILE" ]; then
0098            if [ -f $test_out ]; then
0099                if grep -q "Uncaught exception" $test_out > /dev/
null; then
0100                    echo "compilation failed ❌"
0101                    fails=$((fails+1))
0102                elif grep -q "WARN" $test_out > /dev/null 2>&1;
then
0103                    echo "compilation threw warning 🔶"
0104                    warnings=$((warnings+1))
0105                else
0106                    echo "passed! ✅"
0107                    passes=$((passes+1))
0108                fi
0109            else
0110                echo "passed! ✅"
0111                passes=$((passes+1))
```

```
0112            fi
0113        else
0114            echo "invalid test specification ❌"
0115            fails=$((fails+1))
0116            continue
0117        fi
0118    done
0119
0120    echo "\n$((fails+passes+warnings)) tests run", "$fails
fail(s)", "$warnings warning(s)", "$passes passes"
0121    if [ $((fails+warnings)) -eq 0 ]; then
0122        echo "all tests passed. you done it 💯"
0123    fi
```

./shux/tests/kn_no_return_fail.shux

```
0001    kn foo(int a) int {
0002    }
0003
0004    kn main() int {
0005       0
0006    }
```

./shux/tests/gn_call_for.shux

```
0001    let int x = 5;
0002
0003    gn foo() int {
0004       x
0005    }
0006
0007    kn bar(int x) int {
0008       0
0009    }
0010
0011    kn main() int {
0012       for 5 foo() @ bar;
0013       0
0014    }
```

./shux/tests/simple_vector_global.test
```
0001    COMPILE
```

./shux/tests/simple_vector_no_value.shux
```
0001    kn main() int {
0002       var vector<2> g = (0.0, 0.0);
0003       g[0] = 2.0;
0004       0
0005    }
```

./shux/tests/simple_comment_fail.shux
```
0001    kn main() int {
0002       /* this is a comment that is never closed
0003       0
0004    }
```

./shux/tests/simple_multi_main_fail.shux
```
0001    kn main() int {
0002       0
0003    }
0004
0005    kn main() int {
0006       2
0007    }
```

./shux/tests/graphics_valid_call_test.test
```
0001    COMPILE
```

./shux/tests/simple_scalar_bool_fail.test
```
0001    FAIL
```

```
0002    exception
```

./shux/tests/global_flat_ns_complex_test_1.shux

```
0001    ns foo = {
0002            ns nested = {
0003                    let int x = 2;
0004                    kn baz(int y, string s) int {
0005                            y+2
0006                    }
0007            }
0008            let int x = 1;
0009            let scalar y = 5.0;
0010            kn bar(int x) int {
0011                    x+1
0012            }
0013    }
0014
0015    kn main() int {
0016            0
0017    }
```

./shux/tests/simple_bool_scalar_fail.shux

```
0001    kn main() int {
0002       var bool x = true;
0003       x = 23.53452;
0004       0
0005    }
```

./shux/tests/kn_arg_conflict_global.shux

```
0001    let int a = 5;
0002
0003    kn foo(int a) int {
0004       a
0005    }
0006
0007    kn main() int {
0008       0
0009    }
```

./shux/tests/gn_multi_same_name_fail.shux
```
0001    gn foo(int a) int {
0002       0
0003    }
0004
0005    gn foo(scalar b, scalar c) int {
0006       0
0007    }
0008
0009    kn main() int {
0010       0
0011    }
```

./shux/tests/let_assign_type_fail_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/gn_lookback_formals_fail.shux
```
0001    gn foo(int x, int y) int {
0002       int a = x..1 + y..2;
0003       a*y
0004    }
0005
0006    kn main() int {
0007       int result = do 100 foo(5, 7);
0008       0
0009    }
```

./shux/tests/simple_var_array.shux
```
0001    kn main() int {
0002       var int[2] a = [2,3];
0003       0
0004    }
```

./shux/tests/struct_multi_membs_same_name_2.test
```
0001    FAIL
0002    exception
```

./shux/tests/kn_arg_name_conflict.shux
```
0001    kn foo(int a) int {
0002       int a = 3;
0003       0
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/kn_many_global.shux
```
0001    kn foo(int a) int {
0002       0
0003    }
0004
0005    kn bar(int a) int {
0006       1
0007    }
0008
0009    kn main() int {
0010       int a = foo(1);
0011       int b = foo(2);
0012       a+b
0013    }
```

./shux/tests/gn_call_for.test
```
0001    COMPILE
```

./shux/tests/struct_cannot_init_in_definition.test
```
0001    FAIL
0002    exception
```

./shux/tests/global_flat_ns_only_kn.shux
```
0001    ns foo = {
0002       kn bar(int a) int {
0003          0
0004       }
0005
0006       kn baz(scalar b) scalar {
0007          24.545
0008       }
0009    }
0010
0011    kn main() int {
0012       0
0013    }
```

./shux/tests/gn_lookback_basic.shux
```
0001    gn foo(int a) int {
0002       int x = x..1 : -1;
0003       x
0004    }
0005
0006    kn main() int {
0007       0
0008    }
```

./shux/tests/kn_multi_arg_one_name_fail_2.shux
```
0001    kn foo(int a, string a) int {
0002       0
0003    }
0004
0005    kn main() int {
```

```
0006        0
0007    }
```

./shux/tests/gn_basic.shux
```
0001    gn foo(int a, int b) int {
0002        a + b
0003    }
0004
0005    kn main() int {
0006        0
0007    }
```

./shux/tests/kn_multi_arg.test
```
0001    COMPILE
```

./shux/tests/simple_array_inferred_length.shux
```
0001    kn main() int {
0002        int[2] a = [2,3];
0003        a[1]
0004    }
0005
```

./shux/tests/simple_filter_type_fail.shux
```
0001    kn main() int {
0002        int[5] test = [1,2,3,4,5];
0003        int[5] test_filtered;
0004
0005        test_filtered = test :: (scalar p) -> {
0006            p > 3
0007        };
0008
0009        test_filtered[0]
0010    }
```

```
./shux/tests/global_flat_ns_kn_same_name.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_scalar_bool_fail.shux
0001    kn main() int {
0002       var scalar x = 1.0;
0003       x = false;
0004       0
0005    }
```

```
./shux/tests/let_assign_immut_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/gn_no_return_fail.test
0001    FAIL
0002    exception
```

```
./shux/tests/simple_map_type_fail_2.test
0001    FAIL
0002    exception
```

```
./shux/tests/struct_assign_to_member_array.test
0001    COMPILE
```

./shux/tests/gn_call_for_bad.shux
```
0001    gn foo() int {
0002       int x = x..1 + 1 : 0;
0003       x
0004    }
0005
0006    kn bar(int x) int {
0007       0
0008    }
0009
0010    kn main() int {
0011       for 5 foo(); /* needs @ and function */
0012       0
0013    }
```

./shux/tests/simple_vector_infer_size.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_filter_fail_3.shux
```
0001    kn main() int {
0002       int[5] test = [1,2,3,4,5];
0003       int[5] test_filtered;
0004
0005       test_filtered = test @ (int p) -> {
0006          p > 3
0007       };
0008
0009       test_filtered[0]
0010    }
```

./shux/tests/simple_map_syntax_fail.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_map_type_fail_2.shux
```
0001    kn main() int {
0002       int[5] a = [1,2,3,4,5];
0003       scalar[5] b;
0004
0005       b = a @ (int v) -> {
0006          v + 1
0007       };
0008
0009       0
0010    }
```

./shux/tests/struct_multi_same_name_two_ns.test
```
0001    COMPILE
```

./shux/tests/simple_scalar_literals.test
```
0001    FAIL
0002    exception
```

./shux/tests/simple_scalar.test
```
0001    PASS
```

./shux/tests/let_string.test
```
0001    COMPILE
```

./shux/tests/simple_str_int_fail.shux
```
0001    kn main() int {
0002       var string x = "hi";
0003       x = 1;
0004       0
0005    }
```

./shux/tests/struct_two_ns_allowed.shux
```
0001    ns foo = {
0002       struct p {
0003          int x;
0004          int y;
0005       }
0006    }
0007
0008    ns bar = {
0009       struct p {
0010          int x;
0011          int y;
0012       }
0013    }
0014
0015    kn main() int {
0016       struct foo->p foo_p = foo->p { .x=0; .y=0 };
0017       struct bar->p bar_p = bar->p { .x=0; .y=0 };
0018       0
0019    }
```

./shux/tests/simple_junk_fail_2.shux
```
0001    let 1$potato;
0002
0003    kn main() int {
0004       0
0005    }
```

./shux/tests/blocks_if_else_2.test
```
0001    COMPILE
```

./shux/tests/simple_str_scalar_fail.test
```
0001    FAIL
0002    exception
```

./shux/examples/euler.shux
```
0001    let scalar dt = 0.01;
0002    let scalar g = -9.81;
0003
0004    /* 1D particle data structure with pos and vel */
0005    struct particle {
0006       scalar x;
0007       scalar v;
0008    }
0009
0010    gn euler(struct particle[1000] init) struct particle[1000] {
0011          struct particle[1000] pbuf = (pbuf..1 : init) @ (struct
particle p) -> {
0012          particle {
0013             .v = p.v + g * dt;
0014                    .x = p.x + p.v * dt
0015          }
0016       };
0017             pbuf
0018    }
0019
0020    kn main() int {
0021      var struct particle[1] p_init = [ particle
{ .v=1.0; .x=0.0}];
0022      struct particle[1000] euler = do 1000 euler(p_init);
0023      euler @ (struct particle state) -> {
0024      /* opengl bindings */
0025                                state
0026         };
0027       0
0028    }
0029
```

./shux/examples/hello_world.shux

```
0001    extern print(string s);
0002
0003    kn main() int {
0004            print("hello world!");
0005            0
0006    }
```

./shux/examples/fib.shux
```
0001    gn fib() int {
0002            int y = (y..1 : 1) + (y..2 : 1);
0003            y
0004    }
0005
0006    kn main() int {
0007            int fib5 = do 5 fib();
0008            0
0009    }
```

./shux/src/exceptions.mli
```
0001    open Ast
0002
0003    type id_err = string
0004    exception BinopTypErr of ((* left *) typ * bin_op * (* right
*) typ)
0005    exception UnopTypErr of (un_op * typ)
0006    exception AccessErr of ((* struct *) id_err * (* member *) typ
*  (* member *) id_err)
0007    exception CondTypErr of ((* if *) typ * (* then *) typ * (*
else *) typ)
0008    exception UndeclaredId of id_err
0009    exception NameConflict of id_err
0010    exception GnCallErr of id_err
0011    exception FnCallTypErr of ((* fn *) id_err * (* return *) typ
* (* expected *) typ)
0012    exception FnArgArr of ((* fn *) id_err * (* actual *) typ * (*
formal *) typ)
0013    exception ForbiddenBindTypErr of (id_err * typ)
```

./shux/src/shuxc.ml
```
0001    (* shux frontend entry point -- very basic *)
0002    open Scanner
0003    open Parser
0004    open Semant
0005    open Llvm
0006    open Printf
0007    open Astprint
0008    open Ast_sast
0009    open Sast_cast
0010    open Cast_llast
0011    open Lltranslate
0012
0013    type action = Ast | LLVM
0014
0015    let _ =
0016            let (cin, action) =
0017        if Array.length Sys.argv = 1 then
0018           raise (Failure "Argument spec: ./shuxc [-a / -l]
[program name]")
0019           else if Array.length Sys.argv = 3 then
0020                           let a = match Sys.argv.(1) with
0021                                   | ("-a") -> Ast
0022                                   | ("-l") -> LLVM
0023                                   | _ -> LLVM in
0024                           let c = open_in Sys.argv.(2) in
0025                           (c, a)
0026                   else (open_in Sys.argv.(1), LLVM) in
0027             let lexbuf = Lexing.from_channel cin in
0028             let ast = Parser.program Scanner.token lexbuf in
0029             let check_ast = Semant.check ast in
0030      let sast = Ast_sast.translate_to_sast check_ast in
0031      let cast = Sast_cast.sast_to_cast sast in
0032      let llast = Cast_llast.cast_to_llast cast in
0033      let llvm = Lltranslate.translate llast in
0034            match action with
0035                   | Ast -> print_string
(Astprint.string_of_program ast)
0036                   | LLVM ->
0037                   print_string (Llvm.string_of_llmodule llvm)
```

./shux/src/backend/lltranslate.ml
```
0001    module L = Llvm
0002    open Llast
0003
0004    module StringMap = Map.Make(String)
```

```
0005
0006    let translate (structs,globals,funcs) =
0007      (* debug use *)
0008      (*
0009      let print_llvalue llvalue msg=
0010        let lltype = L.type_of llvalue in
0011        prerr_string ("["^msg^"]"^" lltype:"^(L.string_of_lltype
lltype)^";\n")
0012      in
0013       *)
0014
0015      let the_context = L.global_context () in
0016      let the_module = L.create_module the_context "shux" and
0017          i32_t = L.i32_type the_context and
0018          i8_t = L.i8_type the_context and
0019          i1_t = L.i1_type the_context and
0020          double_t = L.double_type the_context and
0021          double_precision = 16 and (* adjust this to change
number of digits printf will print *)
0022          void_t = L.void_type the_context
0023      in
0024      let str_t = L.pointer_type i8_t in
0025
0026      (* BEGIN EXTERNAL CALLS DEFINITIONS
0027       *
0028       *)
0029
0030      (* LVS *)
0031      (* gl init call set *)
0032      (*
0033      let glClearColor_formals = Array.of_list [float_t; float_t;
float_t; float_t] in
0034      let glClearColor_sign = L.function_type void_t
glClearColor_formals in
0035      let glClearColor_func = L.declare_function "glClearColor"
glClearColor_sign the_module in
0036
0037      let glEnable_formals = Array.of_list [i32_t] in
0038      let glEnable_sign = L.function_type void_t glEnable_formals
in
0039      let glEnable_func = L.declare_function "glEnable"
glEnable_sign the_module in
0040
0041      let glPointSize_formals = Array.of_list [float_t] in
0042      let glPointSize_sign = L.function_type void_t
glPointSize_formals in
0043      let glPointSize_func = L.declare_function "glPointSize"
glPointSize_sign the_module in
0044
0045      let glMatrixMode_formals = Array.of_list [i32_t] in
```

```
0046      let glMatrixMode_sign = L.function_type void_t
glMatrixMode_formals in
0047      let glMatrixMode_func = L.declare_function "glMatrixMode"
glMatrixMode_sign the_module in
0048
0049
0050      (* gl render call set *)
0051      let glClear_formals = Array.of_list [i32_t] in
0052      let glClear_sign = L.function_type void_t glClear_formals in
0053      let glClear_func = L.declare_function "glClear" glClear_sign
the_module in
0054
0055      let glLoadIdentity_formals = Array.of_list [] in
0056      let glLoadIdentity_sign = L.function_type void_t
glLoadIdentity_formals in
0057      let glLoadIdentity_func = L.declare_function
"glLoadIdentity" glLoadIdentity_sign  the_module in
0058
0059      let glOrtho_formals = Array.of_list [float_t; float_t;
float_t; float_t] in
0060      let glOrtho_sign = L.function_type void_t glOrtho_formals in
0061      let glOrtho_func = L.declare_function "glOrtho" glOrtho_sign
the_module in
0062
0063      let glColor4f_formals = Array.of_list [float_t; float_t;
float_t; float_t] in
0064      let glColor4f_sign = L.function_type void_t
glColor4f_formals in
0065      let glColor4f_func = L.declare_function "glColor4f"
glColor4f_sign the_module in
0066
0067      (* for the render loop *)
0068      let glBegin_formals = Array.of_list [i32_t] in
0069      let glBegin_sign = L.function_type void_t glBegin_formals in
0070      let glBegin_func = L.declare_function "glBegin" glBegin_sign
the_module in
0071
0072      let glVertex2f_formals = Array.of_list [float_t; float_t] in
0073      let glVertex2f_sign = L.function_type void_t
glVertex2f_formals in
0074      let glVertex2f_func = L.declare_function "glVertex2f"
glVertex2f_sign the_module in
0075
0076      let glVertexArray_formals = Array.of_list [i32_t; i32_t;
i32_t; L.pointer_type float_t ] in
0077      let glVertexArray_sign = L.function_type void_t
glVertexArray_formals in
0078      let glVertexArray_func = L.declare_function "glVertexArray"
glVertexArray_sign the_module in
0079
```

```
0080      let glEnd_formals = Array.of_list [i32_t] in
0081      let glEnd_sign = L.function_type void_t glEnd_formals in
0082      let glEnd_func = L.declare_function "glEnd" glEnd_sign
the_module in
0083
0084      let glutSwapBuffers_formals = Array.of_list [] in
0085      let glutSwapBuffers_sign = L.function_type void_t
glutSwapBuffers_formals in
0086      let glutSwapBuffers_func = L.declare_function
"glutSwapBuffers" glutSwapBuffers_sign the_module in
0087
0088
0089      (* gl update/idle call set *)
0090      let glutPostRedisplay_formals = Array.of_list [] in
0091      let glutPostRedisplay_sign = L.function_type void_t
glutPostRedisplay_formals in
0092      let glutPostRedisplay_func = L.declare_function
"glutPostRedisplay" glutPostRedisplay_sign the_module in
0093
0094
0095      (* glut main call set *)
0096      let glutInitWindowSize_formals = Array.of_list [i32_t;
i32_t] in
0097      let glutInitWindowSize_sign = L.function_type void_t
glutInitWindowSize_formals in
0098      let glutInitWindowSize_func = L.declare_function
"glutInitWindowSize" glutInitWindowSize_sign the_module in
0099
0100      let glutInit_formals = Array.of_list [L.pointer_type i32_t;
L.pointer_type (L.pointer_type i8_t)] in
0101      let glutInit_sign = L.function_type void_t glutInit_formals
in
0102      let glutInit_func = L.declare_function "glutInit"
glutInit_sign the_module in
0103
0104      let glutCreateWindow_formals = Array.of_list [L.pointer_type
i8_t] in
0105      let glutCreateWindow_sign = L.function_type i32_t
glutCreateWindow_formals in
0106      let glutCreateWindow_func = L.declare_function
"glutCreateWindow" glutCreateWindow_sign the_module in
0107
0108      let glutDisplayFunc_formals = Array.of_list [fptr_void_t] in
0109      let glutDisplayFunc_sign = L.function_type void_t
glutDisplayFunc_formals in
0110      let glutDisplayFunc_func = L.declare_function
"glutDisplayFunc" glutDisplayFunc_sign the_module in
0111
0112      let glutIdleFunc_formals = Array.of_list [fptr_void_t] in
0113      let glutIdleFunc_sign = L.function_type void_t
```

```
glutIdleFunc_formals in
0114      let glutIdleFunc_func = L.declare_function "glutIdleFunc"
glutIdleFunc_sign the_module in
0115
0116      let glutMainLoop_formals = Array.of_list [] in
0117      let glutMainLoop_sign = L.function_type void_t
glutMainLoop_formals in
0118      let glutMainLoop_func = L.declare_function "glutMainLoop"
glutMainLoop_sign the_module in
0119
0120      (* END LVS *)
0121      (* END EXTERNAL CALLS DEFINITIONS
0122       *
0123       *)
0124
0125      (*
************************************************************** *)
0126
0127      (* LVS START initgl modifications *)
0128      let init_gl_func_formals = Array.of_list [] in
0129      let init_gl_func_sign = L.function_type void_t
init_gl_func_formals in
0130      let init_gl_func_llvalue = L.define_function "__init_gl"
init_gl_func_sign the_module in
0131      let builder_init_gl_func = L.builder_at_end the_context
(L.entry_block init_gl_func_llvalue) in
0132      let _ = L.build_call glClearColor_func [| L.const_float
float_t 0.9; L.const_float float_t 0.9; L.const_float float_t 0.9;
L.const_float float_t 1.0 |] "" builder_init_gl_func in
0133      let _ = L.build_call glEnable_func [| L.const_int i32_t 2832
|] "" builder_init_gl_func in
0134      let _ = L.build_call glPointSize_func [| L.const_float
float_t 10.0 |] "" builder_init_gl_func in
0135      let _ = L.build_call glMatrixMode_func [| L.const_int i32_t
5889 |] "" builder_init_gl_func in
0136      let _ = L.build_ret_void builder_init_gl_func in
0137      (* LVS END initgl modifications *)
0138
0139
0140      (*
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&& *)
0141      (* begin work on actual bindings *)
0142
0143      (* graphics_do_update *)
0144      (* user-callable update function *)
0145      let idle_func_formals = Array.of_list [] in
0146      let idle_func_sign = L.function_type void_t
idle_func_formals in
0147      let idle_func_llvalue = L.define_function
"graphics_do_update" idle_func_sign the_module in
```

```
0148      let builder_idle_func = L.builder_at_end the_context
(L.entry_block idle_func_llvalue) in
0149      let _ = L.build_call glutPostRedisplay_func [||] ""
builder_idle_func in
0150      let _ = L.build_ret_void builder_idle_func in
0151
0152      (* graphics_do_render *)
0153      (* a display function rendering a list of 2d scalars *)
0154      (* LVSTOOD doesn't take parameters yet *)
0155      let display_func_formals = Array.of_list [] in
0156      let display_func_sign = L.function_type void_t
display_func_formals in
0157      let display_func_llvalue = L.define_function
"graphics_do_render" display_func_sign the_module in
0158      let builder_display_func = L.builder_at_end the_context
(L.entry_block display_func_llvalue) in
0159      let _ = L.build_call glClear_func [|L.const_int i32_t
16640|] "" builder_display_func in
0160      let _ = L.build_call glLoadIdentity_func [||] ""
builder_display_func in
0161      let _ = L.build_call glOrtho_func [| L.const_float float_t
0.0; L.const_float float_t 800.0; L.const_float float_t 0.0;
L.const_float float_t 600.0;|] "" builder_display_func in
0162      (* user code here -- testing *)
0163      let _ = L.build_call glColor4f_func [| L.const_float float_t
0.2; L.const_float float_t 0.6; L.const_float float_t 1.0;
L.const_float float_t 1.0; |] "" builder_display_func in
0164      let _ = L.build_call glBegin_func [| L.const_int i32_t 0 |]
"" builder_display_func in
0165      let _ = L.build_call glVertex2f_func [| L.const_float
float_t 0.0; L.const_float float_t 0.0 |] "" builder_display_func in
0166      let _ = L.build_call glEnd_func [| L.const_int i32_t 0 |] ""
builder_display_func in
0167      (* end user code *)
0168      let _ = L.build_call glutSwapBuffers_func [||] ""
builder_display_func in
0169      let _ = L.build_ret_void builder_display_func in
0170
0171      (* graphics init *)
0172      (* spoof argc for glutinit *)
0173      let graphics_init_formals = Array.of_list [] in
0174      let graphics_init_sign = L.function_type void_t
graphics_init_formals in
0175      let graphics_init_llvalue = L.define_function
"graphics_init" graphics_init_sign the_module in
0176      let builder_graphics_init = L.builder_at_end the_context
(L.entry_block graphics_init_llvalue) in
0177      let argc_ptr = L.build_alloca i32_t "argc"
builder_graphics_init in
0178      let _ = L.build_store (L.const_int i32_t 0) argc_ptr
```

```
builder_graphics_init in
0179      let _ = L.build_call glutInitWindowSize_func [| (L.const_int
i32_t 800); (L.const_int i32_t 600) |] "" builder_graphics_init in
0180      let _ = L.build_call glutInit_func [| argc_ptr;
(L.const_pointer_null (L.pointer_type (L.pointer_type i8_t))) |] ""
builder_graphics_init in
0181      let _ = L.build_call glutCreateWindow_func [|
(L.const_pointer_null (L.pointer_type i8_t)) |] "dummy"
builder_graphics_init in
0182      let _ = L.build_ret_void builder_graphics_init in
0183
0184      (* graphics_loop *)
0185      (* LVSTODO doesnt take parameters yet *)
0186      let graphics_loop_formals = Array.of_list [] in
0187      let graphics_loop_sign = L.function_type void_t
graphics_loop_formals in
0188      let graphics_loop_llvalue = L.define_function
"graphics_loop" graphics_loop_sign the_module in
0189      let builder_graphics_loop = L.builder_at_end the_context
(L.entry_block graphics_loop_llvalue) in
0190      let _ = L.build_call glutDisplayFunc_func [|
display_func_llvalue |] "" builder_graphics_loop in
0191      let _ = L.build_call glutIdleFunc_func [| idle_func_llvalue
|] "" builder_graphics_loop in
0192      let _ = L.build_call init_gl_func_llvalue [||] ""
builder_graphics_loop in
0193      let _ = L.build_call glutMainLoop_func [||] ""
builder_graphics_loop in
0194      let _ = L.build_ret_void builder_graphics_loop in
0195
0196      (*
********************************************************************** *)
0197       *)
0198
0199      let extract_type = function
0200          LLRegLabel (typ, str) -> typ
0201        | LLRegLit (typ, lit) -> typ
0202        | LLRegDud -> assert false in
0203
0204      (* Define all the structs *)
0205
0206      let rec lleasytyp_of = function (* TODO compromise *)
0207            LLBool -> i1_t
0208          | LLInt  -> i32_t
0209          | LLDouble -> L.double_type the_context
0210          | LLConstString -> str_t
0211          | LLVoid -> void_t
0212          | LLArray (typ, len) ->
0213             (match len with
0214                Some real_len -> L.array_type (lleasytyp_of typ)
```

```
real_len
0215               | None -> assert false
0216              )
0217           | LLStruct str -> i1_t
0218       in
0219
0220       let define_structs = (* a map from struct name to struct
type *)
0221         let define_struct map struc =
0222           let (struct_name, typelist) = struc in
0223           let struct_typ = L.named_struct_type the_context
struct_name in
0224           ignore (L.struct_set_body struct_typ (Array.of_list
(List.map lleasytyp_of typelist)) false);
0225           StringMap.add struct_name struct_typ map
0226         in
0227         List.fold_left define_struct StringMap.empty structs
0228       in
0229
0230       let rec lltyp_of = function (* TODO compromise *)
0231         | LLStruct str -> StringMap.find str define_structs
0232         | a -> lleasytyp_of a
0233       in
0234
0235       let get_struct_by_name struct_name =
0236           StringMap.find struct_name define_structs
0237       in
0238
0239       let promote lit = (* promote two  *)
0240         let llast_typ = extract_type lit in
0241         (match llast_typ with
0242          LLStruct str -> L.pointer_type (lltyp_of llast_typ)
0243          | LLArray (typ, len) -> L.pointer_type (lltyp_of typ)
0244          | _ -> lltyp_of llast_typ
0245         )
0246       in
0247
0248       let lit_to_llvalue = function
0249         | LLLitBool b -> L.const_int i1_t (if b then 1 else 0)
0250         | LLLitInt i -> L.const_int i32_t i
0251         | LLLitDouble f -> L.const_float double_t f
0252         | LLLitString s -> (L.define_global
"stringlit" (L.const_stringz the_context s) the_module)
0253         | _ -> assert false
0254       in
0255
0256       (* define global variables *)
0257       let define_globals =
0258         let define_global map var =
0259           let (gtyp, gname, glit) = var in
```

```
0260          let init_val = (match glit with
0261                            LLLitStruct list ->
0262                            let struct_name = (match gtyp with
LLStruct sname -> sname | _ -> assert false) in
0263                            let struct_lltype = get_struct_by_name
struct_name in
0264                            let list_llvalues = List.map
lit_to_llvalue list in
0265                            ignore(L.const_named_struct
struct_lltype (Array.of_list list_llvalues));
0266                            assert false;
0267                           | LLLitArray _ -> assert false
0268                           | x -> lit_to_llvalue x
0269                          ) in
0270          let global_llvalue = L.define_global gname init_val
the_module in
0271          StringMap.add gname global_llvalue map in
0272        List.fold_left define_global StringMap.empty globals in
0273
0274      (* Define the printf function *)
0275      let printf_t = L.var_arg_function_type i32_t [| str_t |] in
0276      let printf_func = L.declare_function "printf" printf_t
the_module in
0277      let int_format_str =
0278        let str_arr_ptr = L.define_global "fmti" (L.const_stringz
the_context "%d\n") the_module in
0279        L.const_in_bounds_gep str_arr_ptr [| L.const_int i32_t 0;
L.const_int i32_t 0|] and
0280          float_format_str =
0281            let formatter = "%."^(string_of_int
double_precision)^"f\n" in
0282            let str_arr_ptr = L.define_global
"fmti" (L.const_stringz the_context formatter) the_module in
0283        L.const_in_bounds_gep str_arr_ptr [| L.const_int i32_t 0;
L.const_int i32_t 0|] in
0284
0285      let define_funcs =
0286        let translate_func map func=
0287          let fname = func.llfname and
0288              fformals = Array.of_list (List.map promote
(func.llfformals))
0289          in
0290          let func_sign = L.function_type (lltyp_of
func.llfreturn) fformals in
0291          let func_def = L.define_function fname func_sign
the_module in
0292         StringMap.add fname (func_def, func) map
0293        in
0294        List.fold_left translate_func StringMap.empty funcs in
0295
```

```
0296      let get_func_by_name fname =
0297        if(StringMap.mem fname define_funcs= false)
0298        then (print_string (fname^" not found\n"); assert false;)
0299        else(
0300        let func_llvalue, _ = StringMap.find fname define_funcs in
0301        func_llvalue
0302        )
0303      in
0304
0305      let _ =
0306        let build_func func =
0307          let (the_function, _) = StringMap.find func.llfname
define_funcs in
0308          let builder = L.builder_at_end the_context
(L.entry_block the_function) in
0309
0310          let formals_list = Array.to_list (L.params the_function)
in
0311
0312          let get_reg_typ_name = function
0313              LLRegLabel (typ, str) -> (typ, str)
0314            | LLRegLit (typ,lit) -> (typ,"nonameliteral")
0315            | LLRegDud -> assert false in
0316
0317          let build_formals = (* this is a map from formal name to
its stack ptr *)
0318            let build_formal map formal_def formal_param =
0319              let (formal_type,formal_name) = get_reg_typ_name
formal_def in
0320              (match formal_type with
0321                 LLArray (typ, len) ->
0322                 let double_ptr = L.build_alloca (L.type_of
formal_param) "alloc_ptr_arr_formal" builder in
0323                 ignore(L.build_store formal_param double_ptr
builder);
0324                 StringMap.add formal_name double_ptr map
0325               | LLStruct struct_name ->
0326                  let double_ptr = L.build_alloca (L.type_of
formal_param) "alloc_ptr_struct_formal" builder in
0327                  ignore(L.build_store formal_param double_ptr
builder);
0328                  StringMap.add formal_name double_ptr map
0329               | _ -> let formal_ptr = L.build_alloca (lltyp_of
formal_type) formal_name builder in
0330                      ignore(L.build_store formal_param formal_ptr
builder);
0331                      StringMap.add formal_name formal_ptr map
0332             ) in
0333            List.fold_left2 build_formal StringMap.empty
func.llfformals formals_list
```

```
0334          in
0335
0336          let build_locals = (* this is a map from local name to
its stack ptr *)
0337            let build_local map local_def =
0338              let (local_type, local_name) = get_reg_typ_name
local_def in
0339              let local_ptr =
0340                (match local_type with
0341                   LLArray (typ, len) ->
0342                   let aggptr = L.build_alloca (lltyp_of
local_type) local_name builder in
0343                   let ptr_to_first = L.build_in_bounds_gep aggptr
[| L.const_int i32_t 0; L.const_int i32_t 0 |]
0344
"build_local_arr" builder in
0345                   let double_ptr = L.build_alloca (L.type_of
ptr_to_first) "alloc_ptr_arr" builder in
0346                   ignore(L.build_store ptr_to_first double_ptr
builder);
0347                   double_ptr
0348                 | LLStruct struct_name ->
0349                    let struct_typ = get_struct_by_name
struct_name in
0350                    let struct_ptr = L.build_alloca struct_typ
"build_local_struct" builder in
0351                    let double_ptr = L.build_alloca (L.type_of
struct_ptr) "alloc_ptr_struct" builder in
0352                    ignore(L.build_store struct_ptr double_ptr
builder);
0353                    double_ptr
0354                 | _ -> L.build_alloca (lltyp_of local_type)
local_name builder
0355                ) in
0356              StringMap.add local_name local_ptr map in
0357            List.fold_left build_local StringMap.empty
func.llflocals
0358          in
0359
0360          let define_blocks =
0361            let define_block map block_def =
0362              let block_name = block_def.llbname and block_stmts =
block_def.llbbody in
0363              let block_llvalue = L.append_block the_context
block_name the_function in
0364              StringMap.add block_name (block_stmts,
block_llvalue) map in
0365            List.fold_left define_block StringMap.empty
func.llfblocks
0366          in
```

```
0367
0368          let get_block_by_name bname =
0369            let stmts, block_llvalue = StringMap.find bname
define_blocks in
0370            block_llvalue in
0371
0372          (* helper function starts here *)
0373          let llvalue_of_lit typ block_builder = function
0374              LLLitBool bool -> L.const_int i1_t (if bool then 1
else 0)
0375            | LLLitInt int -> L.const_int i32_t int
0376            | LLLitDouble double ->L.const_float double_t double
0377            | LLLitString str -> L.build_global_stringptr str
"globalstr" block_builder
0378            | LLLitArray litlist -> assert false
0379            | LLLitStruct litlist -> assert false
0380          in
0381
0382          let get_reg block_builder = function
0383              LLRegLabel (typ, regname) ->
0384              let ret =
0385              if (StringMap.mem regname build_formals)
0386              then (StringMap.find regname build_formals)
0387              else (
0388                if (StringMap.mem regname build_locals)
0389                then (StringMap.find regname build_locals)
0390                else ( if(StringMap.mem regname define_globals)
0391                       then (StringMap.find regname
define_globals)
0392                       else (print_string (regname^" not
found\n"); assert false)
0393                     )
0394              ) in
0395              ret
0396            | LLRegLit (typ, literal) ->
0397               let literal_ptr = L.build_alloca (lltyp_of typ)
"lit_alloc_inst" block_builder in
0398                ignore(L.build_store (llvalue_of_lit typ
block_builder literal) literal_ptr block_builder);
0399                literal_ptr
0400            | LLRegDud -> assert false
0401          in
0402
0403          let load_reg ptrreglabel block_builder =
0404            let reg = (get_reg block_builder ptrreglabel) in
0405            L.build_load  reg "loadinst" block_builder
0406          in
0407
0408          let store_reg ptrreglabel val_to_store block_builder=
0409            L.build_store val_to_store (get_reg block_builder
```

```
ptrreglabel) block_builder in
0410
0411          let make_tuple list element =
0412            let bundle a = (a,element) in
0413            List.map bundle list
0414          in
0415
0416          let transform_funclabel_list_to_llvalue label_list
block_builder =
0417            let builder_list = List.map (fun _ -> block_builder)
label_list in
0418            let result = List.map2 load_reg label_list
builder_list in
0419            Array.of_list result in
0420
0421          (* helper functions end here *)
0422
0423          let build_terminator block_builder = function
0424              LLBlockReturn label -> L.build_ret (load_reg label
block_builder) block_builder
0425            | LLBlockReturnVoid -> L.build_ret_void block_builder
0426            | LLBlockBr (label,brname1,brname2) ->
0427               L.build_cond_br (load_reg label block_builder)
0428                               (get_block_by_name brname1)
(get_block_by_name brname2) block_builder
0429            | LLBlockJmp brname -> L.build_br (get_block_by_name
brname) block_builder
0430          in
0431
0432          let bind_block_to_stmt stmts = function
0433              "entry" -> make_tuple stmts builder
0434            | bname -> let (_,block_llvalue) = StringMap.find
bname define_blocks in
0435                       make_tuple stmts (L.builder_at_end
the_context block_llvalue)
0436          in
0437
0438          let get_arr_ptr_by_labels agglabel indexlabel
block_builder=
0439            let aggreg = load_reg agglabel block_builder and
indexreg = load_reg indexlabel block_builder in
0440            L.build_in_bounds_gep aggreg [| indexreg |]
"get_arr_inst" block_builder
0441          in
0442
0443          let get_struct_ptr_by_labels agglabel index
block_builder=
0444            let aggreg = load_reg agglabel block_builder in
0445            L.build_struct_gep aggreg index "get_struct_inst"
block_builder
```

```
0446          in
0447
0448          let build_stmt map block_stmt=
0449            let (stmt, block_builder) = block_stmt in
0450            let stmt_llvalue =
0451              (match stmt with
0452                 LLBuildPrintCall label
0453                 -> let argarr = (match (extract_type label) with
0454                                    LLInt -> [|int_format_str;
(load_reg label block_builder) |]
0455                                  | LLDouble -> [|
float_format_str; (load_reg label block_builder) |]
0456                                  | LLConstString -> [| (load_reg
label block_builder) |]
0457                                  | _ -> [| |]) in (* TODO are we
printing more things *)
0458                 L.build_call printf_func argarr "printwhatever"
block_builder
0459               | LLBuildCall (fname,label_list,labelret) ->
0460                  let func_llvalue = get_func_by_name fname and
0461                      func_formal_array_llvalue =
transform_funclabel_list_to_llvalue label_list block_builder in
0462                  (match labelret with
0463                     Some label->
0464                     let call_ret_reg =
0465                       L.build_call func_llvalue
func_formal_array_llvalue "call_ret" block_builder in
0466                     ignore(store_reg label call_ret_reg
block_builder);
0467                     call_ret_reg
0468                   | None -> ignore(L.build_call func_llvalue
func_formal_array_llvalue "" block_builder);
0469                             L.const_int i32_t 0
0470                  )
0471               | LLBuildBinOp (optyp,label1,label2,labelresult) ->
0472                  let reg1 = load_reg label1 block_builder and
reg2 = load_reg label2 block_builder in
0473                  let icmpfinder = function
0474                      LLLT -> L.Icmp.Slt
0475                    | LLEQ -> L.Icmp.Eq
0476                    | LLGT -> L.Icmp.Sgt
0477                    | LLLE -> L.Icmp.Sle
0478                    | LLGE -> L.Icmp.Sge
0479                   and
0480                    fcmpfinder = function
0481                    | LLFLT -> L.Fcmp.Olt
0482                    | LLFEQ -> L.Fcmp.Oeq
0483                    | LLFGT -> L.Fcmp.Ogt
0484                    | LLFLE -> L.Fcmp.Ole
0485                    | LLFGE-> L.Fcmp.Oge
```

```
0486                     and
0487                       iopfinder = function
0488                       | LLAdd -> L.build_add
0489                       | LLSub -> L.build_sub
0490                       | LLMul -> L.build_mul
0491                       | LLDiv -> L.build_sdiv
0492                       | LLMod -> L.build_srem
0493                       | LLAnd -> L.build_and
0494                       | LLOr -> L.build_or
0495                     and
0496                       fopfinder = function
0497                       | LLFAdd -> L.build_fadd
0498                       | LLFSub -> L.build_fsub
0499                       | LLFMul -> L.build_fmul
0500                       | LLFDiv -> L.build_fdiv
0501                   in
0502                   let regbinop = (match optyp with
0503                                       LLIop iop -> (iopfinder iop)
reg1 reg2 "iopinst" block_builder
0504                                     | LLFop fop -> (fopfinder fop)
reg1 reg2 "fopinst" block_builder
0505                                     | LLIBop ibop -> L.build_icmp
(icmpfinder ibop) reg1 reg2 "ibopinst" block_builder
0506                                     | LLFBop fbop -> L.build_fcmp
(fcmpfinder fbop) reg1 reg2 "icmpinst" block_builder
0507                                    ) in
0508                   store_reg labelresult regbinop block_builder
0509                | LLBuildArrayLoad (agglabel,indexlabel,destlabel)
->
0510                   let elementptr = get_arr_ptr_by_labels agglabel
indexlabel block_builder in
0511                   let derefelement = L.build_load elementptr
"arrload_deref" block_builder in
0512                   store_reg destlabel derefelement block_builder
0513                | LLBuildArrayStore (agglabel, indexlabel,
fromlabel) ->
0514                   let elementptr = get_arr_ptr_by_labels agglabel
indexlabel block_builder in
0515                   let val_to_store = load_reg fromlabel
block_builder in
0516                   L.build_store val_to_store elementptr
block_builder
0517                | LLBuildStructLoad (agglabel,index,destlabel) ->
0518                   let elementptr = get_struct_ptr_by_labels
agglabel index block_builder in
0519                   let derefelement = L.build_load elementptr
"structload_deref" block_builder in
0520                   store_reg destlabel derefelement block_builder
0521                | LLBuildStructStore (agglabel, index, fromlabel) -
>
```

```
0522                   let elementptr = get_struct_ptr_by_labels
agglabel index block_builder in
0523                   let val_to_store = load_reg fromlabel
block_builder in
0524                   L.build_store val_to_store elementptr
block_builder
0525                 | LLBuildTerm terminator -> build_terminator
block_builder terminator
0526                 | LLBuildAssign (tolabel, fromlabel) ->
0527                   let val_to_assign = load_reg fromlabel
block_builder in
0528                   store_reg tolabel val_to_assign block_builder
0529                 | LLBuildNoOp -> L.const_int i32_t 0
0530               )
0531                in
0532             StringMap.add ("stmt"^func.llfname) stmt_llvalue map
0533           in
0534           let bundled_stmts = bind_block_to_stmt func.llfbody
"entry" in
0535           ignore(List.fold_left build_stmt StringMap.empty
bundled_stmts); (*build the entry block*)
0536           (* now let's build each block *)
0537           let build_blocks =
0538             let build_block block =
0539               let bname = block.llbname in
0540               let bundled_stmts = bind_block_to_stmt block.llbbody
bname in
0541               ignore(List.fold_left build_stmt StringMap.empty
bundled_stmts);
0542               ()
0543             in
0544             List.iter build_block func.llfblocks
0545           in
0546           let _ = build_blocks in
0547           () (* end of build_funcs *)
0548         in
0549         List.iter build_func funcs in
0550
0551       (*let _ = Llvm_analysis.assert_valid_module the_module in*)
0552       the_module
```

./shux/src/backend/ast_sast.ml

```
0001    open Ast
0002    open Sast
0003    open Semant
0004
```

```
0005    type svar = {
0006        id : string;
0007        scope : Sast.sscope;
0008        svar_type : Sast.styp;
0009    }
0010
0011    (* keep track of names and types *)
0012    type strans_env = {
0013        variables : svar list VarMap.t;
0014        sfn_decl : sfn_decl VarMap.t;
0015        sstruct_map : sstruct_def VarMap.t
0016    }
0017
0018    let print_styp = function
0019      | SInt -> "int"
0020      | SFloat -> "float"
0021      | SString -> "string"
0022      | SBool -> "bool"
0023      | SStruct(b,e) -> "struct"
0024      | SArray(s,i) -> "array"
0025      | SPtr -> "ptr"
0026      | SVoid  -> "void"
0027
0028    let print_bind_name = function
0029        | SBind(s, name, scope) -> print_string name
0030
0031    let rec to_styp senv = function
0032        | Some x -> (match x with
0033        | Int -> SInt
0034        | Float -> SFloat
0035        | String -> SString
0036        | Bool -> SBool
0037        | Vector(i) -> SArray(SFloat, Some i)
0038        | Struct(ns) ->  let s = flatten_ns_list ns in
0039                         let sstruct_binds = (VarMap.find s
senv.sstruct_map).ssfields
0040
                    in SStruct(s, sstruct_binds)
0041        | Array(t,i) -> let n_styp = to_styp senv (Some t) in
SArray(n_styp, i)
0042        | Ptr -> SPtr
0043        | Void -> SVoid)
0044        | None -> SVoid
0045
0046    (* iorf: true for int, false for float *)
0047    and to_sbin_op iorf = function
0048        | Add -> if iorf then SBinopInt SAddi else SBinopFloat
SAddf
0049        | Sub -> if iorf then SBinopInt SSubi else SBinopFloat
SSubf
```

```
0050         | Mul -> if iorf then SBinopInt SMuli else SBinopFloat
SMulf
0051         | Div -> if iorf then SBinopInt SDivi else SBinopFloat
SDivf
0052         | Exp -> if iorf then SBinopInt SExpi else SBinopFloat
SExpf
0053         | Eq  -> if iorf then SBinopInt SEqi else SBinopFloat SEqf
0054         | Lt  -> if iorf then SBinopInt SLti else SBinopFloat SLtf
0055         | Gt  -> if iorf then SBinopInt SGti else SBinopFloat SGtf
0056         | Neq -> if iorf then SBinopInt SNeqi else SBinopFloat
SNeqf
0057         | Leq -> if iorf then SBinopInt SLeqi else SBinopFloat
SLeqf
0058         | Geq -> if iorf then SBinopInt SGeqi else SBinopFloat
SGeqf
0059         | Mod -> SBinopInt SMod
0060         | LogAnd -> SBinopBool SLogAnd
0061         | LogOr -> SBinopBool SLogOr
0062         | Filter -> SBinopFn SFilter
0063         | Map -> SBinopFn SMap
0064         | For -> SBinopGn SFor
0065         | Index -> SBinopPtr SIndex
0066         | Do -> SBinopGn SFor (* will never be called *)
0067
0068     and get_styp_from_sexpr = function
0069         | SLit(s,_) -> s
0070         | SId(s,_,_) -> s
0071         | SLookback(s,_,_) -> s
0072         | SAccess(s,_,_) -> s
0073         | SBinop(s,_,_,_) -> s
0074         | SAssign(s,_,_) -> s
0075         | SKnCall(s,_,_) -> s
0076         | SGnCall(s,_,_) -> s
0077         | SLookbackDefault(s,_,_,_) -> s
0078         | SUnop(s,_,_) -> s
0079         | SCond(s,_,_,_) -> s
0080         | _ -> SVoid (* jooooohn *)
0081
0082     and to_slit senv = function
0083         | LitInt(i) -> SLitInt(i)
0084         | LitFloat(f) -> SLitFloat(f)
0085         | LitBool(b) -> SLitBool(b)
0086         | LitStr(s) -> SLitStr(s)
0087         | LitKn(l) -> SLitKn(to_slambda senv l)
0088         | LitVector(el) -> SLitArray(List.map (get_sexpr senv) el)
0089         | LitArray(e) -> SLitArray(List.map (get_sexpr senv) e)
0090         | LitStruct(s,e) ->
0091               let translate_structfield senv = function
0092                   | StructField(name, expr) ->(name, get_sexpr
senv expr)
```

```
0093              in SLitStruct(flatten_ns_list s, List.map
(translate_structfield senv) e)
0094
0095    and slit_to_styp senv = function
0096        | SLitInt(i) -> SInt
0097        | SLitFloat(f) -> SFloat
0098        | SLitBool(b) -> SBool
0099        | SLitStr(s) -> SString
0100        | SLitKn(l) -> l.slret_typ
0101        | SLitArray(elist) -> SArray(get_styp_from_sexpr (List.hd
elist), Some (List.length elist))
0102        | SLitStruct(name, slist) ->
0103              let sdef = VarMap.find name senv.sstruct_map
0104              in SStruct(name, sdef.ssfields)
0105
0106
0107    and to_sunop iorf = function
0108        | LogNot -> SLogNot
0109        | Neg -> if iorf then SNegi else SNegf
0110        | Pos -> raise (Failure "Pos isnt supposed to exist in
SAST")
0111
0112    and to_sbind env = function
0113        | Bind(m, t, s) -> SBind(to_styp env (Some t), s,
mut_to_scope m)
0114
0115
0116    and get_inherited body decls retexpr =
0117        let uniq l =
0118            let same (t1,s1) (t2,s2) = t1=t2 && s1=s2
0119            in let rec exists s b = function
0120                | [] -> b
0121                | hd::tl -> if (same hd s) then exists s true tl
0122                                           else exists s b tl
0123            in let rec uniq_rec uniques = function
0124                 | [] -> uniques
0125                 | hd::tl ->
0126                    if (exists hd false uniques)
0127                             then uniq_rec uniques tl
0128                             else uniq_rec (hd::uniques) tl
0129            in uniq_rec [] l
0130        in let match_sbind typ name b binding =
0131            if b then b else (match binding with
0132                | SBind(t, s, scope) -> t=typ && s=name)
0133        in let rec add_inherit decls inherits = function
0134            | [] -> inherits
0135            | (t,s)::tl -> if (List.fold_left (match_sbind t s)
false decls)
0136                           then add_inherit decls inherits tl
0137                           else let new_bind = SBind(t, s,
```

```
SLocalVal)
                            in let new_inherits =
new_bind::inherits
                            in add_inherit decls new_inherits tl
        in let rec detect_names = function
            | SId(t,s,c) -> [(t,s)]
                                        | SAccess(t, se, name) ->
                                                                let
expr_names = detect_names se
                in (t,name)::expr_names
            | SBinop(st, se1, bop, se2) ->
                let names1 = detect_names se1
                and names2 = detect_names se2
                in names1@names2
            | SAssign(st, se1, se2) ->
                let names1 = detect_names se1
                and names2 = detect_names se2
                in names1@names2
            | SKnCall(st, kname, form_list) ->
                let exprs = List.map fst form_list
                in let name_list = List.map detect_names exprs
                in List.flatten name_list
            | SGnCall(st, gname, form_list) ->
                let exprs = List.map fst form_list
                in let name_list = List.map detect_names exprs
                in List.flatten name_list
           | SUnop(t, sun, se) -> detect_names se
           | SCond(t, se1, se2, se3) ->
                let name_list = List.map detect_names
[se1;se2;se3]
                in List.flatten name_list
           | _ -> []
        in let rec get_inherited_rec decls inherits = function
            | [] -> inherits
            | hd::tl -> get_inherited_rec decls ((detect_names
(fst hd))::inherits) tl
        in let inherited_body = get_inherited_rec decls [] body
        in let full_body = (detect_names retexpr)::inherited_body
        in let flat_names = List.flatten full_body
        in let uniq_names = uniq flat_names
        in let inherits = add_inherit decls [] uniq_names
                    in inherits

    and to_slambda senv l =
        let lformals = translate_fn_formals l.lformals senv
        in let aret = l.lret_expr
        in let rec hoist_lambda (vdecls, exprs) = function
            | [] -> (List.rev vdecls, List.rev exprs)
            | VDecl(b,e)::tl -> (match e with
                | Some e -> let id = get_bind_name b
```

```
0183                             in let asn = Assign(Id([id]), e)
0184                             in hoist_lambda (VDecl(b, Some
e)::vdecls, asn::exprs) tl
0185                 | None -> hoist_lambda (VDecl(b,e)::vdecls, exprs)
tl)
0186             | Expr(e)::tl -> hoist_lambda (vdecls, e::exprs) tl
0187         in let vdecl_to_local senv = function
0188             | VDecl(b,e) -> to_sbind senv b
0189             | Expr(e) -> raise (Failure "Lambda hoisting failed")
0190         in let (decls, abody) = hoist_lambda ([],[]) l.lbody
0191         in let llocals = List.map (vdecl_to_local senv) decls
0192         in let l_env = List.fold_left place_formals senv
[llocals;lformals]
0193         in let body_intermediate = List.map (get_sexpr l_env)
abody
0194         in let lbody = List.map (fun x -> (x, get_styp_from_sexpr
x)) body_intermediate
0195         in (match aret with
0196             | Some x -> let sret_expr = get_sexpr l_env x
0197                         in let inherited = get_inherited lbody
(llocals@lformals) sret_expr
0198                         in let sret_typ = get_styp_from_sexpr
sret_expr
0199                         in { slret_typ = sret_typ; slformals =
lformals;
0200                              sllocals = llocals; slinherit =
inherited;
0201                              slbody = lbody; slret_expr = Some
(sret_expr, sret_typ) }
0202             | None -> let inherited = get_inherited lbody
(llocals@lformals) SExprDud
0203                       in
0204                       { slret_typ = SVoid; slformals = lformals;
0205                         sllocals = llocals; slinherit = inherited;
0206                         slbody = lbody; slret_expr = None; })
0207
0208     and mut_to_scope = function
0209         | Mutable -> SLocalVar
0210         | Immutable -> SLocalVal
0211
0212     and get_sfn_name = function
0213         | SGnDecl(g) -> g.sgname
0214         | SKnDecl(k) -> k.skname
0215         | SExDud(e) -> e.skname
0216
0217     and translate_struct_defs env struct_def =
0218         {ssname  = struct_def.sname; ssfields = List.map (to_sbind
env) struct_def.fields }
0219
0220     and get_struct_bind name = function
```

```
0221        | [] -> assert(false)
0222        | SBind(t,n,scope)::tl -> if name=n then SBind(t,n,scope)
0223                                   else get_struct_bind name tl
0224
0225    (* translate expr -> sexpr. uses senv for bookkeeping *)
0226    and get_sexpr senv = function
0227        | Lit(a) -> let sliteral = to_slit senv a
0228                    in SLit(slit_to_styp senv sliteral, sliteral)
0229        | Id(ns) -> let s = flatten_ns_list ns
0230                    in if VarMap.mem s senv.variables then
0231                    let v = List.hd (VarMap.find s senv.variables)
0232                                           in SId(v.svar_type, s,
v.scope)
0233                    else if VarMap.mem s senv.sfn_decl
0234                     then SId(SPtr, s, SLocalVal)
0235                    else assert(false)
0236        | Lookback(nstr, i) -> let str = flatten_ns_list nstr
0237                               in let var = List.hd (VarMap.find
str senv.variables)
0238                               in let typ = var.svar_type
0239              in SLookback(typ, str, i)
0240        | Binop(e1, bin_op, e2) ->
0241           let st1 = get_sexpr senv e1 in (match bin_op with
0242               | Add | Sub | Mul | Div | Mod | Exp ->
0243                                 let sbinop = (match
get_styp_from_sexpr st1 with
0244                                                                 | SInt ->
to_sbin_op true bin_op
0245                                                                 | SFloat
-> to_sbin_op false bin_op
0246                                                                 | _ ->
raise (Failure "Not Integer/Float type on binop")) in
0247                                    SBinop(get_styp_from_sexpr
st1, st1, sbinop, get_sexpr senv e2)
0248                 | Eq | Lt | Gt | Neq | Leq | Geq ->
0249                    let sbinop = (match get_styp_from_sexpr st1
with
0250                                                                 | SInt ->
to_sbin_op true bin_op
0251                                                                 | SFloat
-> to_sbin_op false bin_op
0252                                                                 | _ ->
raise (Failure "Not Integer/Float type on binop")) in
0253                       SBinop(SBool, st1, sbinop, get_sexpr senv
e2)
0254                 | LogAnd | LogOr -> SBinop(SBool, st1, to_sbin_op
true bin_op, get_sexpr senv e2)
0255                 | Filter -> let expr2 = (match e2 with
0256                                  | Id(nn) -> let n = flatten_ns_list
nn
```

```
0257                                            in SId(SBool, n,
SKnLambda([]))
0258                                 | _ -> get_sexpr senv e2)
0259                           in SBinop(SArray(get_styp_from_sexpr st1,
None), st1, SBinopFn SFilter, expr2)
0260                   | Map ->  let expr2 = (match e2 with
0261                                 | Id(nn) -> let n = flatten_ns_list
nn
0262                                             in let kn =
VarMap.find n senv.sfn_decl
0263                                             in let kn_typ = (match
kn with
0264                                                 | SGnDecl(sgn) ->
assert(false)
0265                                                 | SExDud(s) ->
assert(false)
0266                                                 | SKnDecl(skn) ->
skn.skret_typ)
0267                                             in SId(kn_typ, n,
SKnLambda([]))
0268                                 | _ -> get_sexpr senv e2)
0269                                  in
SBinop(SArray(get_styp_from_sexpr expr2, None),
0270
                                        st1, SBinopFn SMap, expr2)
0271                   | Index -> let st1type = get_styp_from_sexpr st1
in
0272                       (match st1type with
0273                      | SArray(s, i) -> SBinop(s, st1, SBinopPtr
SIndex, get_sexpr senv e2)
0274                      | _ -> raise (Failure "Indexing needs
sarray"))
0275                   | For -> SBinop(SArray(get_styp_from_sexpr
(get_sexpr senv e2), None), st1, SBinopGn SFor, get_sexpr senv e2)
0276                   | Do -> let st2 = get_sexpr senv e2
0277                           in let st2_typ = get_styp_from_sexpr st2
0278                           in let st1_typ = get_styp_from_sexpr st1
0279                           in if st1_typ != SInt then assert(false)
else (* needs to be an integer. semant should already have handled
this *)
0280                           let fake_for = SBinop(SArray(st2_typ,
None), st1, SBinopGn SFor, st2)
0281                           in let index = SBinop(st1_typ,st1,
SBinopInt SSubi, SLit(SInt, SLitInt(-1)))
0282                           in SBinop(st2_typ, fake_for, SBinopPtr
SIndex, index))
0283                   | Assign(e1, e2) -> let st1 = get_sexpr senv e1
in
0284
        SAssign(get_styp_from_sexpr st1, st1, get_sexpr senv e2)
```

```
0285                  | Call(s, elist) -> (match s with
0286                      | Some ns -> let s = flatten_ns_list ns
0287                                   in let sexpr_list = List.map
(get_sexpr senv) elist
0288
in let call_formals = List.map (fun x -> (x, get_styp_from_sexpr x))
sexpr_list
0289                                                  in let f =
VarMap.find s senv.sfn_decl in (match f with
0290                                          | SGnDecl(gn) ->
SGnCall(gn.sgret_typ, s, call_formals)
0291                                          | SKnDecl(kn) ->
SKnCall(kn.skret_typ, s, call_formals)
0292                             | SExDud(en) -> SExCall(en.skret_typ, s,
call_formals))
0293                      | None -> SGnCall(SInt, "_", []))
0294                  | Uniop(u, e) -> (match u with
0295                      | LogNot -> let st1 = get_sexpr senv e in
0296                                  SUnop(get_styp_from_sexpr st1,
to_sunop true u, st1)
0297                      | Neg -> let st1 = get_sexpr senv e in
0298                                                    let sunop =
(match get_styp_from_sexpr st1 with
0299                                                                  |
SInt -> to_sunop true u
0300                                                                  |
SFloat -> to_sunop false u
0301                                                                  |
_ -> raise (Failure "Not Integer/Float type on unnop"))
0302                                                 in
SUnop(get_styp_from_sexpr st1, sunop, st1)
0303                      | Pos -> get_sexpr senv e)
0304                | LookbackDefault(e1, e2) ->
0305                                                        let se1
= get_sexpr senv e1
0306                                                        in let
se2 = get_sexpr senv e2
0307                                                        and i =
(match e1 with
0308                         | Lookback(str, i) -> i
0309                         | _ -> raise (Failure "LookbackDefault not
preceded by Lookback expression"))
0310                      in SLookbackDefault(get_styp_from_sexpr se1, i,
se1, se2)
0311                | Cond(e1, e2, e3) ->
0312                                                      let se1
= get_sexpr senv e1
0313                                                      and se2
= get_sexpr senv e2
0314                                                  and se3 =
```

```
get_sexpr senv e3
0315                                                                in
SCond(get_styp_from_sexpr se2, se1, se2, se3)
0316                 | Access(e,  str) -> let sex = get_sexpr senv e
0317                                     in let styp =
get_styp_from_sexpr sex
0318                                     in let sbinds = (match styp
with
0319                                         | SStruct(s, sbind) ->
sbind
0320                                         | _ -> assert(false))
0321                                     in let bind = get_struct_bind
str sbinds
0322                                     in let t  = (match bind with
0323                                         | SBind(t, s, scope) ->
t)
0324
                       in SAccess(t, sex , str)
0325
0326    (* this translates letdecls and also builds an environment for
further translation *)
0327    and translate_letdecl senv globals =
0328        let global_mapper (sglobals, senv) = function
0329           | LetDecl(b,e) ->
0330                                                            let name =
get_bind_name b
0331                                                            and st =
to_styp senv (Some (get_bind_typ b))
0332                                                            in let
new_bind = SBind(st, name, SGlobal)
0333                                                            in let
new_var = { id = name; scope = SGlobal;
0334

svar_type = st }
0335                                                            in let
new_varmap =
0336
        if VarMap.mem name senv.variables
0337
                         then let varlist = VarMap.find name
senv.variables
0338
                                              in VarMap.add name
(new_var::varlist) senv.variables
0339
                         else VarMap.add name [new_var] senv.variables
0340                                                            in let
new_env = { variables = new_varmap; sfn_decl = senv.sfn_decl;
0341
```

```
sstruct_map = senv.sstruct_map }
0342                                                            in
(SLetDecl(new_bind, get_sexpr senv e)::sglobals, new_env)
0343            | StructDef(s) ->
0344                                                            let
to_struct_binds senv b =
0345
                        let name = get_bind_name b
0346
                        and t = get_bind_typ b
0347
                        in SBind(to_styp senv (Some t), name,
SStructField)
0348                                                            in let sdef
= {ssname = s.sname; ssfields = List.map (to_struct_binds senv)
s.fields}
0349                                                            in let
new_structs = VarMap.add s.sname sdef senv.sstruct_map
0350                                                            in let
new_env = { variables = senv.variables; sfn_decl = senv.sfn_decl;
0351

sstruct_map = new_structs }
0352                                                            in
((SStructDef sdef)::sglobals, new_env)
0353            | ExternDecl(e) ->
0354                                                            let
sret_type = to_styp senv e.xret_typ
0355                                                            and
new_formals = List.map (to_sbind senv) e.xformals (* they are all
immutables *)
0356                                                            in let
new_extern = { sxalias = e.xalias;
0357

                sxfname = e.xfname;
0358

                sxret_typ = sret_type;
0359

                sxformals = new_formals; }
0360
0361                                                            in let
new_func = { skname = e.xalias; skret_typ = sret_type;
0362

        skformals = new_formals; sklocals = [];
0363
```

```
        skbody = []; skret_expr = None }
                                                                in let
new_fnmap = VarMap.add new_func.skname (SExDud new_func) senv.sfn_decl
                                                                in let
new_env = { variables = senv.variables; sfn_decl = new_fnmap;

sstruct_map = senv.sstruct_map }
                                                                in
((SExternDecl new_extern)::sglobals, new_env)
        in let (sglob, new_env) = List.fold_left global_mapper
([], senv) globals in (List.rev sglob, new_env)

    (* used in kn and fn translations *)
    and translate_fn_formals formals senv =
       List.map (fun (Bind(m,t,s)) -> SBind(to_styp senv (Some t),
s, SLocalVal)) formals

    (* pushing formal arguments to senv *)
    and push_formal svar senv =
        let new_variables =
           if VarMap.mem svar.id senv.variables then
              let old_varlist = VarMap.find svar.id senv.variables
              in VarMap.add svar.id (svar::old_varlist)
senv.variables
           else
              VarMap.add svar.id [svar] senv.variables
        in { variables = new_variables; sfn_decl = senv.sfn_decl;
             sstruct_map = senv.sstruct_map }

    (* place a list of SBinds into an environment and return new
env *)
    and place_formals senv formals =
        let place_formal senv = function
            | SBind(t, name, s) ->
              let svar = { id = name; scope = s; svar_type = t }
              in push_formal svar senv
        in List.fold_left place_formal senv formals

    (* traverse an expression tree and find the maximum lookback
value *)
    and lookback_walk exprs =
        (* get max integer from a list of integers *)
        let get_max l =
            let rec get_max_helper maximum = function
                | [] -> maximum
                | hd:: tl ->
                    if hd > maximum then get_max_helper hd tl
                                    else get_max_helper maximum tl
```

```
0402            in get_max_helper 0 l
0403        in let rec lookback_rec_walk maxi = function
0404            | Lit(l) -> maxi
0405            | Id(sl) -> maxi
0406            | Lookback (sl, i) -> if i > maxi then i else maxi
0407            | Binop(e1, bin_op, e2) ->
0408                let i1 = lookback_rec_walk maxi e1
0409                in let i2 = lookback_rec_walk maxi e2
0410                in get_max [i1;i2;maxi]
0411            | Assign(e1, e2) ->
0412                let i1 = lookback_rec_walk maxi e1
0413                in let i2 = lookback_rec_walk maxi e2
0414                in get_max [i1;i2;maxi]
0415            | Call(str, elist) ->
0416                let ilist = List.map (lookback_rec_walk maxi)
elist
0417                in get_max ilist
0418            | Uniop(u,e) -> let i = lookback_rec_walk maxi e
0419                            in if i > maxi then i else maxi
0420            | LookbackDefault(e1, e2) ->
0421                 let i1 = lookback_rec_walk maxi e1
0422                                                                 in
let i2 = lookback_rec_walk maxi e2
0423                                                                 in
get_max [i1;i2;maxi]
0424            | Cond(e1,e2,e3) ->
0425                let i1 = lookback_rec_walk maxi e1
0426                in let i2 = lookback_rec_walk maxi e2
0427                in let i3 = lookback_rec_walk maxi e3
0428                in get_max [i1;i2;i3;maxi]
0429            | Access(e1, str) -> let i = lookback_rec_walk maxi e1
0430                                 in if i > maxi then i else maxi
0431         in let ivalues = List.map (lookback_rec_walk 0) exprs
0432         in get_max ivalues
0433
0434    and translate_kn_decl senv kn =
0435        let name = kn.fname
0436        and ret_typ = to_styp senv kn.ret_typ
0437        and knformals = translate_fn_formals kn.formals senv
0438        in let rec hoist_body (vdecls, local_exprs) = function
0439            | [] -> (List.rev vdecls, List.rev local_exprs)
0440            | VDecl(b,e)::tl -> (match e with
0441                                 | Some e -> let id = get_bind_name b
0442
            in let asn = Assign(Id([id]), e)
0443
                in hoist_body (VDecl(b,Some e)::vdecls,
asn::local_exprs) tl
0444                                 | None -> hoist_body
(VDecl(b,e)::vdecls, local_exprs) tl)
```

```
0445             | Expr(e)::tl -> hoist_body (vdecls, e::local_exprs)
tl
0446         in let vdecl_to_local senv = function
0447             | VDecl(b,e) -> to_sbind senv b
0448             | Expr(e) -> raise (Failure "hoisting failed.
vdecl_to_local should only accept VDecl")
0449         in let (vdecls, local_exprs) = hoist_body ([],[]) kn.body
0450         in let klocals = List.map (vdecl_to_local senv) vdecls
0451         in let kn_env = List.fold_left place_formals senv
[knformals;klocals]
0452         in let body_intermediate = List.map (get_sexpr kn_env)
local_exprs
0453         in let kbody = List.map (fun x -> (x, get_styp_from_sexpr
x)) body_intermediate
0454         in (match kn.ret_expr with
0455             | Some x -> let kret_expr = (get_sexpr kn_env x,
ret_typ)
0456                                                                 in
{ skname = name; skret_typ = ret_typ; skformals = knformals;
0457
                 sklocals = klocals; skbody = kbody; skret_expr = Some
kret_expr }
0458             | None -> { skname = name; skret_typ = ret_typ;
skformals = knformals;
0459                                                       sklocals =
klocals; skbody = kbody; skret_expr = None })
0460
0461     and translate_gn_decl senv gn =
0462         let name = gn.fname
0463         and ret_typ = to_styp senv gn.ret_typ
0464         and gnformals = translate_fn_formals gn.formals senv
0465         in let rec hoist_body (vals, vars, exprs) = function
0466             | [] -> (List.rev vals, List.rev vars, List.rev exprs)
0467             | VDecl(b,e)::tl -> let mut = get_bind_mut b in
0468                (match e with
0469                  | Some e -> let id = get_bind_name b
0470                              in let asn = Assign(Id([id]), e)
0471                              in if mut = Mutable then
0472                                   hoist_body (vals, VDecl(b, Some
e)::vars, asn::exprs) tl
0473                              else hoist_body (VDecl(b, Some
e)::vals, vars, asn::exprs) tl
0474                  | None -> if mut = Mutable then
0475                              hoist_body(vals, VDecl(b,
e)::vars, exprs) tl
0476                            else hoist_body(VDecl(b, e)::vals,
vars, exprs) tl)
0477             | Expr(e)::tl -> hoist_body (vals, vars, e::exprs) tl
0478         in let vdecl_to_local senv = function
0479             | VDecl(b,e) -> to_sbind senv b
```

```
0480             | Expr(e) -> raise (Failure "hoisting failed.
vdecl_to_local should only accept VDecl")
0481         in let (vals, vars, expr) = hoist_body ([], [], [])
gn.body
0482         in let gvals = List.map (vdecl_to_local senv) vals
0483         in let gvars = List.map (vdecl_to_local senv) vars
0484         in let gn_env = List.fold_left place_formals senv
[gnformals;gvals;gvars]
0485         in let gmax_iter = (match gn.ret_expr with
0486                                 | Some x -> lookback_walk (x::expr)
0487                                 | None -> lookback_walk expr)
0488         in let body_intermediate = List.map (get_sexpr gn_env)
expr
0489         in let gbody = List.map (fun x -> (x, get_styp_from_sexpr
x)) body_intermediate
0490         in (match gn.ret_expr with
0491            | Some x -> let gret_expr = (get_sexpr gn_env x,
ret_typ)
0492               in { sgname = name; sgret_typ = ret_typ; sgmax_iter
= gmax_iter; sgformals = gnformals;
0493                    sglocalvals = gvals; sglocalvars = gvars;
sgbody = gbody;
0494                    sgret_expr = Some gret_expr }
0495            | None ->  { sgname = name; sgret_typ = ret_typ;
sgmax_iter = gmax_iter; sgformals = gnformals;
0496                    sglocalvals = gvals; sglocalvars = gvars;
sgbody = gbody;
0497                    sgret_expr = None })
0498
0499     and translate_fndecls senv sfn_decls = function
0500         | [] -> List.rev sfn_decls
0501         | hd::tl -> let translate_decl senv f =
0502                     (match f.fn_typ with
0503                         | Kn -> SKnDecl (translate_kn_decl senv f)
0504                         | Gn -> SGnDecl (translate_gn_decl senv f))
0505           in let new_fn = translate_decl senv hd
0506           in let new_fnmap = VarMap.add (get_sfn_name new_fn)
new_fn senv.sfn_decl
0507           in let new_senv = { variables = senv.variables; sfn_decl
= new_fnmap;
0508                               sstruct_map = senv.sstruct_map }
0509           in translate_fndecls new_senv (new_fn::sfn_decls) tl
0510
0511     let if_letdecls = function
0512         | LetDecl(b, e) -> true
0513         | _ -> false
0514
0515     let empty_senv = { variables = VarMap.empty; sfn_decl =
VarMap.empty;
0516                        sstruct_map = VarMap.empty; }
```

```
0517
0518    let translate_to_sast (ns, globals, functions) =
0519        let (sglobals, senv) = translate_letdecl empty_senv
globals in
0520        let sfn_list = translate_fndecls senv [] functions in
0521        (sglobals, sfn_list)
```

```
./shux/src/backend/codegen.ml
0001    module L = Llvm
0002    module A = Ast
0003
0004    module StringMap = Map.Make(String)
0005
0006    let translate (namespaces, globals, functions) =
0007             let context = L.global_context () in (* we only need
a global data container *)
0008             let the_module = L.create_module context "shux" (*
Container *)
0009      in the_module
0010      (*
0011      and i32_t  = L.i32_type  context
0012      and i8_t   = L.i8_type   context
0013      (* and i1_t = L.i1_type context *) (* reserved for bool *)
0014      and void_t = L.void_type context in
0015
0016      (* TODO: Other than Int all are placeholders *)
0017      let ltype_of_typ = function
0018        | A.Int -> i32_t
0019        | A.Float -> i32_t
0020        | A.String -> i32_t
0021        | A.Bool -> i32_t
0022        | A.Struct struct_name -> i32_t
0023        | A.Array  element_type -> i32_t
0024        | A.Vector count -> i32_t
0025      in
0026      let ltype_of_typ_opt = function
0027        | None -> void_t
0028        | Some y -> ltype_of_typ y in
0029
0030      let global_var_count = 0 in
0031
0032      (* Declare printf(), which later we will change from built-
in to linked extern function *)
0033      let printf_t = L.var_arg_function_type i32_t [|
L.pointer_type i8_t |] in
0034      let printf_func = L.declare_function "printf" printf_t
```

```
the_module in
0035
0036      (* Define each function, including args and ret type *)
0037      let function_decls =
0038        let function_decl map fdecl =
0039          let function_name = fdecl.A.fname
0040          (* and function_type = fdecl.A.fn_typ *)
0041          and formal_types =
0042            Array.of_list []  (* TODO: empty formals for now *) in
0043          let function_sign_type =
0044            L.function_type (ltype_of_typ_opt fdecl.A.ret_typ)
formal_types in
0045           StringMap.add function_name (L.define_function
function_name function_sign_type the_module, fdecl) map
0046        in
0047      List.fold_left function_decl StringMap.empty functions
0048      in
0049
0050      (* Fill in the body of the given function *)
0051      let build_function_body fdecl =
0052        (* Prep work *)
0053        let (the_function, _) = StringMap.find fdecl.A.fname
function_decls in
0054        let builder_global = L.builder_at_end context
(L.entry_block the_function) in
0055
0056        (*
0057        let int_format_str = L.build_global_stringptr "%d\n" "fmt"
builder_global in
0058        *)
0059        let format_str = L.build_global_stringptr "%s\n" "fmt"
builder_global in
0060
0061        (* Construct local variables, TODO later
0062          this is hard because we have mixed decl of bindings and
exprs *)
0063
0064        (* Construct expr builders, only implementing Call now,
Lit using placeholders *)
0065        let rec construct_expr builder = function
0066          | A.Lit i ->  (match i with
0067                           | A.LitInt j -> L.const_int i32_t j
0068                           | A.LitFloat j -> L.const_int i32_t 0
0069                           | A.LitBool b -> L.const_int i32_t 0
0070                           | A.LitKn l-> L.const_int i32_t 0
0071                           | A.LitVector elist -> L.const_int i32_t
0
0072                           | A.LitArray elist -> L.const_int i32_t
0
0073                           | A.LitStruct sflist -> L.const_int
```

```
i32_t 0
0074                           | A.LitStr str ->
ignore(global_var_count = global_var_count+1);
L.build_global_stringptr str ("mystring"^(string_of_int
global_var_count)) builder
0075                         )
0076           | A.Id str -> L.const_int i32_t 0
0077           | A.Binop (expr, binop, expr2) -> L.const_int i32_t 0
0078           | A.Assign (expr, expr2) -> L.const_int i32_t 0 (*
Create a local var if not string *)
0079           | A.Call (func, [expr]) -> (match func with
0080                                        | None -> L.const_int i32_t
0
0081                                        | Some y -> (match y with
0082                                                       | "print" ->
L.build_call printf_func [| format_str; (construct_expr builder expr)
|] "printf" builder
0083                                                       | _ ->
L.const_int i32_t 0
0084                                                    )
0085                                      )
0086           | A.Call (_, l) -> L.const_int i32_t 0 (* oh my gosh
horrible *)
0087           | A.Uniop (unop, expr) -> L.const_int i32_t 0
0088           | A.Cond (expr_if, expr_val, expr_else) -> L.const_int
i32_t 0
0089         in
0090
0091         (* define the terminal adder for each basic block *)
0092         let add_terminal builder f =
0093           match L.block_terminator (L.insertion_block builder)
with
0094             | Some _ -> ()
0095             | None -> ignore (f builder) in
0096
0097         (* Construct stmt builders *)
0098         let rec construct_stmt builder = function
0099         | A.VDecl (binding, expr_opt) -> builder; (* TODO:
implement variable bindings here *)
0100         | A.Expr e -> ignore (construct_expr builder e); builder
in
0101
0102         (* Build the code for each statement in the function
0103         let builder = construct_stmt builder (A.Block
fdecl.A.body) in
0104         *)
0105         let builder_global = List.fold_left construct_stmt
builder_global fdecl.A.body in
0106
0107         (* Add a dummy return in builder here, TODO: make it take
```

```
the value from fdecl.A.ret_expr*)
0108
0109        add_terminal builder_global (L.build_ret (L.const_int
(ltype_of_typ A.Int)0))
0110      in
0111      (* End of build_function_body *)
0112
0113      List.iter build_function_body functions;
0114      Llvm_analysis.assert_valid_module the_module; (* check
correctness *)
0115      the_module
0116      *)
```

```
./shux/src/backend/cast_llast.ml
0001    open Sast
0002    open Cast
0003    open Llast
0004
0005    module StringMap = Map.Make(String)
0006
0007    let cast_to_llast cast =
0008      let rec ctyp_to_lltyp = function
0009          SInt -> LLInt
0010        | SFloat -> LLDouble
0011        | SBool -> LLBool
0012        | SArray (styp, int) -> LLArray (ctyp_to_lltyp styp, int)
0013        | SStruct (name, _) -> LLStruct name
0014        | SVoid -> LLInt
0015        | SString -> LLConstString
0016        | SPtr -> assert false;
0017      in
0018
0019      let rec translate_clit = function
0020          CLitInt i -> LLLitInt i
0021        | CLitFloat f -> LLLitDouble f
0022        | CLitBool b -> LLLitBool b
0023        | CLitStr s -> LLLitString s
0024        | _ -> LLLitInt 0 (*TODO fix this*)
0025      in
0026
0027
0028      let a_decl i = "a" ^ (string_of_int i) in
0029      let t_decl i = "t" ^ (string_of_int i) in
0030      let c_decl i = "c" ^ (string_of_int i) in
0031      let then_decl i = "then" ^ (string_of_int i) and
0032          else_decl i = "else" ^ (string_of_int i) and
```

```
0033          merge_decl i = "merge" ^ (string_of_int i) in
0034
0035      let for_cond i = "for_cond" ^ (string_of_int i) and
0036          for_body i = "for_body" ^ (string_of_int i) and
0037          for_merge i = "for_merge" ^ (string_of_int i) in
0038
0039      let dud = LLRegLit (LLInt,(LLLitInt 42)) in
0040
0041      let add_inst_to_branch inst bname map =
0042        if(StringMap.mem bname map)
0043        then(let branch_inst_list = StringMap.find bname map in
0044             StringMap.add bname (inst::branch_inst_list) map)
0045        else(StringMap.add bname (inst::[]) map);
0046      in
0047
0048      let rec walk_cstmt
(a_stack,c_stack,t_stack,cnt,head,blabels,llinsts) = function
0049        | CExpr(t, e) -> prerr_string "CExpr->("; (assert
(StringMap.mem "entry" llinsts)); let ret = walk_cexpr a_stack c_stack
t_stack cnt head blabels llinsts e in prerr_string ")"; ret
0050        | CPushAnon(t, s) ->
0051           let v = a_decl cnt in
0052           prerr_string ("CPushAnon "^v^"->");
0053           let vreg = LLRegLabel (ctyp_to_lltyp t, v) in
0054           walk_cstmt ((v::a_stack),c_stack,t_stack,(cnt + 1),
(vreg::head),blabels,llinsts) s
0055
0056        | CReturn opt -> prerr_string "CReturn->";(match opt with
0057                            Some (typ, CPushAnon(t, cstmt)) when
t=typ ->
0058                            let v = a_decl cnt in
0059                            let vreg = LLRegLabel (ctyp_to_lltyp
typ, v) in
0060                            let (cnt, head, r1, blabels, llinsts)
=
0061                              walk_cstmt (
0062                                  (v::a_stack),
0063                                  c_stack,
0064                                  t_stack,
0065                                  (cnt + 1),
0066                                  (vreg::head),
0067                                  blabels,
0068                                  llinsts) cstmt
0069                            in
0070                            let llinst = LLBuildTerm
(LLBlockReturn vreg) in
0071                            let llinsts = add_inst_to_branch
llinst (List.hd blabels) llinsts in
0072                            (cnt, head, vreg, blabels, llinsts)
0073                          | Some _ -> assert false
```

```
0074                            | None ->
0075                               let llinst = LLBuildTerm
LLBlockReturnVoid and
0076                                   llreg = LLRegLit (LLInt,
(LLLitInt 77711)) in
0077                               let llinsts = add_inst_to_branch
llinst (List.hd blabels) llinsts in
0078                               (cnt, head, llreg, blabels, llinsts)
0079                           )
0080
0081         | CCond (t,sif,sthen,selse) ->
0082            (
0083            match (sif, sthen, selse) with
0084              | (CPushAnon(ti, ifstmt), CPushAnon(tt, thenstmt),
CPushAnon(te, elsestmt)) ->
0085                  let v = a_decl cnt in
0086                  let vreg = LLRegLabel (ctyp_to_lltyp ti, v) in
0087                  let (cnt, head, rif, blabels,llinsts) =
0088                    walk_cstmt ((v::a_stack), c_stack, t_stack, (cnt
+ 1), vreg::head, blabels,llinsts) ifstmt in
0089                  let thenname = then_decl cnt
0090                  and elsename = else_decl cnt and mergename =
merge_decl cnt in
0091                  let llinstbr = LLBuildTerm (LLBlockBr (vreg,
thenname, elsename)) in
0092                  let llinsts = add_inst_to_branch llinstbr (List.hd
blabels) llinsts in
0093
0094                  let v = a_decl cnt in
0095                  let vreg = LLRegLabel (ctyp_to_lltyp tt, v) in
0096                  let (cnt, head, _, blabels,llinsts) =
0097                    walk_cstmt (v::a_stack,c_stack, t_stack, (cnt +
1), vreg::head,
0098                                thenname::blabels, llinsts) thenstmt
in
0099                  let llthenjmp = LLBuildTerm (LLBlockJmp mergename)
in
0100                  let llinsts = add_inst_to_branch llthenjmp
(List.hd blabels) llinsts in
0101
0102                  let v = a_decl cnt in
0103                  let vreg = LLRegLabel (ctyp_to_lltyp te, v) in
0104                  let (cnt, head, _, blabels, llinsts) =
0105                    walk_cstmt (v::a_stack, c_stack,t_stack, (cnt +
1), vreg::head,
0106                                elsename::blabels, llinsts) elsestmt
in
0107                  let llelsejmp = LLBuildTerm (LLBlockJmp mergename)
in
0108                  let llinsts = add_inst_to_branch llelsejmp
```

```
(List.hd blabels) llinsts in
0109                (cnt, head, vreg, mergename::blabels, llinsts)
0110              | _ -> assert false
0111            )
0112         | CLoop (cmpstmt, bodystmt) ->
0113            let v = c_decl cnt in
0114            let vreg = LLRegLabel (LLInt, v) in
0115            let (cnt, head, cmpreg, blabels, llinsts) =
0116              walk_cstmt (a_stack, v::c_stack, t_stack, (cnt+1),
vreg::head, blabels, llinsts) cmpstmt in
0117            let condname = for_cond cnt and bodyname = for_body cnt
and mergename = for_merge cnt in
0118            let inst_init_entry = LLBuildAssign (vreg,(LLRegLit
(LLInt, LLLitInt 0)))
0119            and jmp_entry = LLBuildTerm (LLBlockJmp condname) in
0120            let llinsts = add_inst_to_branch inst_init_entry
(List.hd blabels) llinsts in
0121            let llinsts = add_inst_to_branch jmp_entry (List.hd
blabels) llinsts in
0122            let result_of_binop_reg = LLRegLabel (LLInt,
(v^"result")) in
0123            let eval_forcond = LLBuildBinOp (LLIBop LLLT, vreg,
cmpreg, result_of_binop_reg) in
0124            let branch_forcond = LLBuildTerm (LLBlockBr
(result_of_binop_reg, bodyname, mergename)) in
0125            let blabels = condname::blabels in
0126            let llinsts = add_inst_to_branch eval_forcond (List.hd
blabels) llinsts in
0127            let llinsts = add_inst_to_branch branch_forcond
(List.hd blabels) llinsts in
0128            let (cnt, head, _, blabels, llinsts) =
0129              walk_cstmt (a_stack, c_stack, t_stack, cnt, head,
bodyname::blabels, llinsts) bodystmt in
0130            let jmp_forbody = LLBuildTerm (LLBlockJmp condname) in
0131            let llinsts = add_inst_to_branch jmp_forbody (List.hd
blabels) llinsts in
0132            (cnt, head, dud, mergename::blabels, llinsts)
0133         | CBlock stmt_list ->
0134            prerr_string "CBlock->[";
0135            let fold_block (cnt, head, llreg, blabels, llinsts)
stmt  =
0136              walk_cstmt (a_stack, c_stack, t_stack, cnt, head,
blabels, llinsts) stmt in
0137            let ret = List.fold_left fold_block (cnt, head, dud,
blabels, llinsts) stmt_list
0138             in prerr_string "]";
0139    ret
0140         | _ -> assert false
0141
0142      and walk_cexpr a_stack c_stack t_stack cnt head blabels
```

```
llinsts= function
0143          CLit (typ, lit) -> prerr_string "CLit->"; (cnt, head,
LLRegLit (ctyp_to_lltyp typ,(translate_clit lit)), blabels,llinsts)
0144        | CId (typ, str) -> prerr_string ("CId"^str^"->"); (cnt,
head, LLRegLabel (ctyp_to_lltyp typ,str), blabels, llinsts)
0145        | CBinop (typx, expr1, op, expr2) ->
0146           let (cnt,head,e1, blabels,llinsts) = walk_cexpr a_stack
c_stack t_stack cnt head blabels llinsts expr1 in
0147           let (cnt,head,e2, blabels,llinsts) = walk_cexpr a_stack
c_stack t_stack cnt head blabels llinsts expr2 in
0148           let v = t_decl cnt in
0149           let vreg = LLRegLabel (ctyp_to_lltyp typx, v) in
0150           let (cnt, head, t_stack) = (cnt + 1, vreg::head,
vreg::t_stack) in
0151           let llinst =
0152             (match op with
0153              | CBinopInt sb -> (match sb with
0154                                   SAddi -> LLBuildBinOp (LLIop
LLAdd, e1, e2, vreg)
0155                                 | SSubi -> LLBuildBinOp (LLIop
LLSub, e1, e2, vreg)
0156                                 | SMuli -> LLBuildBinOp (LLIop
LLMul, e1, e2, vreg)
0157                                 | SDivi -> LLBuildBinOp (LLIop
LLDiv, e1, e2, vreg)
0158                                 | SMod -> LLBuildBinOp (LLIop
LLMod, e1, e2, vreg)
0159                                 | SEqi -> LLBuildBinOp (LLIBop
LLEQ, e1, e2, vreg)
0160                                 | SLti -> LLBuildBinOp (LLIBop
LLLT, e1, e2, vreg)
0161                                 | SLeqi -> LLBuildBinOp
(LLIBop LLLE, e1, e2, vreg)
0162                                 | SGti -> LLBuildBinOp (LLIBop
LLGT, e1, e2, vreg)
0163                                 | SGeqi -> LLBuildBinOp
(LLIBop LLGE, e1, e2, vreg)
0164                                 | _ -> assert false
0165                               )
0166              | CBinopFloat sb -> (match sb with
0167                                   SAddf -> LLBuildBinOp (LLIop
LLAdd, e1, e2, vreg)
0168                                 | SSubf -> LLBuildBinOp (LLIop
LLSub, e1, e2, vreg)
0169                                 | SMulf -> LLBuildBinOp (LLIop
LLMul, e1, e2, vreg)
0170                                 | SDivf -> LLBuildBinOp (LLIop
LLDiv, e1, e2, vreg)
0171                                 | SEqf -> LLBuildBinOp (LLIBop
LLEQ, e1, e2, vreg)
```

```
0172                                      | SLtf -> LLBuildBinOp (LLIBop
LLLT, e1, e2, vreg)
0173                                      | SLeqf -> LLBuildBinOp (LLIBop
LLLE, e1, e2, vreg)
0174                                      | SGtf -> LLBuildBinOp (LLIBop
LLGT, e1, e2, vreg)
0175                                      | SGeqf -> LLBuildBinOp (LLIBop
LLGE, e1, e2, vreg)
0176                                      | _ -> assert false
0177                                     )
0178              | CBinopBool sb -> (match sb with
0179                                      | SLogAnd -> LLBuildBinOp (LLIop
LLAnd, e1, e2, vreg)
0180                                      | SLogOr -> LLBuildBinOp (LLIop
LLOr, e1, e2, vreg)
0181                                    )
0182              | CBinopPtr ptr -> assert false
0183              | CBinopDud -> assert false
0184             ) in
0185           let llinsts = add_inst_to_branch llinst (List.hd
blabels) llinsts in
0186           (cnt, head, vreg, blabels, llinsts)
0187        | CAssign (typ, expr1, expr2) ->
0188           prerr_string "CAssign->";
0189
0190           let (cnt,head,e1, blabels, llinsts) = walk_cexpr
a_stack c_stack t_stack cnt head blabels llinsts expr1 in
0191
0192           let (cnt,head,e2, blabels,llinsts) = walk_cexpr a_stack
c_stack t_stack cnt head blabels llinsts expr2 in
0193           (*       ignore(List.iter prerr_string
(pretty_print_regs [e1;e2])); *)
0194           (*
0195           let zeroint = LLRegLit(LLInt, (LLLitInt 0))  (* TODO
need to store both int and doubles *)
0196           and zerofloat = LLRegLit(LLDouble, (LLLitDouble 0.))
0197           and zerobool = LLRegLit(LLBool, (LLLitBool false)) in
0198           let llinst = (match typ with
0199                           SInt -> LLBuildBinOp (LLIop LLAdd,
zeroint, e2, e1)
0200                         | SFloat -> LLBuildBinOp (LLFop LLFAdd,
zerofloat, e2, e1)
0201                         | SBool ->
0202                            let assert_bool = function
0203                                LLRegLabel (typ,str) ->
if(typ=LLBool) then(true) else(false)
0204                              | LLRegLit (typ, lit) ->
if(typ=LLBool) then(true) else(false)
0205                              | _ -> assert false
0206                           in
```

```
0207                             assert ((assert_bool e2)=true);
0208                             assert ((assert_bool e1)=true);
0209                             LLBuildBinOp (LLIop LLAdd, zerobool,
e2, e1)
0210                          | _ -> assert false
0211                         ) in
0212             *)
0213           let llinst = LLBuildAssign (e1, e2) in
0214           let llinsts = add_inst_to_branch llinst (List.hd
blabels) llinsts in
0215           (cnt, head, e1, blabels, llinsts)
0216         | CPeekAnon typ ->
0217           (match typ with
0218              SVoid -> (cnt, head, dud, blabels, llinsts)
0219            | _-> prerr_string "PeekAnon->";
0220           let anon = (match a_stack  with h::t -> h | [] ->
assert false) in
0221           (cnt, head, LLRegLabel (ctyp_to_lltyp typ, anon),
blabels, llinsts)
0222           )
0223         | CPeek2Anon typ ->
0224           (match typ with
0225              SVoid -> (cnt, head, dud, blabels, llinsts)
0226            | _-> prerr_string "PeekAnon2->";
0227                 let anon = (List.nth a_stack 1) in
0228                 (cnt, head, LLRegLabel (ctyp_to_lltyp typ,
anon), blabels, llinsts)
0229           )
0230         | CPeek3Anon typ ->
0231           (match typ with
0232              SVoid -> (cnt, head, dud, blabels, llinsts)
0233            | _-> prerr_string "PeekAnon3->";
0234                 let anon = (List.nth a_stack 2) in
0235                 (cnt, head, LLRegLabel (ctyp_to_lltyp typ,
anon), blabels, llinsts)
0236           )
0237         | CCall (frettyp, fname, fstmts) ->
0238           let fold_block (cnt, head, llreg, blabels, llinsts,
reglist) stmt  =
0239             let (cnt, head, llreg, blabels, llinsts) = walk_cstmt
(a_stack, c_stack, t_stack, cnt, head, blabels, llinsts) stmt
0240             in
0241             (cnt, head, llreg, blabels,llinsts, llreg::reglist)
0242           in
0243           let (cnt, head, _, blabels, llinsts, reglist) =
0244             List.fold_left fold_block (cnt, head, dud, blabels,
llinsts,[]) fstmts
0245           in
0246           (match frettyp with
0247            | SVoid ->
```

```
0248                (match fname with
0249                   "print" ->
0250                   let llcallinst = LLBuildPrintCall (List.nth
reglist 0) in
0251                   let llinsts = add_inst_to_branch llcallinst
(List.hd blabels) llinsts in
0252                   (cnt, head, dud, blabels, llinsts)
0253                 | _ ->
0254                    let llcallinst = LLBuildCall (fname, List.rev
reglist, None) in
0255                    let llinsts = add_inst_to_branch llcallinst
(List.hd blabels) llinsts in
0256                    (cnt, head, dud, blabels, llinsts)
0257                )
0258            | _ ->
0259              let v = t_decl cnt in
0260              let vreg = LLRegLabel ((ctyp_to_lltyp frettyp), v)
in
0261              let (cnt, head, t_stack) = (cnt+1, vreg::head,
vreg::t_stack) in
0262              let llcallinst = LLBuildCall ("kn_"^fname, List.rev
reglist, Some vreg) in
0263              let llinsts = add_inst_to_branch llcallinst (List.hd
blabels) llinsts in
0264              (cnt, head, vreg, blabels, llinsts)
0265           )
0266        | CLoopCtr ->
0267           prerr_string "CLoopCtr->";
0268           let ctr = (match c_stack  with h::t -> h | [] -> assert
false) in
0269           (cnt,head,LLRegLabel (LLInt,ctr), blabels, llinsts)
0270        | CExCall (frettyp, fname, fstmts) ->
0271           prerr_string "CExCall->";
0272           let fold_block (cnt, head, llreg, blabels, llinsts,
reglist) stmt  =
0273             let (cnt, head, llreg, blabels, llinsts) = walk_cstmt
(a_stack, c_stack, t_stack, cnt, head, blabels, llinsts) stmt
0274             in
0275             (cnt, head, llreg, blabels,llinsts, llreg::reglist)
0276           in
0277           let (cnt, head, _, blabels, llinsts, reglist) =
0278             List.fold_left fold_block (cnt, head, dud, blabels,
llinsts,[]) fstmts
0279           in
0280           (match fname with
0281                  "print" ->
0282                  let llcallinst = LLBuildPrintCall (List.nth
reglist 0) in
0283                  let llinsts = add_inst_to_branch llcallinst
(List.hd blabels) llinsts in
```

```
0284                   (cnt, head, dud, blabels, llinsts)
0285                 | _ ->
0286                    assert false
0287             )
0288           | _ -> assert false
0289       in
0290
0291       let translate_cdecl list = function
0292         | CFnDecl cfunc ->
0293            ignore(prerr_string (cfunc.cfname^"defined\n"));
0294            let fold_block (cnt, head, llreg, blabel, llinsts) stmt
=
0295              walk_cstmt ([],[] ,[], cnt, head, blabel,llinsts)
stmt in
0296            let initmap = StringMap.add "entry" [] StringMap.empty
in
0297            let blocks = List.fold_left fold_block (0, [],
LLRegDud,["entry"], initmap) cfunc.cbody in
0298            let (_,temps,_,blabels,insts) = blocks in
0299            let declare_fml_lcl (SBind (ctyp, cstr, _)) =
0300              LLRegLabel (ctyp_to_lltyp ctyp, cstr) in
0301            let formals = List.map declare_fml_lcl cfunc.cformals
in
0302            let locals = List.map declare_fml_lcl cfunc.clocals in
0303
0304            let block_list_gen key insts block_list =
0305              (match key with
0306                | "entry" -> block_list
0307                | str -> {llbname=str;llbbody= (List.rev
insts)}::block_list
0308              ) in
0309            let block_list = StringMap.fold block_list_gen insts []
in
0310
{llfname=cfunc.cfname;llfformals=formals;llflocals=(temps@locals);llfb
ody=(List.rev (StringMap.find "entry" insts));llfreturn=
0311             if(cfunc.cret_typ = SVoid) then (LLVoid) else
(ctyp_to_lltyp cfunc.cret_typ);llfblocks=block_list}::list
0312         | _ -> list
0313       in
0314
0315       let define_structs =
0316         let unbind (SBind (ctyp, _, _))=
0317           ctyp_to_lltyp ctyp
0318         in
0319         let define_struct list = function
0320           | CStructDef cstruct ->
0321              let struct_name = cstruct.ssname and
0322                  field_list = cstruct.ssfields in
0323              (struct_name, List.map unbind field_list)::list
```

```
0324           | _ -> list
0325         in
0326         List.fold_left define_struct [] cast
0327       in
0328
0329       let define_globals =
0330         let define_global list = function
0331           | CConstDecl (SBind (ctyp, cname,_), clit) ->
0332              (ctyp_to_lltyp ctyp, cname, translate_clit clit) ::
list
0333           | _ -> list
0334         in
0335         List.fold_left define_global [] cast
0336       in
0337
0338       let translate_cprogram =
0339         List.fold_left translate_cdecl [] cast
0340       in
0341       (define_structs,define_globals,translate_cprogram) (*struct
llglobal func*)
```

./shux/src/backend/cast.mli

```
0001     type ctyp = Sast.styp
0002
0003     type canon_val = ctyp
0004
0005     type cbind = Sast.sbind
0006
0007     type cbin_op =
0008       | CBinopInt of Sast.sbin_op_i
0009       | CBinopFloat of Sast.sbin_op_f
0010       | CBinopBool of Sast.sbin_op_b
0011       | CBinopPtr of Sast.sbin_op_p
0012       | CBinopDud
0013
0014     type cun_op = Sast.sun_op
0015
0016     type clit =
0017       | CLitInt of int
0018       | CLitFloat of float
0019       | CLitBool of bool
0020       | CLitStr of string
0021       | CLitArray of clit list
0022       | CLitStruct of (string * clit) list
0023       | CLitDud
0024
```

```
0025    type cexpr =
0026      | CLit of ctyp * clit
0027      | CId of ctyp * string
0028      | CLoopCtr                    (* access the counter inside a
CLoop *)
0029      | CPeekAnon of ctyp           (* access the temp value of a
CBlock *)
0030      | CPeek2Anon of ctyp           (* access the temp value of a
CBlock *)
0031      | CPeek3Anon of ctyp           (* access the temp value of a
CBlock *)
0032      | CBinop of ctyp * cexpr * cbin_op * cexpr
0033      | CAccess of ctyp * cexpr * string
0034      | CAssign of ctyp * cexpr * cexpr
0035      | CCall of ctyp * string * cstmt list
0036      | CExCall of ctyp * string * cstmt list
0037      | CUnop of ctyp * cun_op * cexpr
0038      | CExprDud
0039
0040    and cstmt =
0041      | CExpr of ctyp * cexpr
0042      | CCond of ctyp * (* if *) cstmt * (* then *) cstmt * (*
else *) cstmt
0043      | CPushAnon of ctyp * cstmt
0044    (*  ctyp tmp; { /* push tmp to AStack */
0045     *    cstmt where CPeekAnon := tmp /* peek AStack */
0046     *  }
0047     *  /* pop AStack */
0048     *)
0049      | CBlock of cstmt list
0050    (* {
0051     * cstmt
0052     * cstmt
0053     * ...
0054     * }
0055     *)
0056      | CLoop of cstmt (* int *) * cstmt
0057    (*  int cond = cexpr;
0058     *  int ctr;
0059     *  /* push (ctr, cond) to LStack */
0060     *  for (ctr = 0; ctr < cexpr; ctr++) {
0061     *    cstmt where CLoopCtr := ctr /* fst (peek LStack) */
0062     *  }
0063     *  /* pop LStack */
0064     *)
0065      | CReturn of (ctyp * cstmt) option
0066    (* return cstmt
0067     *)
0068      | CStmtDud
0069
```

```
0070    type cfn_decl = {
0071      cfname      : string;
0072      cret_typ    : ctyp;
0073      cformals    : cbind list;
0074      clocals     : cbind list;
0075      cbody       : cstmt list;
0076    }
0077
0078    type cstruct_def = Sast.sstruct_def
0079
0080    type cdecl =
0081      | CFnDecl of cfn_decl
0082      | CStructDef of cstruct_def
0083      | CConstDecl of cbind * clit
0084      | CExternDecl of Sast.sextern_decl
0085      | CDeclDud
0086
0087    type cprogram = cdecl list
```

./shux/src/backend/sast_cast.ml

```
0001    open Sast
0002    open Cast
0003
0004    module StringMap = Map.Make(String)
0005
0006
0007    let string_of_type t =
0008      let rec str s = function
0009        | SInt -> s ^ "SInt"
0010        | SFloat -> s ^ "SFloat"
0011        | SString -> s ^ "SString"
0012        | SBool -> s ^ "SBool"
0013        | SStruct(i, b) -> s ^ "SStruct " ^ i
0014        | SArray(t, Some n) -> s ^ "SArray[" ^ (string_of_int n )^
"] of " ^ str "" t
0015        | SArray(t, None) -> s ^ "SArray[] of " ^ str "" t
0016        | SPtr -> s ^ "SPtr"
0017        | SVoid -> s ^ "SVoid"
0018
0019    in str "" t
0020
0021    let die = false
0022    let war = false
0023    let bug s = raise (Failure ("[BUG]: " ^ s))
0024    (* let debug s = prerr_string ("[DEBUG]: " ^ s ^ "\n") *)
0025    let debug s = ()
```

```
0026    let db s = prerr_string (s ^ "\n")
0027    let warn d s = if war then prerr_string ("[WARN]: " ^ s ^
"\n"); if die then assert false else d
0028
0029    let warn_t d t s = if war then prerr_string ("[WARN]: " ^ s ^
" (" ^ (string_of_type t) ^ ")\n"); if die then assert false else d
0030
0031    let print_type t =
0032      if war then prerr_string ((string_of_type t) ^ "\n")
0033
0034
0035    let type_check t1 t2 s = (* default t1 *)
0036      if war then if t1=t2 then t1 else (print_type t1; print_type
t2; warn t1 s) else t1
0037
0038    let string_of_binop_int = function
0039      | SAddi -> "SAddi"
0040      | SSubi -> "SSubi"
0041      | SMuli -> "SMuli"
0042      | SDivi -> "SDivi"
0043      | SMod -> "SMod"
0044      | SExpi -> "SExpi"
0045      | SEqi -> "SEqi"
0046      | SLti -> "SLti"
0047      | SGti -> "SGti"
0048      | SNeqi -> "SNeqi"
0049      | SLeqi -> "SLeqi"
0050      | SGeqi -> "SGeqi"
0051
0052    let print_binop o =
0053      let string_of_binop = function
0054        | SBinopInt o -> string_of_binop_int o
0055        | SBinopFloat o -> "float"
0056        | SBinopBool o -> "bool"
0057        | SBinopPtr o -> "ptr"
0058        | SBinopFn o -> "fn"
0059        | SBinopGn o -> "gn"
0060      in prerr_string ((string_of_binop o) ^ "\n")
0061
0062    let map_tuple l p =
0063      let build e = (e, p)
0064      in List.map build l
0065
0066    let map_opt f = function
0067      | Some(x) -> Some(f x)
0068      | None -> None
0069
0070    let styp_of_sexpr = function
0071      | SLit(t, _) -> t
0072      | SId(t, _, _) -> t
```

```
0073      | SLookback(t, _, _) -> t
0074      | SAccess(t, _, _) -> t
0075      | SBinop(t, _, _, _) -> t
0076      | SAssign(t, _, _) -> t
0077      | SKnCall(t, _, _) -> t
0078      | SGnCall(t, _, _) -> t
0079      | SExCall(t, _, _) -> t
0080      | SLookbackDefault(t, _, _, _) -> t
0081      | SUnop(t, _, _) -> t
0082      | SCond(t, _, _, _) -> t
0083      | SLoopCtr -> SInt
0084      | SPeek2Anon t -> t
0085      | SExprDud -> bug "somebody is using Sexpr duds"
0086
0087    let sast_to_cast (let_decls, f_decls) =
0088      let prefix_x s = "extern_" ^ s    (* extern decl *)
0089      in let prefix_s s = "struct_" ^ s (* struct defn *)
0090    (*   in let prefix_l s = "let_" ^ s    (* let decl *) *)
0091      in let prefix_kn s = if s="main" then s else "kn_" ^ s    (*
kn function *)
0092      in let prefix_lambda s i = "lambda_" ^ i ^ "_" ^ s
0093      in let prefix_gn s = "gn_" ^ s    (* gn function *)
0094      in let prefix_gns s = "gns_" ^ s  (* gn struct *)
0095      in let prefix_gnx s = "gnx_" ^ s
0096      in let gnc = prefix_gnx "gnc"     (* gn execution state
counter name *)
0097      in let prefix_ref s = "ref_" ^ s  (* for arrays that return
by reference *)
0098      in let ret_ref = "ret_ref"
0099      in let gns_hash = Hashtbl.create 42
0100
0101      in let kn_to_fn kn =
0102        let walk_stmt (e, t) =
0103          let rec walk_anon sexpr styp sanon = (* this will yield
a reversed list *)
0104            let emit t v = (* set sanon register to the value of v
*)
0105    (*
0106              if t=SVoid then CExpr(t, v) else
0107    *)
0108              CExpr(t, CAssign(t, sanon, v))
0109
0110            in let push_anon t e last =
0111              (* push new sanon of type t onto stack, walk e, then
do last *)
0112              CPushAnon(t, CBlock(List.rev (last :: walk_anon e t
(CPeekAnon t))))
0113
0114            in let push_anon_nop t e =
0115              (* push new sanon of type t onto stack, walk e *)
```

```
0116               CPushAnon(t, CBlock(List.rev (walk_anon e t
(CPeekAnon t))))
0117
0118             in let rec walk_r acc rtyp rexpr =
0119               let walk_primitive xxx =
0120                 let lit t l =
0121                   let tr_lit = match l with
0122                     | SLitInt i -> CLitInt i
0123                     | SLitFloat f -> CLitFloat f
0124                     | SLitBool b -> CLitBool b
0125                     | SLitStr s -> CLitStr s
0126                     | _ -> warn_t CLitDud t "encountered
collection type literal in walk_primitive"
0127                   in let lit = CLit(t, tr_lit)
0128                   in emit t lit :: acc
0129
0130                 in let id t n =
0131                   let id = CId(t, n)
0132                   in emit t id :: acc
0133
0134                 in let walk_assign t l r =
0135                   let emit_r = CExpr(t, CAssign(t, CPeek2Anon t,
CPeekAnon t))
0136                   in push_anon t r emit_r :: acc
0137
0138                 in let walk_call t i a =
0139                   let emit_arg = CExpr(t, CAssign(t, CPeek2Anon t,
CPeekAnon t))
0140                   in let map_act (e, t) =
0141                     push_anon t e emit_arg
0142                   in let eval_call =
0143                     CCall(t, i, List.map map_act a)
0144                   in emit t eval_call :: acc
0145
0146                 in let walk_sex t i a =
0147                   let map_act (e, t) =
0148                     push_anon t e (CExpr(t, CAssign(t, CPeek2Anon
t, CPeekAnon t)))
0149                   in let eval_call =
0150                     CExCall(t, i, List.map map_act a)
0151                   in emit t eval_call :: acc
0152
0153                 in let walk_unop t o e =
0154                   let acc = walk_r acc t e (* leaves sanon
register containing result *)
0155                   in let unop = CUnop(t, o, sanon)
0156                   in emit t unop :: acc
0157
0158                 in let walk_binop t l o r =
0159                   let tr_binop = match o with
```

```
0160                    (* same type *)
0161                    | SBinopInt o -> CBinopInt o
0162                    | SBinopFloat o -> CBinopFloat o
0163                    | SBinopBool o -> CBinopBool o
0164                    (* change of type *)
0165                    | SBinopPtr o -> CBinopPtr o
0166                    | _ -> warn CBinopDud "encountered invalid
binary operator in walk_primitive"
0167
0168                  in let primitive xxx = (* operators whose temp
value don't change type *)
0169                    let eval_binop = CBinop(t, CPeek2Anon t,
tr_binop, CPeekAnon t)
0170                    in let emit_res = CExpr(t, CAssign(t,
CPeek3Anon t, eval_binop))
0171                    in let eval_r = push_anon (styp_of_sexpr r) r
emit_res
0172                    in push_anon (styp_of_sexpr l) l eval_r :: acc
0173
0174                  in let dereference xxx = (* operators whose
operands are of Array t and int *)
0175                    let eval =  (* TODO: make sure I understand
what the fuck is going on here *)
0176                      let arr_t = styp_of_sexpr l
0177                      in let ind_t = type_check (styp_of_sexpr r)
SInt
0178                        "encountered type mismatch in dereference
in walk_primitive"
0179                      in let eval_deref = CBinop(t, CPeek2Anon
arr_t, tr_binop, CPeekAnon ind_t)
0180                      in let emit_deref = CExpr(t, CAssign(t,
CPeek3Anon t, eval_deref))
0181                      in let eval_r = push_anon ind_t r emit_deref
0182                      in push_anon arr_t l eval_r
0183                    in eval :: acc
0184
0185                  in match o with
0186                    | SBinopPtr SIndex -> dereference ()
0187                    | SBinopFn _ -> warn acc "encountered
functional binop in walk_primitive"
0188                    | SBinopGn _ -> warn acc "encountered
generator binop in walk_primitive"
0189                    | _ -> primitive ()
0190
0191                in let walk_access t e s =
0192                  let st_t = styp_of_sexpr e
0193                  in let eval_access = CAccess(t, CPeekAnon st_t,
s)
0194                  in let emit_access = CExpr(t, CAssign(t,
CPeek2Anon t, eval_access))
```

```
0195                   in let eval_struct = push_anon st_t e
emit_access
0196                   in eval_struct :: acc
0197
0198                 in let walk_cond t iff the els =
0199                   let cond_t = type_check (styp_of_sexpr iff)
SBool
0200                     "non-boolean conditional expression in
walk_primitive"
0201                   in let eval_merge = CExpr(t, CAssign(t,
CPeek2Anon t, CPeekAnon t))
0202                   in let eval_iff = push_anon_nop cond_t iff
0203                   in let eval_the = push_anon t the eval_merge
0204                   in let eval_els = push_anon t els eval_merge
0205                   in let eval_cond = CCond(t, eval_iff, eval_the,
eval_els)
0206                   in eval_cond :: acc
0207
0208                 in match rexpr with
0209                   | SLit(t, l) -> lit t l
0210                   | SId(t, n, _) -> id t n (* don't care about
scope *)
0211                   | SUnop(t, o, e) -> walk_unop t o e
0212                   | SBinop(t, l, o, r) -> walk_binop t l o r
0213                   | SAccess(t, e, s) -> walk_access t e s
0214                   | SCond(t, iff, the, els) -> walk_cond t iff the
els
0215                   | SKnCall(t, i, a) -> walk_call t i a
0216                   | SAssign(t, l, r) -> walk_assign t l r (*
requires new nested walk *)
0217                   | SLoopCtr -> emit SInt CLoopCtr :: acc
0218                   | SPeek2Anon t -> emit t (CPeek2Anon t) :: acc
0219                   | SExCall(t, i, a) -> walk_sex t i a
0220
0221                   (* should never be called like this *)
0222                   | SGnCall(_, _, _) -> warn acc "encountered
naked generator call in walk_primitive"
0223                   | _ -> warn acc "encountered unexpected catch-
all in walk_primitive"
0224
0225               in let walk_array element_t element_c =
0226                 let deref = CBinopPtr SIndex
0227
0228                 in let lit t l =
0229                   let l = match l with (* unwrap to list of
expressions, emit by value *)
0230                     | SLitArray l -> l
0231                     | _ -> warn [] "encountered non-array type
literal in walk_array"
0232                   in let at = type_check rtyp t "literal type
```

```
mismatch in walk_array"
0233                   in let et = match at with
0234                     | SArray(t, _) -> type_check element_t t
"literal element type mismatch in walk_array"
0235                     | _ -> warn element_t "non_array type for
array type in walk_array"
0236                   in let assign e i =
0237                     let i = CLit(SInt, CLitInt i)
0238                     in let access =
0239                       CBinop(et, CPeek2Anon at, deref, i)
0240                     in let emit =
0241                       CExpr(et, CAssign(et, access, CPeekAnon t))
0242                     in push_anon et e emit
0243                   in let for_each (acc, i) e =
0244                     (assign e i :: acc, i + 1)
0245                   in let (eval_lit, _) =
0246                     List.fold_left for_each (acc, 0) l
0247                   in eval_lit
0248
0249                 in let id t n = (* reference *)
0250                   let id = CId(t, n)
0251                   in emit t id :: acc
0252
0253                 in let walk_assign t l r = (* reference *)
0254                   let emit_r = CExpr(t, CAssign(t, CPeek2Anon t,
CPeekAnon t))
0255                   in push_anon t r emit_r :: acc
0256
0257                 in let walk_cond t iff the els = (* reference *)
0258                   let cond_t = type_check (styp_of_sexpr iff)
SBool
0259                     "non-boolean conditional expression in
walk_array"
0260                   in let eval_merge = CExpr(t, CAssign(t,
CPeek2Anon t, CPeekAnon t))
0261                   in let eval_iff = push_anon_nop cond_t iff
0262                   in let eval_the = push_anon t the eval_merge
0263                   in let eval_els = push_anon t els eval_merge
0264                   in let eval_cond = CCond(t, eval_iff, eval_the,
eval_els)
0265                   in eval_cond :: acc
0266
0267                 in let walk_access t e s = (* reference *)
0268                   let st_t = styp_of_sexpr e
0269                   in let eval_access = CAccess(t, CPeekAnon st_t,
s)
0270                   in let emit_access = CExpr(t, CAssign(t,
CPeek2Anon t, eval_access))
0271                   in let eval_struct = push_anon st_t e
emit_access
```

```
0272                   in eval_struct :: acc
0273
0274                 in let walk_binop t l o r =
0275                   let dereference xxx = (* operators whose
operands are of Array t and int *)
0276                     let eval =  (* TODO: make sure I understand
what the fuck is going on here *)
0277                       let arr_t = styp_of_sexpr l
0278                       in let ind_t = type_check (styp_of_sexpr r)
SInt
0279                         "encountered non-SInt r-operand in
walk_array"
0280                       in let eval_deref = CBinop(t, CPeek2Anon
arr_t, deref, CPeekAnon ind_t)
0281                       in let emit_deref = CExpr(t, CAssign(t,
CPeek3Anon t, eval_deref))
0282                       in let eval_r = push_anon ind_t r emit_deref
0283                       in push_anon arr_t l eval_r
0284                     in eval :: acc
0285
0286                   in let generator xxx =
0287                     let gn_call id actuals =
0288                       let gn_name = prefix_gn id
0289                       in let gns_name = prefix_gns id
0290                       in let gns_fields = Hashtbl.find gns_hash
gns_name
0291                       in let gns_typ = SStruct(gns_name,
gns_fields)
0292
0293                       in let init_gns =
0294                         let set_field (a, at) (SBind(st, id, _)) =
0295                           let t = type_check at st
0296                             "encountered generator struct type
mismatch in walk_array"
0297                           in let get_field =
0298                             CAccess(t, CPeek2Anon gns_typ, id)
0299                           in let emit_field =
0300                             CExpr(t, CAssign(t, get_field,
CPeekAnon t))
0301                           in push_anon t a emit_field
0302                         in let rec init_fields inits gns_fields
actuals =
0303                           let (f, ft) = match gns_fields with
0304                             | [] -> assert false
0305                             | f::ft -> (f, ft)
0306                           in match actuals with
0307                             | [] -> inits
0308                             | a::at -> init_fields (set_field a
f :: inits) ft at
0309
```

```
0310                          in init_fields [] gns_fields actuals
0311                        in let eval_cnt = (* TODO: check what t is
equal to here *)
0312                          let cnt_t = type_check (styp_of_sexpr r)
SInt
0313                            "encountered non-SInt in gnc evaluation
in walk_array"
0314                          in push_anon_nop cnt_t l
0315                        in let call_loop =
0316                          let curr = CBinop(t, CPeek3Anon rtyp,
deref, CLoopCtr)
0317                          in let emit_val =
0318                            CExpr(t, CAssign(t, curr, CPeekAnon t))
0319                          in let set_ctr =
0320                            let ctr = CAccess(SInt, CPeek2Anon
gns_typ, gnc)
0321                            in CExpr(SInt, CAssign(SInt, ctr,
CLoopCtr))
0322                          in let call_gn =
0323                            push_anon t (SKnCall(t, gn_name,
[ (SPeek2Anon gns_typ, gns_typ) ])) emit_val
0324                          in CLoop(eval_cnt, CBlock [set_ctr;
call_gn])
0325                        in CPushAnon(gns_typ,
CBlock(List.rev(call_loop :: init_gns)))
0326                      in match r with
0327                      | SGnCall(gn_t, id, actuals) when
gn_t=element_t -> gn_call id actuals :: acc
0328                      | SGnCall(gn_t, id, actuals) -> warn (gn_call
id actuals :: acc)
0329                          "gn call type mismatch in walk_array"
0330                      |  _ -> warn acc "encountered non-SGnCall in
right operand of SFor"
0331
0332                    in let map xxx =
0333                      let atl = styp_of_sexpr l
0334                      in let etl = match atl with
0335                        | SArray(t, Some _) -> t
0336                        | SArray(t, None) -> warn t "left operand of
map is None array type in walk_array"
0337                        | _ -> warn SVoid "left operand of map is
not an array type in walk_array"
0338                      in let atr = t
0339                      in let (etr, kn_i, kn_c) = match r with
0340                        | SId(t, i, SKnLambda c) -> (t, i, c)
0341                        | _ -> warn (SVoid, "", []) "right operand
of map call incorrect in walk_array"
0342                      in let (etr, cnt) = match atr with
0343                        | SArray(t, Some cnt) when etr=t -> (etr,
cnt)
```

```
0344                       | _ -> warn_t (etr, 0) atr "map kernel
return type mismatch in walk_array"
0345                     in let for_each =
0346                       CExpr(SInt, CLit(SInt, CLitInt cnt))
0347                     in let curr =
0348                       CBinop(etr, CPeek3Anon atr, deref, CLoopCtr)
0349                     in let emit =
0350                       CExpr(etr, CAssign(etr, curr, CPeekAnon
etr))
0351                     in let closure =
0352                       List.map (fun (SBind(t, i, s)) -> (SId(t, i,
s), t)) kn_c
0353                     in let make_call =
0354                       SKnCall(etr, kn_i, (SPeek2Anon etl, etl) ::
closure)
0355                     in let do_map =
0356                       push_anon etr make_call emit
0357                     in let map_loop = CLoop(for_each, do_map)
0358                     in push_anon atl l map_loop :: acc
0359
0360                   in let filter xxx =
0361                     (*
0362                     let at = type_check (styp_of_sexpr l) t
0363                       "type mismatch of filtered lhs in
walk_array"
0364                     in
0365                     *)
0366                     acc
0367
0368                   in match o with
0369                     | SBinopPtr SIndex -> dereference ()
0370                     | SBinopGn SFor -> generator ()
0371                     | SBinopFn SMap -> map ()
0372                     | SBinopFn SFilter -> filter (* our worst
nightmare *) ()
0373                     | _ -> warn acc "encountered invalid binop for
walk_array"
0374
0375                 in let walk_call t i a =
0376                   let map_act (e, t) =
0377                     push_anon_nop t e
0378                   in let ret_ref =
0379                     CExpr(t, CPeekAnon t) (* just pass it in by
reference *)
0380                   in let eval_call =
0381                     CCall(t, i, ret_ref :: List.map map_act a)
0382                   in CExpr(t, eval_call) :: acc (* no need for
emit, use side effect *)
0383
0384                 in match rexpr with
```

```
0385                    | SLit(t, l) -> lit t l
0386                    | SId(t, n, _) -> id t n
0387                    | SBinop(t, l, o, r) -> walk_binop t l o r
0388                    | SAccess(t, e, s) -> walk_access t e s
0389                    | SCond(t, iff, the, els) -> walk_cond t iff the
els
0390                    | SKnCall(t, i, a) -> walk_call t i a
0391                    | SAssign(t, l, r) -> walk_assign t l r
0392                    | SPeek2Anon t -> emit t (CPeek2Anon t) :: acc
0393
0394                    (* no array type unary operators *)
0395                    | SLoopCtr -> warn acc "encountered SLoopCtr in
walk_array"
0396                    | SUnop(t, o, e) -> warn acc "encountered unop
in walk_array"
0397                    (* should never be called like this *)
0398                    | SGnCall(_, _, _) -> warn acc "encountered
naked SGnCall in walk_array"
0399                    | _ -> warn acc "encountered unexpected catch-
all expression in walk_array"
0400
0401                in let walk_struct id members =
0402                  let id t n = (* reference *)
0403                    let id = CId(t, n)
0404                    in emit t id :: acc
0405
0406                  in let lit t l =
0407                    let (id, l) = match l with
0408                      | SLitStruct(id, l) -> (id, l)
0409                      | _ -> warn ("", []) "encountered non-struct
type literal in walk_struct"
0410                    in let map_struct acc (f, e) =
0411                      let t = styp_of_sexpr e
0412                      in let access =
0413                        CAccess(t, CPeek2Anon rtyp, f)
0414                      in let emit =
0415                        CExpr(t, CAssign(t, access, CPeekAnon t))
0416                      in push_anon t e emit :: acc
0417                    in List.fold_left map_struct acc l
0418
0419                  in let walk_call t i a =
0420                    let map_act (e, t) =
0421                      push_anon_nop t e
0422                    in let ret_ref =
0423                      CExpr(t, CPeekAnon t) (* just pass it in by
reference *)
0424                    in let eval_call =
0425                      CCall(t, i, ret_ref :: List.map map_act a)
0426                    in CExpr(t, eval_call) :: acc (* no need for
emit, use side effect *)
```

```
0427
0428                in match rexpr with
0429                  | SLit(t, l) -> lit t l
0430                  | SId(t, n, _) -> id t n
0431                  | SKnCall(t, i, a) -> walk_call t i a
0432                  | SPeek2Anon t -> emit t (CPeek2Anon t) :: acc
0433                  | _ -> warn acc "encountered unexpected catch-
all expression in walk_struct"
0434
0435              in let walk_ptr xxx = match rexpr with
0436                | SId(t, i, _) -> emit (type_check t rtyp
"walk_ptr type mismatch") (CId(t, i)) :: acc
0437                | _ -> warn acc "encountered non SId for SPtr
type"
0438
0439              in match rtyp with
0440                | SArray(t, n) -> debug "walk_r on array type";
walk_array t n
0441                | SStruct(i, b) -> debug "walk_r on struct type";
walk_struct (prefix_s i) b
0442                | SPtr ->  walk_ptr ()
0443                | _ -> debug "walk_r on primitive type";
walk_primitive ()
0444
0445            in let walk_l ltyp lexpr =
0446              let rec lvalue_tr typ ass anon =
0447                 let primitive_assign xxx =
0448                  let assign_to e = CExpr(typ, CAssign(typ, e,
CPeek2Anon typ))
0449                  in let nop e = e
0450                  in let access t f e = CAccess(t, e, f)
0451                  in let index t l = CBinop(t, l, CBinopPtr
SIndex, CPeekAnon SInt)
0452                  in let get_index r = walk_r [] SInt r
0453
0454                  in let rec do_assign f = function
0455                    | SId(t, n, _) -> [ assign_to (f(CId(t, n))) ]
0456                    | SPeek2Anon t -> [ assign_to (f(CPeek3Anon
t)) ] (* need to look beyond the Push *)
0457                    | SAccess(t, e, i) -> do_assign (access t i) e
0458                    | SBinop(t, l, SBinopPtr SIndex, r) ->
do_assign (index t) l @ get_index r
0459                    | _ -> warn [ CStmtDud ] "encountered non-
lvalue in lvalue_tr"
0460                  in let stmts = List.map (do_assign nop) ass
0461                  in let stmts = List.map List.rev stmts
0462                  in let stmts = List.flatten stmts
0463                  in CPushAnon(SInt, CBlock stmts) (* push some
space for potential index *)
0464
```

```
0465                in let array_assign t n =
0466                  let index t a = CBinop(t, anon, CBinopPtr
SIndex, CLoopCtr)
0467                  in let get_anon = index t anon
0468                  in let get_cond = CExpr(SInt, CLit(SInt,
(CLitInt n)))
0469                  in let map_ass a =
0470                     SBinop(t, a, SBinopPtr SIndex, SLoopCtr)
0471                  in let get_index =
0472                    List.map map_ass ass
0473                  in CLoop(get_cond, lvalue_tr t get_index
get_anon)
0474
0475                in let struct_assign id binds =
0476                  let map_binds (SBind(t, n, _)) =
0477                    let map_ass a =
0478                      SAccess(t, a, n)
0479                    in let access_ass =
0480                      List.map map_ass ass
0481                    in let access_anon =
0482                      CAccess(t, anon, n)
0483                    in (t, access_ass, access_anon)
0484                  in let for_each_field =
0485                    List.map map_binds binds
0486                  in let translate (t, ass, anon) =
0487                    lvalue_tr t ass anon
0488                  in let translate_each_field =
0489                    List.map translate for_each_field
0490                  in CBlock translate_each_field
0491
0492                in match typ with
0493                  | SArray(t, Some n) -> array_assign t n
0494    (*               | SArray(_, None) -> warn CStmtDud
"encountered None-size array type in lvalue_tr" *)
0495                  | SStruct(i, b) -> struct_assign (prefix_s i) b
0496                  | SPtr -> warn CStmtDud "encountered pointer
type in lvalue_tr"
0497                  | SVoid -> if ass=[] then CBlock [] else
0498                    warn CStmtDud "encountered assignment to void
type in lvalue_tr"
0499                  | _ -> if ass=[] then CBlock[] else
primitive_assign ()
0500
0501              in let rec walk ass = function
0502                | SAssign(t, l, r) when t=ltyp -> walk (l :: ass)
r
0503                | SAssign(_, _, _) -> warn [] "encountered
assignment type mismatch in lvalue_tr"
0504                | e -> lvalue_tr ltyp ass sanon :: walk_r [] ltyp
e (* reversed *)
```

```
0505              in walk [] lexpr
0506
0507            in walk_l styp sexpr
0508          in CPushAnon(t, CBlock(List.rev (walk_anon e t
(CPeekAnon t)))) (* in order *)
0509
0510        in let walk_ret = function
0511          | Some (e, t) -> CReturn (Some (t, (walk_stmt (e, t))))
0512          | None -> CReturn None
0513
0514        in let fn_decl kn = CFnDecl
0515          { cfname = prefix_kn kn.skname; cret_typ = kn.skret_typ;
0516            cformals = kn.skformals; clocals = kn.sklocals;
0517            cbody = List.rev (walk_ret kn.skret_expr ::
List.rev_map walk_stmt kn.skbody) }
0518
0519        in let rec hoist_lambdas kn =
0520          let hoist n { slret_typ; slformals; sllocals; slbody;
slret_expr; slinherit } =
0521            hoist_lambdas
0522            { skname = prefix_lambda kn.skname n; skret_typ =
slret_typ;
0523              skformals = slformals @ slinherit; sklocals =
sllocals; skbody = slbody;
0524              skret_expr = slret_expr }
0525
0526          in let rec fish acc p = function
0527            | SLit(_, SLitKn(l)) -> (hoist p l) :: acc
0528            | SBinop(_, l, _, r) -> fish [] (p ^ "l") l @ fish acc
(p ^ "r") r
0529            | SAssign(_, l, r) -> fish [] (p ^ "l") l @ fish acc
(p ^ "r") r
0530            | SCond(_, i, t, e) -> fish [] (p ^ "i") i @ fish []
(p ^ "t") t @ fish acc (p ^ "e") e
0531            | SUnop(_, _, e) -> fish acc (p ^ "u") e
0532            | SKnCall(_, _, a) -> (List.concat (List.map (fish []
(p ^ "k")) (List.map fst a))) @ acc
0533            | SGnCall(_, _, a) -> (List.concat (List.map (fish []
(p ^ "g")) (List.map fst a))) @ acc
0534            | SAccess(_, e, _) -> fish acc (p ^ "a") e
0535            | _ -> acc
0536          in let rec bait p = function
0537            | SLit(t, SLitKn(l)) -> SId(t, prefix_lambda kn.skname
p, SKnLambda l.slinherit)
0538            | SBinop(t, l, o, r) -> SBinop(t, bait (p ^ "l") l, o,
bait (p ^ "r") r)
0539            | SAssign(t, l, r) -> SAssign(t, bait (p ^ "l") l,
bait (p ^ "r") r)
0540            | SCond(ty, i, t, e) -> SCond(ty, bait (p ^ "i") i,
bait (p ^ "t") t, bait (p ^ "e") e)
```

```
0541             | SUnop(t, o, e) -> SUnop(t, o, bait (p ^ "u") e)
0542             | SKnCall(t, s, a) -> SKnCall(t, s, List.map (fun (a,
t) -> (bait (p ^ "k") a, t)) a)
0543             | SGnCall(t, s, a) -> SGnCall(t, s, List.map (fun (a,
t) -> (bait (p ^ "g") a, t)) a)
0544             | SAccess(t, e, s) -> SAccess(t, bait (p ^ "a") e, s)
0545             | s -> s
0546
0547           in let walk_stmt (lambdas, body, p) (e, t) =
0548             (fish lambdas p e, ((bait p e), t) :: body, p ^ "x")
0549           in let (lambdas, body, _) = List.fold_left walk_stmt
([], [], "z") kn.skbody
0550           in let body = List.rev body
0551           in List.rev (fn_decl {kn with skbody = body} ::
List.concat lambdas)
0552         in hoist_lambdas kn
0553
0554       in let walk_kn kn =
0555         let kn = { kn with skname = kn.skname }
0556
0557         in let ret_id t = SId(t, ret_ref, SLocalVar)
0558         in let ret_bind t = SBind(t, ret_ref, SLocalVar)
0559         in let tr t id s = match s with
0560           | SLocalVal | SLocalVar -> (t, prefix_ref id, s)
0561           | _ -> (t, id, s)
0562         in let walk_binds (SBind(t, id, s)) =
0563           let (t, id, s) = tr t id s in SBind(t, id, s)
0564         in let walk_body (e, t) =
0565           let rec walk = function
0566             | SId(t, id, s) -> let (t, id, s) = tr t id s in
SId(t, id, s)
0567             | SBinop(t, l, o, r) -> SBinop(t, walk l, o, walk r)
0568             | SAssign(t, l, r) -> SAssign(t, walk l, walk r)
0569             | SCond(t, iff, the, els) -> SCond(t, walk iff, walk
the, walk els)
0570             | SUnop(t, o, e) -> SUnop(t, o, walk e)
0571             | SAccess(t, e, s) -> SAccess(t, walk e, s)
0572             | SKnCall(t, s, a) -> SKnCall(t, s, List.map (fun (e,
t) -> (walk e, t)) a)
0573             | SGnCall(t, s, a) -> SGnCall(t, s, List.map (fun (e,
t) -> (walk e, t)) a)
0574             | s -> s
0575           in (walk e, t)
0576         in let walk_ret ret =
0577           let assign_ret (e, t) = (SAssign(t, ret_id t, e), t)
0578           in match ret with
0579             | Some(e, t) -> Some(assign_ret(walk_body (e, t)))
0580             | None -> warn None "encountered None return expr in
reference-returning kn in walk_kn"
0581         in let ref_kn xxx = { kn with skbody = List.map walk_body
```

```
kn.skbody;
0582                            skformals = ret_bind kn.skret_typ ::
List.map walk_binds kn.skformals;
0583                            sklocals = List.map walk_binds
kn.sklocals;
0584                            skret_expr = walk_ret kn.skret_expr }
0585         in let kn = match kn.skret_typ with
0586           | SArray(_, Some _) -> ref_kn ()
0587           | SArray(t, None) -> warn kn "encountered kn that
returns None sized array type in walk_kn"
0588           | SStruct(_, _) -> ref_kn ()
0589           | _ -> kn
0590
0591           (*
0592         in let local_bindings =
0593           let fold_map acc (SBind(t, i, s)) = StringMap.add i (t,
s) acc
0594           in List.fold_left fold_map StringMap.empty (kn.skformals
@ kn.sklocals)
0595     *)
0596         in let kn_infer xxx =
0597           let infer_array (e, t) =
0598             let co l r =
0599               let r_none xxx = match r with
0600                 | SArray(t, Some n) -> r
0601                 | SArray(t, None) -> bug "L R array type both
None"
0602                 | _ -> assert false
0603               in let r_some ln = match r with
0604                 | SArray(t, Some n) when ln=n -> l
0605                 | SArray(t, Some n) -> bug "L R array Some size
value mismatch"
0606                 | SArray(t, None) -> l
0607                 | _ -> assert false
0608               in match l with
0609                 | SArray(t, Some n) -> r_some n
0610                 | SArray(t, None) -> r_none ()
0611                 | _ -> assert false
0612
0613             in let rec walk_r l_typ = function
0614               | SLit(t, l) -> let it = co l_typ t in (SLit(it, l),
it)
0615               | SId(t, i, s) -> let it = co l_typ t in (SId(it, i,
s), it)
0616               | SKnCall(t, id, a) -> let it = co l_typ t in
(SKnCall(it, id, a), it)
0617               | SGnCall(t, id, a) -> let it = co l_typ t in
(SGnCall(it, id, a), it)
0618               | SExCall(t, id, a) -> let it = co l_typ t in
(SExCall(it, id, a), it)
```

```
0619              | SPeek2Anon t -> let it = co l_typ t in (SPeek2Anon
it, it)
0620    (*          | SBinop(t, l, SBinopGn o, r) -> let (l, it) =
walk_r l_typ l in (SBinop(it, l, SBinopGn o, r), it) *)
0621              | SBinop(t, l, SBinopFn o, r) -> let (l, it) =
walk_r l_typ l in (SBinop(it, l, SBinopFn o, r), it)
0622    (*          | SBinop(t, l, SBinopPtr o, r) -> let (l, it) =
walk_r l_typ l in (SBinop(it, l, SBinopPtr o, r), it) *)
0623              | e -> let t = co l_typ (styp_of_sexpr e) in (e, t)
0624
0625            in let coerce l lr_typ =
0626              let ll_typ = styp_of_sexpr l
0627              in let (t, rn) = match lr_typ with
0628                | SArray(t, Some n) -> (t, n)
0629                | SArray(t, None) -> (t, 0)
0630                | _ -> assert false
0631              in let n = match ll_typ with
0632                | SArray(t, Some n) -> if n<rn then Some n else if
rn=0 then None else Some rn
0633                | SArray(t, None) -> if rn=0 then None else Some
rn
0634                | _ -> assert false
0635              in SArray(t, n)
0636            in let rec walk_l l_typ = function
0637              | SAssign(t, l, r) -> SAssign(t, l, walk_l (coerce l
l_typ) r)
0638              | e -> let (e, it) = walk_r l_typ e in e
0639            in match t with
0640              | SArray(_) -> (walk_l t e, t)
0641              | _ -> (e, t)
0642
0643          in { kn with skbody = List.map infer_array kn.skbody;
0644               skret_expr = map_opt infer_array kn.skret_expr }
0645        in kn_to_fn (kn_infer ())
0646
0647      in let walk_gn gn =
0648        let prefix_gnv s = "gnv_" ^ s         (* for local vars *)
0649        in let gns_arg = prefix_gnx "arg"           (* gn
execution state argument name *)
0650
0651        in let st_fields =
0652          let a_decl = function
0653            | SBind(t, n, SLocalVal) -> SBind(SArray(t, Some
gn.sgmax_iter), n, SStructField)
0654            | SBind(t, n, s)-> warn (SBind(t, n, s)) "encountered
non-SLocalVal binding in gn formals or local vals in walk_gn"
0655          in let ctr_decl =
0656            SBind(SInt, gnc, SLocalVar)
0657          in ctr_decl :: List.map a_decl (gn.sgformals @
gn.sglocalvals)
```

```
0658
0659         in let gns_typ_name = prefix_gns gn.sgname
0660         in let gns_typ = SStruct(gns_typ_name, st_fields) (*
struct type name *)
0661         in let defn_cstruct =
0662           Hashtbl.add gns_hash gns_typ_name st_fields;
0663           CStructDef { ssname = gns_typ_name; ssfields =
st_fields }
0664
0665         in let gn_to_kn =
0666           let st_id = SId(gns_typ, gns_arg, SLocalVar)
0667           in let st_element t id = SAccess(t, st_id, id)
0668           in let st_var t = st_element (SArray(t, Some
gn.sgmax_iter))
0669           in let st_cnt = st_element SInt gnc
0670           in let wrap_int n = SLit(SInt, SLitInt n)
0671
0672           in let prefix_var = function
0673             | SBind(t, n, SLocalVar) -> SBind(t, prefix_gnv n,
SLocalVar)
0674             | SBind(t, n, s)-> warn (SBind(t, n, s)) "encountered
non-SLocalVar in gn local vars in walk_gn"
0675
0676           in let lb_st t id n =
0677             (* should be gnx_arg.id[(gnx_ctr - n) % max_iter] *)
0678             let idx = SBinop(SInt, st_cnt, SBinopInt SSubi,
wrap_int n)
0679             in let idx = SBinop(SInt, idx, SBinopInt SMod,
wrap_int gn.sgmax_iter)
0680             in SBinop(t, st_var t id, SBinopPtr SIndex, idx)
0681
0682           in let lb_cmp n = SBinop(SBool, wrap_int n, SBinopInt
SLeqi, st_cnt)
0683
0684           in let rec lookback (e, t) =
0685             let sid t id = function
0686               | SGlobal as s -> SId(t, id, s) (* global prefixing
will happen in walk_kn *)
0687               | SKnLambda _ as s -> SId(t, id, s)
0688               | SLocalVar as s -> SId(t, prefix_gnv id, s)
0689               | SLocalVal -> lb_st t id 0
0690               | SStructField as s -> warn (SId(t, id, s))
0691                 "encountered SStructField binding scope in
walk_gn"
0692
0693             in let rec lb = function
0694               | SId(t, id, s) -> sid t id s
0695               | SLookback(t, id, n) -> lb_st t id n
0696               | SLookbackDefault(t, n, f, e) -> SCond(t, lb_cmp n,
lb f, lb e)
```

```
0697              | SAccess(t, e, id) -> SAccess(t, lb e, id)
0698              | SBinop(t, l, o, r) -> SBinop(t, lb l, o, lb r)
0699              | SAssign(t, l, r) -> SAssign(t, lb l, lb r)
0700              | SKnCall(t, id, a) -> SKnCall(t, id, List.map
lookback a)
0701              | SGnCall(t, id, a) -> SGnCall(t, id, List.map
lookback a)
0702              | SUnop(t, o, e) -> SUnop(t, o, lb e)
0703              | SCond(t, i, f, e) -> SCond(t, lb i, lb f, lb e)
0704              | e -> e
0705            in (lb e, t)
0706          in { skname = prefix_gn gn.sgname; skret_typ =
gn.sgret_typ;
0707               skformals = [ SBind(gns_typ, gns_arg,
SLocalVar) ];
0708               sklocals = List.map prefix_var gn.sglocalvars;
0709               skbody = List.map lookback gn.sgbody;
0710               skret_expr = map_opt lookback gn.sgret_expr }
0711
0712        in defn_cstruct :: kn_to_fn gn_to_kn
0713
0714      in let walk_fns f_decls =
0715        let rec walk = function
0716          | [] -> []
0717          | SGnDecl(g)::t -> let r = walk_gn g in r @ walk t
0718          | SKnDecl(k)::t -> let r = walk_kn k in r @ walk t
0719          | SExDud(_)::t -> warn (walk t) "came across SEx booty
call juicy"
0720        in walk f_decls
0721    (*   in let let_map = Hashtbl.create 42 *)
0722      in let walk_static let_decls =
0723        (*
0724        let interp_expr t e =
0725          let interp_lit = function
0726            | SLitInt i -> CLitInt i
0727            | SLitFloat f -> CLitFloat f
0728            | SLitBool b -> CLitBool b
0729            | SLitStr s -> CLitStr s
0730            | SLitArray l -> assert false
0731            | SLitStruct(id, l) -> assert false
0732            | _ -> assert false
0733          in let interp_primitive xxx = match e with
0734            | SLit(t, l) -> interp_lit l
0735            | SId(t, i, s) -> CLitDud
0736            | SBinop(t, l, o, r) -> assert false
0737            | _ -> assert false
0738          in let interp_array at n =
0739            CLitDud
0740          in let interp_struct id binds =
0741            CLitDud
```

```
0742          in match t with
0743            | SArray(t, Some n) -> interp_array t n
0744            | SArray(t, None) -> warn CLitDud "None size array
encountered in let decls"
0745            | SStruct(i, b) -> interp_struct i b
0746            | SPtr | SVoid -> warn CLitDud "invalid type
encountered in let declarations"
0747            | _ -> interp_primitive ()
0748        in let assign_let n t e =
0749          let let_val = interp_expr t e
0750          in let _ = Hashtbl.add let_map n let_val
0751          in CLitDud
0752          *)
0753        let walk = function
0754    (*       | SLetDecl(SBind(t, n, s), e) -> CConstDecl(SBind(t,
n, s), interp_expr t e) *)
0755          | SLetDecl(SBind(t, n, s), e) -> CConstDecl(SBind(t, n,
s), CLitDud)
0756          | SStructDef s -> CStructDef {s with ssname = prefix_s
s.ssname}
0757          | SExternDecl x -> CExternDecl {x with sxalias =
prefix_x x.sxalias}
0758        in walk let_decls
0759
0760      (* function entry point: walk entire program *)
0761      in let walk_program l f =
0762        let r = List.map walk_static l in r @ walk_fns f
0763      in walk_program let_decls f_decls
```

./shux/src/backend/semant.ml

```
0001    open Ast
0002    open Astprint
0003    open Sast
0004
0005    (* map variable to type
0006             or stack of types if overridden *)
0007    module VarMap = Map.Make(struct
0008            type t = string
0009            let compare x y = Pervasives.compare x y
0010                    end)
0011
0012    (* Set of var names *)
0013    module VarSet = Set.Make(struct
0014          type t = string
0015          let compare x y = Pervasives.compare x y
0016                    end)
```

```
0017
0018    module StringMap = Map.Make(String)
0019
0020    type struct_type = {
0021      struct_id : string;
0022      fields : (string * Ast.typ) list;
0023    }
0024
0025    type var = {
0026             id : string;
0027      mut : Ast.mut;
0028             var_type : Ast.typ;
0029             initialized : bool;
0030    }
0031
0032    type trans_env = {
0033                     (*  a stack of variables mapped to a name *)
0034        scope : var list VarMap.t;
0035
0036        (* list of structs defined in the block *)
0037        structs : struct_type VarMap.t;
0038
0039        (* list of already defined kernels/generators *)
0040        fn_map : fn_decl VarMap.t;
0041
0042                     (* new variables defined within a scope
0043           to ensure that name conflicts dont happen
0044           within a scope  *)
0045                     new_variables : var list;
0046
0047        lookbacks : var VarMap.t;
0048    }
0049
0050    let noop_func = { fname = "nop"; fn_typ = Kn; ret_typ = None;
0051                      formals = []; body = []; ret_expr = None; }
0052
0053    let flatten_ns_list ns_list =
0054        let rec flatten_ns_rec flat = function
0055            | [] -> flat
0056            | hd::tl -> flatten_ns_rec (flat ^ "_" ^ hd) tl
0057        in flatten_ns_rec (List.hd ns_list) (List.tl ns_list)
0058
0059    let get_sfield_name = function
0060     | StructField(id, _ ) -> id
0061
0062    let get_bind_typ = function
0063       | Bind(_,t,_) -> (match t with
0064           | Struct(str_list) -> Struct([flatten_ns_list
str_list])
0065           | _ -> t)
```

```
0066
0067    let get_bind_name = function
0068       | Bind(_,_,s) -> s
0069
0070    let get_bind_mut = function
0071       | Bind(m,_,_) -> m
0072
0073    let convert_ret_typ = function
0074       | Some x -> x
0075       | None -> Void
0076
0077    let compare_ast_typ l r = match(l,r) with
0078       | (Array(t1, _), Array(t2, _)) -> t1=t2 (* this is a weak
way to type check arrays *)
0079       | (l,r) -> l=r                          (* but we're just
going to Let It Happpen 😭 *)
0080
0081    (* ensure that arrays are initialized properly when declared
*)
0082    let check_array_init = function
0083        | Array(t,i) -> (match i with
0084            | Some i -> true
0085            | None -> raise (Failure "Arrays need to have sizes to
be initialized."))
0086        | _ -> true
0087
0088    (* ensure that a variable isnt part of new_variables when its
defined *)
0089    let check_var_notdef var env =
0090        let check_var v b new_var =
0091           if b then
0092              if v.id = new_var.id then false else b
0093           else false in
0094        List.fold_left (check_var var) true env.new_variables
0095
0096    (* called within the assign expression to initialize variable
in env *)
0097    let initialize_var name env =
0098        let var_list = VarMap.find name env.scope
0099        in let var = List.hd var_list
0100        in let new_var = { id = var.id; mut=var.mut; var_type =
var.var_type;
0101                           initialized = true; }
0102        in let new_scope = VarMap.add name (new_var :: List.tl
var_list) env.scope
0103        in { scope = new_scope; structs = env.structs; fn_map =
env.fn_map;
0104             new_variables = env.new_variables; lookbacks =
env.lookbacks; }
0105
```

```
0106    (* return new trans env with var v added *)
0107    let push_variable_env v env =
0108        if not (check_var_notdef v env)
0109            then let err_msg = v.id ^ " defined more than once in
scope." in
0110                     raise (Failure err_msg)
0111        else let new_scope =
0112            if VarMap.mem v.id env.scope then
0113                let oldvarlist = VarMap.find v.id env.scope
0114                in VarMap.add v.id (v::oldvarlist) env.scope
0115            else
0116                VarMap.add v.id [v] env.scope
0117        in { scope = new_scope; structs = env.structs; fn_map =
env.fn_map;
0118             new_variables = v::env.new_variables; lookbacks =
env.lookbacks; }
0119
0120    (* check expression
0121    tr_env: current translation environment
0122    expr: expression to be checked
0123
0124    returns the type of the expression *)
0125    let rec type_of_lit tr_env = function
0126       | LitInt(l) -> Int
0127       | LitFloat(l) -> Float
0128       | LitStr(l) -> String
0129       | LitBool(l) -> Bool
0130       | LitStruct(id, l) ->
0131        let nid = flatten_ns_list id in
0132        if VarMap.mem nid tr_env.structs then
0133            let s = VarMap.find nid tr_env.structs in
0134            if List.length s.fields != List.length l
0135            then let err_msg = "Wrong number of fields in struct "
0136                    ^ nid ^ ". Expected: " ^ string_of_int
(List.length s.fields) ^
0137                    " Got: " ^ string_of_int (List.length l) in
raise(Failure err_msg)
0138            else
0139                let match_lits = match_sfields tr_env l in
0140                let match_fields b sfield = b && match_lits sfield
in
0141                    if (List.fold_left match_fields true s.fields)
then
0142                        Struct([nid])
0143                    else
0144                        let err_msg = "Struct literal doesn't
match struct defined" ^
0145                                        "as" ^ nid in raise(Failure
err_msg)
0146        else let err_msg = "Struct " ^ nid ^ " is not defined"
```

```
0147                            in raise (Failure err_msg)
0148
0149       | LitVector(l) -> let vector_check v =
0150                              if check_expr tr_env v = Float then
true
0151                              else raise (Failure "Vector literals
need to consist entirely of floats")
0152                            in ignore(List.map vector_check l);
Vector(List.length l)
0153       | LitArray(l) ->
0154          let arr_length = List.length l in
0155                               let rec array_check arr typ =
0156            if (arr = []) then Array(typ, Some (arr_length)) else
0157            let nxt_typ = check_expr tr_env (List.hd arr) in
0158            if nxt_typ != typ then raise (Failure ("Array types
not consistent."))
0159            else array_check (List.tl arr) (check_expr tr_env
(List.hd arr)) in
0160          array_check (List.tl l) (check_expr tr_env (List.hd l))
0161       | LitKn(l) -> lambda_checker l tr_env
0162
0163    and check_expr tr_env expr =
0164             match expr with
0165              | Lit(a) -> type_of_lit tr_env a
0166       | Id(nvar) -> let var = flatten_ns_list nvar in
0167          if VarMap.mem var tr_env.scope then
0168              let found_var = List.hd (VarMap.find var
tr_env.scope)
0169              in if found_var.initialized
0170                  then found_var.var_type
0171              else
0172                  raise (Failure ("Variable " ^ var ^ " has not
been initialized"))
0173                                 else if VarMap.mem var tr_env.fn_map
then
0174                  let _ = print_string ("Function pointer to " ^
var ^ "detected. Be careful out there yo.")
0175                  in Ptr
0176          else raise (Failure ("Variable " ^ var ^ " has not been
declared"))
0177       | Binop(e1, op, e2) ->
0178          let t1 = check_expr tr_env e1 in
0179          (match op with
0180           | Add | Sub | Mul | Div -> if t1 != check_expr tr_env
e2
0181              then raise (Failure "Can't do binop on incompatible
types.")
0182              else if t1 != Int && (check_expr tr_env e2) != Float
then
0183                 raise (Failure "Binop only defined over integers
```

```
or scalars.")
0184              else t1
0185            | Mod -> if t1 = Int && (check_expr tr_env e2) = Int
then Int else
0186               raise (Failure "Bad types for mod operator")
0187            | Exp -> if t1 = Float && (check_expr tr_env e2) =
Float then Float else
0188
0189               raise (Failure "Bad types for exponent operator")
0190            | Eq | Lt | Gt | Neq | Leq | Geq -> if t1 !=
(check_expr tr_env e2)
0191               then raise (Failure "Can't do binop on incompatible
types.") else Bool
0192            | LogAnd | LogOr -> if t1 != (check_expr tr_env e2) ||
t1 != Bool
0193               then raise (Failure "Logical and/or only applies to
bools.") else Bool
0194            | Filter | Map as fm -> let (t,i) = (match t1 with
0195                  | Array(v, i) -> (v, i)
0196                  | _ -> raise (Failure "Left hand needs to be []
for map/filter"))
0197            in
0198                  if (match e2 with
0199                      | Lit(n) -> (match n with
0200                                   | LitKn(l) ->
0201                                             (List.length l.lformals) =
1 &&
0202                                             ((get_bind_typ (List.hd
l.lformals)) = t)
0203                                   | _ -> raise (Failure "Filter/Map
right hand literals needs to be a lambda :("))
0204                      | Id(nn) -> let n =  flatten_ns_list nn in
0205                                   if VarMap.mem n tr_env.fn_map then
0206                                      let k = VarMap.find n
tr_env.fn_map in
0207                                      if k.fn_typ = Kn
0208                                          then List.length k.formals = 1
0209                                          else raise (Failure "Map/
Filter function needs to be a kernel")
0210                                 else raise (Failure ("Kernel call in
filter/map " ^ n ^ "doesnt" ^ " exist"))
0211                      | _ -> raise (Failure "Filter not given a
lambda."))
0212                  then (match fm with
0213                   | Filter -> let filt_typ = (match e2 with
0214                                    | Id(nn) -> let n =
flatten_ns_list nn in
0215                                                let kn = VarMap.find
n tr_env.fn_map
0216                                                in convert_ret_typ
```

```
kn.ret_typ
0217                                | _ -> (check_expr tr_env e2))
0218                          in if filt_typ = Bool then Array(t, i)
else
0219                          raise (Failure "Filter kernel needs to
return Bool")
0220                  | Map -> let map_typ = (match e2 with
0221                                | Id(nn) -> let n =
flatten_ns_list nn
0222                                            in let kn =
VarMap.find n tr_env.fn_map
0223                                            in convert_ret_typ
kn.ret_typ
0224                                | _ -> (check_expr tr_env e2)) in
0225                           Array(map_typ, i)
0226                  | _ -> raise (Failure "The OCaml compiler has a
strict type system."))
0227                else raise (Failure ("Map/Filter needs kernel that
takes single"
0228                ^"parameter matching the [] to be mapped/
filtered"))
0229            | Index -> if (check_expr tr_env e2) = Int then match
t1 with
0230               | Array(v, i) -> v
0231               | Vector(l) -> Float
0232               | _ -> raise (Failure "Indexing needs an array/
vector to index into")
0233              else raise (Failure "Indexing needs to be by integer
expression only.")
0234           | For -> (match e2 with
0235                        | Call(str, elist) -> (match str with
0236                            | Some ns -> let s = flatten_ns_list ns
in
0237                                if VarMap.mem s tr_env.fn_map then
0238                                let gn = VarMap.find s
tr_env.fn_map
0239                                and tlist = List.map (check_expr
tr_env) elist
0240                                in if (gn.fn_typ = Kn) then raise
(Failure ("Cannot call " ^
0241                                                          "kernel
with a for expression"))
0242                                else let match_formal b fform cform
=
0243                                   if b then (compare_ast_typ
(get_bind_typ fform) cform) else b
0244                                in
0245                                if (List.fold_left2 match_formal
true gn.formals tlist)
0246                                then match gn.ret_typ with
```

```
0247                                    | Some x -> Array(x, None)
0248                                    | None -> Void
0249                                else raise (Failure ("Formals don't
match for generator call " ^ s))
0250                                else raise (Failure ("Generator " ^
s ^ " not defined in call"))
0251                            | None -> Int)
0252                            | _ -> raise (Failure "For iterator
needs a generator call "))
0253          | Do ->  (match e2 with
0254                        | Call(str, elist) -> (match str with
0255                            | Some ns -> let s = flatten_ns_list ns
in
0256                                if VarMap.mem s tr_env.fn_map then
0257                                let gn = VarMap.find s
tr_env.fn_map
0258                                and tlist = List.map (check_expr
tr_env) elist
0259                                in if (gn.fn_typ = Kn) then raise
(Failure ("Cannot call " ^
0260                                                            "kernel
with a do expression"))
0261                                else let match_formal b fform cform
=
0262                                    if b then (compare_ast_typ
(get_bind_typ fform) cform) else b
0263                                in
0264                                if (List.fold_left2 match_formal
true gn.formals tlist)
0265                                then match gn.ret_typ with
0266                                    | Some x -> x
0267                                    | None -> Void
0268                                else raise (Failure ("Formals don't
match for generator call " ^ s))
0269                                else raise (Failure ("Generator " ^
s ^ " not defined in call"))
0270                            | None -> Int)
0271                            | _ -> raise (Failure "For iterator
needs a generator call ")))
0272       | Assign(e1, e2) ->
0273                (* need to short circuit the initialization
checker here *)
0274            let match_typ exp1 exp2 =
0275                let t1 = (match exp1 with
0276                    | Id(l) -> let flat_name = flatten_ns_list l
0277                                in if VarMap.mem flat_name
tr_env.scope
0278                                then let var = List.hd (VarMap.find
flat_name tr_env.scope)
0279                                in var.var_type
```

```
0280                                else raise ( Failure ("Variable " ^
flat_name ^ " is not defined"))
0281                    | _ -> check_expr tr_env exp1) and
0282                    t2 = check_expr tr_env exp2 in
0283                if t1 = t2 then t1
0284                else raise (Failure "Assignment types don't match.
shux can't
0285                autocast types") in
0286             let get_mutability l =
0287                    let v = List.hd (VarMap.find l tr_env.scope)
in
0288                    if v.mut = Mutable then v.var_type
0289                    else if v.initialized = false then v.var_type
0290                    else raise (Failure "Cannot assign to
immutable type")
0291             in
0292       (match e1 with
0293           | Id l -> ignore (match_typ e1 e2); get_mutability
(flatten_ns_list l)
0294           | Assign(l1,l2) -> match_typ e1 e2
0295           | Access(x,y)-> (match x with
0296               | Id(l) -> ignore(get_mutability (flatten_ns_list
l)); match_typ e1 e2;
0297               | _ -> raise(Failure "Semant not implemented for
accessing into non-ids"))
0298           | Binop(idx1, Index, _) -> (match idx1 with
0299               | Id(l) -> ignore (get_mutability (flatten_ns_list
l))
0300               | _ -> raise (Failure "Semant not implemented for
indexing into non-ids"));
0301           match_typ e1 e2
0302           | _ -> raise (Failure "Assign can only be done against
an id or struct field")
0303       )
0304
0305       | Call(str, elist) -> (match(str) with
0306           | Some ns -> let s = flatten_ns_list ns in
0307                        if VarMap.mem s tr_env.fn_map then
0308                          let fn = VarMap.find s tr_env.fn_map and
0309                          tlist = List.map (check_expr tr_env)
elist
0310                          in
0311                          if(fn.fn_typ = Gn) then raise (Failure
("Cannot call " ^
0312                              "generator without an iterator"))
else
0313                          let match_formal b fform cform =
0314                                 if b then get_bind_typ fform =
cform
0315                                 else b
```

```
0316                                in
0317                                if (List.fold_left2 match_formal true
fn.formals tlist)
0318                                then match(fn.ret_typ) with
0319                                    | Some x -> x
0320                                    | None -> Void
0321                                else let err_msg = "Formals dont
match for function" ^
0322                                       " call " ^ s in raise
(Failure err_msg)
0323               else let err_msg = "Kernel " ^ s ^ " is not defined
in call." in
0324               raise (Failure err_msg)
0325            | None -> Int)
0326
0327        | Uniop(unop, e) -> (match unop with
0328           | Pos | Neg -> let t = check_expr tr_env e in if t = Int
|| t = Float then t else
0329              raise (Failure "Pos/Neg unioperators only valid for
ints or floats")
0330           | LogNot -> if check_expr tr_env e = Bool then Bool else
0331              raise (Failure "Logical not only applies to
booleans"))
0332        | LookbackDefault(e1, e2) ->
0333               let t1 = check_expr tr_env e1
0334               and t2 = check_expr tr_env e2
0335               in if t1 = t2 then t1
0336               else raise (Failure ("Lookback variable has type " ^
_string_of_typ t1 ^
0337                                    " but lookback default returns
" ^ _string_of_typ t2))
0338        | Cond(e1, e2, e3) -> if check_expr tr_env e1 = Bool then
0339             let t2 = check_expr tr_env e2
0340             and t3 = check_expr tr_env e3
0341             in if (t2 = t3) then t2
0342             else raise (Failure "Ternary operator return type
mismatch")
0343          else raise (Failure "Ternary operator conditional needs
to be a boolean
0344          expr")
0345        | Lookback(nstr, i) -> let str = flatten_ns_list nstr in
0346               if VarMap.mem str tr_env.lookbacks then
0347                   let found_var = (VarMap.find str
tr_env.lookbacks)
0348                   in if found_var.mut = Immutable then
found_var.var_type
0349                      else raise (Failure "Lookback not allowed for
mutable types")
0350               else raise (Failure ("Name " ^ str ^ " is not
defined in lookback expression."))
```

```
0351       | Access(id, str) -> let fname = (match id with
0352                                         | Id(l) -> flatten_ns_list l
(* for namespace *)
0353                                         | _ -> "")
0354              in
0355             (* check structs *)
0356              let t1 = check_expr tr_env id in
0357              (match t1 with
0358                 | Struct(nid) -> let id = flatten_ns_list nid in
0359                                  if VarMap.mem id tr_env.structs then
0360                                  let s = VarMap.find id tr_env.structs
in
0361                                  let find_field tuple field =
0362                                          if (fst tuple)="NONE" then
0363                                             if (fst field)=str
0364                                                then (fname, snd field)
else tuple
0365                                          else tuple in
0366                                          let matched = List.fold_left
find_field ("NONE", Int)
0367                                s.fields in
0368                                if (fst matched)="NONE" then
0369                                        let err_msg = "Struct field
named " ^ str ^ " is "  ^
0370                                       "not defined." in raise
(Failure err_msg)
0371                                 else (snd matched)
0372                       else let err_msg = "Struct named " ^ id ^ "
not defined." in
0373                                  raise (Failure err_msg)
0374                  | _ -> raise (Failure "Can't access field of a
type that's not a
0375                                    struct"))
0376
0377    (* function checker for lambdas *)
0378    (* returns the type of lambda *)
0379    (* basically a duplication of the code for check_body *)
0380    and lambda_checker l env =
0381        let formal_var = List.hd l.lformals
0382        in let formal_name = get_bind_name formal_var
0383        in let formal_type = get_bind_typ formal_var
0384        in let m = Immutable
0385        in let v = { id = formal_name; var_type = formal_type; mut
= m; initialized = true }
0386        in let formal_env = push_variable_env v env
0387        in let body = l.lbody
0388        in let ret = l.lret_expr
0389        in
0390        let check_stmt env = function
0391            | VDecl(b,e) -> (match e with
```

```
0392                | Some e -> let t1 = check_expr env e
0393                            and t2 = get_bind_typ b
0394                            and var_name = get_bind_name b
0395                            and m = get_bind_mut b in
0396                (* let _ = check_array_init t2 in *)
0397                if compare_ast_typ t2 t1 then
0398                    let v = {id = var_name; var_type = t2; mut =
m; initialized = true }
0399                            in push_variable_env v env
0400                   else raise (Failure "Type mismatch in lambda
body")
0401                | None  -> let t = get_bind_typ b
0402                           and var_name = get_bind_name b
0403                           and m = get_bind_mut b
0404                       in let _ = check_array_init t
0405                       in let v = { id = var_name; var_type = t;
mut = m; initialized = true }
0406                       in push_variable_env v env)
0407            | Expr(e) -> let _ = check_expr env e in
0408                (match e with
0409                | Assign(e1,e2) -> (match e1 with
0410                    | Id(l) -> initialize_var (flatten_ns_list l)
env
0411                    | _ -> env)
0412                | _ -> env)
0413         in (match ret with
0414            | Some r -> check_expr (List.fold_left check_stmt
formal_env body) r
0415            | None -> Void)
0416
0417
0418    and get_sfield_typ tr_env = function
0419     | StructField(_, expr) -> check_expr tr_env expr
0420
0421    (* type checking helper for structs *)
0422    and
0423    match_sfields env lit struct_field =
0424       let get_typ = get_sfield_typ env in
0425       let match_lit b lit strct =
0426          if b then
0427             if get_sfield_name lit = (fst struct_field) then
0428                 let err_msg = "Struct field " ^ (fst
struct_field) ^
0429                     " takes place more than once in struct
literal." in
0430                     raise (Failure err_msg)
0431             else b
0432          else
0433             if get_sfield_name lit = (fst struct_field) then
0434                                         if get_typ lit = (snd
```

```
struct_field) then
                                                                  true
                                                               else let
err_msg = "Type mismatch in struct literal: " ^ _string_of_typ
(get_typ lit)
                                          ^ " doesn't match " ^
_string_of_typ (snd struct_field)
                                            in raise (Failure err_msg)
                                     else
                  false
        in List.fold_left (fun b x -> match_lit b x struct_field)
false lit


    let fltn_global nsname globs =
            let handle_glob nsname = function
                    | LetDecl(Bind(m,t,n),e) ->
                                          let newn = nsname ^ "_" ^ n
in LetDecl(Bind(m,t,newn),e)
                    | StructDef s -> StructDef( { sname = nsname
^ "_" ^ s.sname;
                                          fields = s.fields} )
                    | ExternDecl e -> ExternDecl( { xalias =
nsname ^ "_" ^ e.xalias;
                                           xfname = e.xfname;
                                           xret_typ = e.xret_typ;
                                           xformals = e.xformals; } )

            in List.map (fun x -> handle_glob nsname x) globs

    let fltn_fn nsname fndecls =
            let rec handle_function nsname fndecl =
                    { fname = nsname ^ "_" ^ fndecl.fname;
                            fn_typ = fndecl.fn_typ;
                            ret_typ = fndecl.ret_typ;
                            formals = fndecl.formals;
                            body = fndecl.body;
                            ret_expr = fndecl.ret_expr; } in
            List.map (fun x -> handle_function nsname x) fndecls

    let rec flatten_ns ns_list =
            let rec handle_ns ns =
                    match ns.nbody with
                    | ([], glob, fn) -> (fltn_global ns.nname
glob, fltn_fn ns.nname fn)
                    | (nestns, glob, fn) -> let flat_ns =
flatten_ns nestns in
                             (fltn_global ns.nname (glob @ (fst
flat_ns)),
                              fltn_fn ns.nname (fn @ (snd
```

```
flat_ns)))
0474             in List.fold_left (fun (a,b) (an, bn) -> (a@an,
b@bn)) ([], [])
0475             (List.map handle_ns ns_list)
0476
0477    let check_globals g =
0478       let rec check_global_inner tr_env = function
0479            | [] -> tr_env
0480            | hd::tl -> (match hd with
0481               | LetDecl(bind, expr) -> (match bind with
0482                  | Bind(mut, typ, s) ->
0483                   let t2 = check_expr tr_env expr in
0484                   if compare_ast_typ typ t2 then
0485                           let v = { id = s; var_type = typ;
mut=mut; initialized = true} in
0486                           let vlist = if VarMap.mem s tr_env.scope
then
0487                                    v :: VarMap.find s tr_env.scope
else [v] in
0488                           check_global_inner { scope = VarMap.add
s vlist tr_env.scope;
0489                             structs = tr_env.structs;
0490                             fn_map = tr_env.fn_map;
0491                             new_variables = [];
0492                             lookbacks = tr_env.lookbacks; } tl
0493                  else let err_msg = "Type mismatch in let decl: "
^ _string_of_typ typ ^
0494                                     " " ^ s ^ " is assigned to "
^ _string_of_typ t2
0495                                     in raise(Failure  err_msg))
0496               | StructDef(s) -> if VarMap.mem s.sname
tr_env.structs
0497                                 then let err_msg = "Struct with
name " ^ s.sname ^
0498                                 " defined more than once." in
raise(Failure err_msg)
0499                                 else
0500                                 let check_unique m bind =
0501                                     let name = get_bind_name bind
in
0502                                     if StringMap.mem name m then
0503                                         let err_msg = "Field name
" ^ name ^ " defined more than once in struct "
0504                                                             ^ s.sname
in raise (Failure err_msg)
0505                                     else StringMap.add name name
m
0506                                 in ignore(List.fold_left
check_unique StringMap.empty s.fields);
0507                      let map_fields = function
```

```
0508                        | Bind(m, t, id) -> if VarMap.mem (s.sname
^ "_" ^ id) tr_env.scope then
0509                                                let err_msg =
"Namespace/struct field contention for struct "
0510                                                                ^
s.sname ^ " and field " ^ id in raise(Failure err_msg)
0511                                               else
0512                                               (match t with
0513                                                   | Array(typ, i) ->
(match i with
0514                                                       | Some i ->
(id,t)
0515                                                       | None -> let
err_msg = "Array " ^ id ^ " initialized with no fixed "
0516
^ "size in struct " ^ id
0517
in raise (Failure err_msg))
0518                                                   | _ -> (id,t))
0519                     in let nfields = List.map map_fields s.fields
in
0520                     let st = {struct_id = s.sname; fields =
nfields} in
0521                     check_global_inner { scope = tr_env.scope;
0522                                          structs = VarMap.add
s.sname st tr_env.structs;
0523                                          fn_map = tr_env.fn_map;
0524                                          new_variables = [];
0525                                          lookbacks =
tr_env.lookbacks; } tl
0526                | ExternDecl(e) ->
0527                     let f = { fname = e.xalias; fn_typ = Kn;
0528                               ret_typ = e.xret_typ;
formals=e.xformals;
0529                               body = []; ret_expr = None } in
0530                     let ntr_env =
0531                           { scope = tr_env.scope; structs =
tr_env.structs;
0532                             fn_map = VarMap.add f.fname f
tr_env.fn_map;
0533                             new_variables = [];
0534                             lookbacks = tr_env.lookbacks;} in
0535                     check_global_inner ntr_env tl)
0536
0537       in
0538       let env_default = { scope = VarMap.empty; structs =
VarMap.empty;
0539                           fn_map = VarMap.empty; new_variables =
[];
0540                           lookbacks = VarMap.empty;  } in
```

```
0541       check_global_inner env_default g
0542
0543    let get_lookback_env fn_typ formals body env =
0544        let rec grab_exprs decls exprs = function
0545            | [] ->  (decls, exprs)
0546            | hd::tl -> (match hd with
0547                | VDecl(b,e) -> (match e with
0548                    | Some e -> grab_exprs (b::decls) (e::exprs)
tl
0549                    | None -> grab_exprs (b::decls) exprs tl)
0550                | Expr(e) -> grab_exprs decls (e::exprs) tl)
0551
0552        in let rec kn_lookback = function
0553            | [] ->  true
0554            | hd::tl -> (match hd with
0555                | Binop(e1,binop,e2) -> if kn_lookback [e1;e2]
then kn_lookback tl
0556                                          else false
0557                | Assign(e1,e2) -> if (kn_lookback [e1;e2])
0558                                         then kn_lookback tl else
false
0559                | Call(s, elist) -> if kn_lookback elist then
kn_lookback tl else false
0560                | Uniop(_,e) -> if kn_lookback [e] then
kn_lookback tl else false
0561                | Cond(e1,e2,e3) -> if kn_lookback[e1;e2;e3] then
kn_lookback tl
0562                                                                 else
false
0563                | Access(e,_) -> if kn_lookback[e] then
kn_lookback tl else false
0564                | Lookback(s,i) -> false
0565                | LookbackDefault(e1,e2) -> false
0566                | _ -> kn_lookback tl)
0567
0568       in let rec gn_rec_lookback = function
0569               | Lookback(slist, i) ->
0570                   [flatten_ns_list slist]
0571               | Assign(e1,e2) ->
0572                   let lb1 = gn_rec_lookback e1
0573                   and lb2 = gn_rec_lookback e2 in lb1@lb2
0574               | Binop(e1,_,e2) ->
0575                   let lb1 = gn_rec_lookback e1
0576                   and lb2 = gn_rec_lookback e2 in lb1@lb2
0577               | Call(_, elist) ->
0578                   let lb_list = List.map gn_rec_lookback elist
0579                   in List.flatten lb_list
0580               | Uniop(_,e) -> gn_rec_lookback e
0581               | LookbackDefault(e1,e2) ->
0582                   let lb1 = gn_rec_lookback e1
```

```
0583                    and lb2 = gn_rec_lookback e2 in lb1@lb2
0584                | Cond(e1,e2,e3) ->
0585                    let lb1 = gn_rec_lookback e1
0586                    and lb2 = gn_rec_lookback e2
0587                    and lb3 = gn_rec_lookback e3 in lb1@lb2@lb3
0588                | Access(e, _) -> gn_rec_lookback e
0589                | _ -> []
0590
0591        in let rec get_lookback_bindings map bindings = function
0592            | [] -> bindings
0593            | Bind(m,t,name)::tl -> if StringMap.mem name map
0594                then let var = { id = name;  mut = m; var_type = t;
0595                                 initialized = false; }
0596                in get_lookback_bindings map (var::bindings) tl
0597                else get_lookback_bindings map bindings tl
0598
0599        in let rec add_lookbacks senv = function
0600            | [] -> senv
0601            | hd::tl ->
0602              let new_lbs = VarMap.add hd.id hd senv.lookbacks
0603              in let new_env = { scope = senv.scope; structs =
senv.structs;
0604                                 fn_map = senv.fn_map;
new_variables = senv.new_variables;
0605                                  lookbacks = new_lbs; }
0606              in add_lookbacks new_env tl
0607
0608        in let rec get_lb_names lbexprs = function
0609            | [] -> lbexprs
0610            | hd::tl -> get_lb_names ((gn_rec_lookback hd)@lbexprs)
tl
0611
0612        in let (decls, exprs) = grab_exprs [] [] body
0613        in let lb_names = (get_lb_names [] exprs)
0614        in let rec mapify map = function
0615            | [] -> map
0616            | hd::tl -> if StringMap.mem hd map
0617                            then mapify map tl
0618                            else mapify (StringMap.add hd hd map)
tl
0619        in (match fn_typ with
0620          | Kn -> if kn_lookback exprs then env
0621                                       else raise (Failure "Kn
can't have lookbacks")
0622          | Gn -> let (decls, exprs) = grab_exprs [] [] body
0623                  in let lb_names = get_lb_names [] exprs
0624                  in let smap = mapify StringMap.empty lb_names
0625                  in let bindings = get_lookback_bindings smap []
decls
0626                  in add_lookbacks env bindings)
```

```
0627
0628    let check_body f env =
0629        let check_formals formals env =
0630            let check_formal env old_formals formal =
0631                if List.mem formal old_formals then
0632                    let err_msg = "Formal " ^ get_bind_name formal
^ " has been"
0633                                  ^ " defined more than once." in
raise (Failure err_msg)
0634                else formal  :: old_formals
0635            in List.fold_left (check_formal env) [] formals and
0636            place_formal env formal =
0637               let formal_name = get_bind_name formal and
0638                   formal_type = get_bind_typ formal and
0639                   m = Immutable
0640               in let v = { id = formal_name; var_type =
formal_type; mut = m; initialized=true }
0641               in push_variable_env v env
0642        in let formal_env =
0643           List.fold_left place_formal env (check_formals
f.formals env)
0644        in let lookback_env = get_lookback_env f.fn_typ f.formals
f.body formal_env
0645        in let body = f.body and
0646            ret = f.ret_expr
0647        in
0648        let check_stmt env = function
0649            | VDecl(b,e) -> (match e with
0650               | Some exp -> let t1 = check_expr env exp and
0651                                  t2 = get_bind_typ b and
0652                                  var_name = get_bind_name b
0653                                  and m = get_bind_mut b in
0654                   (* ensure that arrays cannot be initialized
without sizes *)
0655                   (* let _ = check_array_init t2 in  *)
0656                   if compare_ast_typ t2 t1 then
0657                       let v = { id = var_name; var_type = t2; mut
= m; initialized = true}
0658                       in push_variable_env v env
0659                   else let err_msg = "Type " ^ _string_of_typ t1
^ " cannot be assigned"
0660                                      ^ " to type " ^
_string_of_typ t2
0661                       in raise(Failure err_msg)
0662               | None -> let t = get_bind_typ b and
0663                            (* ensure that arrays cannot be
initialized without sizes *)
0664                            var_name = get_bind_name b and
0665                            m = get_bind_mut b
0666               in let _ = check_array_init t
```

```
0667                in let v = { id = var_name; var_type = t; mut = m;
initialized = false}
0668                in push_variable_env v env )
0669             | Expr(e) -> let _ = check_expr env e in
0670                 (match e with
0671                 | Assign(e1, e2) -> (match e1 with
0672                     | Id(l) -> initialize_var (flatten_ns_list l)
env
0673                     | _ -> env)
0674                 | _ -> env)
0675           in let ret_typ = convert_ret_typ f.ret_typ
0676           and tr = (match ret with
0677              | Some r -> check_expr (List.fold_left check_stmt
lookback_env body) r
0678              | None -> Void)
0679           in if (tr = ret_typ) then
0680                       { scope = env.scope; structs = env.structs;
0681                         fn_map = VarMap.add f.fname f env.fn_map;
0682                         new_variables = env.new_variables;
0683                         lookbacks = env.lookbacks; }
0684              else
0685                       let err_msg  = "Function " ^ f.fname ^ "
has type "
0686                       ^ _string_of_typ ret_typ ^
0687                       " but returns type " ^ _string_of_typ tr
0688                       in raise (Failure err_msg)
0689
0690    (* checks if main is defined *)
0691    let check_main functions =
0692             let check_if_main b f =
0693                     if b then b
0694             else if f.fname = "main" && f.fn_typ = Kn then true
else b in
0695         List.fold_left check_if_main false functions
0696
0697    (* main type checking goes on here *)
0698    let check_functions functions run_env =
0699       if check_main functions then
0700           let check_function tr_env f =
0701              if VarMap.mem f.fname tr_env.fn_map then
0702                 let err_msg = "Function " ^ f.fname ^ " is
defined more than once" in
0703                 raise(Failure err_msg)
0704              else if VarMap.mem f.fname tr_env.scope then
0705                 let err_msg = "Function " ^ f.fname ^ " name
conflicts with global"
0706                 ^ " variable" in raise(Failure err_msg)
0707              else check_body f tr_env
0708           in List.fold_left check_function run_env functions
0709        else raise (Failure "Main kn not defined")
```

```
0710
0711    (* entry point *)
0712    let check (ns, globals, functions) =
0713            let flat_ns = flatten_ns ns in
0714      let globs_with_ns = (fst flat_ns) @ globals in
0715            let global_env = check_globals globs_with_ns in
0716            let nfunctions = noop_func::functions
0717      in ignore (check_functions ((snd flat_ns) @ nfunctions)
global_env);
0718            ([], fst flat_ns @ globals, snd flat_ns @ nfunctions)
```

```
./shux/src/backend/sast.mli
0001    type sscope = (* replacing Mutable vs Immutable *)
0002      | SLocalVal
0003      | SLocalVar
0004      | SGlobal
0005      | SStructField
0006      | SKnLambda of sbind list
0007
0008    and styp =
0009      | SInt
0010      | SFloat
0011      | SString
0012      | SBool
0013      | SStruct of string * sbind list
0014      | SArray of styp * int option
0015      | SPtr
0016      | SVoid
0017
0018    and sbind = SBind of styp * string * sscope
0019
0020    type sbin_op_i =
0021      | SAddi | SSubi | SMuli | SDivi | SMod | SExpi
0022      | SEqi | SLti | SGti | SNeqi | SLeqi | SGeqi
0023
0024    type sbin_op_f =
0025      | SAddf | SSubf | SMulf | SDivf | SExpf
0026      | SEqf | SLtf | SGtf | SNeqf | SLeqf | SGeqf
0027
0028    type sbin_op_b =
0029      | SLogAnd | SLogOr
0030
0031    type sbin_op_p =
0032      | SIndex
0033
0034    type sbin_op_fn =
```

```
0035      | SFilter | SMap
0036
0037    type sbin_op_gn =
0038      | SFor
0039
0040    type sbin_op =
0041      | SBinopInt of sbin_op_i
0042      | SBinopFloat of sbin_op_f
0043      | SBinopBool of sbin_op_b
0044      | SBinopPtr of sbin_op_p
0045      | SBinopFn of sbin_op_fn
0046      | SBinopGn of sbin_op_gn
0047
0048    type sun_op =
0049      | SLogNot | SNegi | SNegf
0050
0051    type slit =
0052      | SLitInt of int
0053      | SLitFloat of float
0054      | SLitBool of bool
0055      | SLitStr of string
0056      | SLitKn of slambda
0057      | SLitArray of sexpr list
0058      | SLitStruct of string * ((string * sexpr) list)
0059
0060    and sexpr =
0061      | SLit of styp * slit
0062      | SId of styp * string * sscope
0063      | SLookback of styp * string * int
0064      | SAccess of styp * sexpr * string
0065      | SBinop of styp * sexpr * sbin_op * sexpr
0066      | SAssign of styp * sexpr * sexpr
0067      | SKnCall of styp * string * (sexpr * styp) list
0068      | SGnCall of styp * string * (sexpr * styp) list
0069           | SExCall of styp * string * (sexpr * styp) list
0070      | SLookbackDefault of styp * int * sexpr * sexpr
0071      | SUnop of styp * sun_op * sexpr
0072      | SCond of styp * sexpr * sexpr * sexpr
0073      | SLoopCtr (* CLoopCtr, useful for recursion *)
0074      | SPeek2Anon of styp
0075      | SExprDud
0076
0077    and slambda = {
0078      slret_typ   : styp;
0079      slformals   : sbind list;
0080      slinherit   : sbind list;
0081      sllocals    : sbind list;         (* no lookback, const-ness
not enforced *)
0082      slbody      : (sexpr * styp) list;
0083      slret_expr  : (sexpr * styp) option;
```

```
0084    }
0085
0086    type skn_decl = {
0087      skname      : string;
0088      skret_typ   : styp;
0089      skformals   : sbind list;
0090      sklocals    : sbind list;         (* do not have lookback *)
0091      skbody      : (sexpr * styp) list;
0092      skret_expr  : (sexpr * styp) option;
0093    }
0094
0095    type sgn_decl = {
0096      sgname      : string;
0097      sgret_typ   : styp;
0098      sgmax_iter  : int;
0099      sgformals   : sbind list;
0100      sglocalvals : sbind list;
0101      sglocalvars : sbind list;
0102      sgbody      : (sexpr * styp) list;
0103      sgret_expr  : (sexpr * styp) option;
0104    }
0105
0106    type sfn_decl =
0107      | SGnDecl of sgn_decl
0108      | SKnDecl of skn_decl
0109      | SExDud of skn_decl
0110
0111    type sstruct_def = {
0112      ssname      : string;
0113      ssfields    : sbind list;
0114    }
0115
0116    type sextern_decl = {
0117             sxalias                  : string;
0118      sxfname     : string;
0119      sxret_typ   : styp;
0120      sxformals   : sbind list;
0121    }
0122
0123    type slet_decl =
0124      | SLetDecl of sbind * sexpr
0125      | SStructDef of sstruct_def
0126      | SExternDecl of sextern_decl
0127
0128    and sprogram = slet_decl list * sfn_decl list
```

./shux/src/backend/llast.mli
```
0001    type lltyp =
0002      | LLBool (* i1 *)
0003      | LLInt (* i32 *)
0004      | LLDouble (* double_type *)
0005      | LLConstString (* name and content, only used for
representing strings, simply i8* *)
0006      | LLArray of lltyp * int option (* inside formal we need to
declare int*; inside local we declare int[len] *)
0007      | LLStruct of string
0008      | LLVoid (* only used for declaring function types *)
0009
0010    type lllit =
0011      | LLLitBool of bool
0012      | LLLitInt of int
0013      | LLLitDouble of float
0014      | LLLitString of string
0015      | LLLitArray of lllit list
0016      | LLLitStruct of lllit list
0017
0018    type llreg =
0019      | LLRegLabel of lltyp * string(* register can store a name
and an lltyp value *)
0020      | LLRegLit of lltyp * lllit
0021      | LLRegDud
0022
0023    type llops_iop =
0024      | LLAdd
0025      | LLSub
0026      | LLMul
0027      | LLDiv
0028      | LLMod
0029      | LLAnd
0030      | LLOr
0031
0032    type llops_fop =
0033      | LLFAdd
0034      | LLFSub
0035      | LLFMul
0036      | LLFDiv
0037
0038    type llops_ibop =
0039      | LLLT
0040      | LLEQ
0041      | LLGT
0042      | LLLE
0043      | LLGE
0044
0045    type llops_fbop =
0046      | LLFLT
```

```
0047      | LLFEQ
0048      | LLFGT
0049      | LLFGE
0050      | LLFLE
0051
0052    type llops_typ =
0053      | LLIop of llops_iop
0054      | LLFop of llops_fop
0055      | LLIBop of llops_ibop
0056      | LLFBop of llops_fbop
0057
0058    type llblock_term = (* a block must be terminated by either a
jump or a return *)
0059      | LLBlockReturn of llreg
0060      | LLBlockReturnVoid
0061      | LLBlockBr of llreg * string * string (* 1st of boolean,
2nd and 3 of label type *)
0062      | LLBlockJmp of string (* register must be a branch label
type *)
0063
0064    type llstmt =
0065      | LLBuildBinOp of llops_typ * llreg * llreg * llreg
0066      | LLBuildCall of string * llreg list * llreg option(* 1
storing func def, 2 storing list of formals, 3 storing retval, void
functions dont have return val *)
0067      | LLBuildPrintCall of llreg (* register storing the integer
that you want to print *)
0068      | LLBuildArrayLoad of llreg * llreg * llreg (* 1 has the arr
label, 2 has index, 3 has dest label *)
0069      | LLBuildArrayStore of llreg * llreg * llreg (* 1 has the
arr label, 2 has index, 3 has source label *)
0070      | LLBuildStructLoad of llreg * int * llreg (* 1 has the
struct label, 2 has the index, 3 has dest label *)
0071      | LLBuildStructStore of llreg * int * llreg (* 1 has the
struct label, 2 has the index, 3 has source label *)
0072      | LLBuildTerm of llblock_term
0073      | LLBuildAssign of llreg * llreg (* assign from the right to
the left*)
0074      | LLBuildNoOp
0075
0076
0077    type llblock = {
0078        llbname : string;
0079        llbbody : llstmt list;
0080    }
0081
0082    type llfunc_def = {
0083        llfname : string;
0084        llfformals : llreg list;
0085        llflocals : llreg list;
```

```
0086        llfbody : llstmt list;
0087        llfreturn : lltyp;
0088        llfblocks : llblock list;
0089    }
0090
0091    type llglobal = lltyp * string * lllit
0092    type llstruct_def = string * lltyp list
0093
0094    type llprog = llstruct_def list * llglobal list * llfunc_def
list
0095
```

```
./shux/src/frontend/astprint.ml
0001    open Ast
0002
0003    (* Pretty printing *)
0004    let nop x = x
0005
0006    (* option helpers *)
0007    let is_some = function
0008      | Some _ -> true
0009      | None -> false
0010
0011    let string_of_opt_default d f = function
0012      | Some s -> f s
0013      | None -> d
0014
0015    let string_of_opt f s = string_of_opt_default "" f s
0016
0017    let rec string_of_id = function
0018      | [] -> ""
0019      | [s] -> s
0020      | s::l -> s ^ "->" ^ string_of_id l
0021
0022    let rec _string_of_typ = function
0023      | Int -> "int"
0024      | Float -> "float"
0025      | String -> "string"
0026      | Bool -> "bool"
0027      | Ptr -> "_ptr"
0028      | Struct t -> "struct " ^ string_of_id t
0029      | Array(t, i) -> _string_of_typ t ^ "[" ^ (match i with
Some(n) -> string_of_int n | None -> "") ^ "]"
0030      | Vector t -> "vector<" ^ string_of_int t ^ ">"
0031      | Void -> "__void__" (* should not be used *)
0032
```

```
0033    let string_of_typ x = string_of_opt _string_of_typ x
0034
0035    let string_of_fn_typ = function
0036      | Kn -> "kn"
0037      | Gn -> "gn"
0038
0039    type op_typ = Infix | Prefix | PostfixPair
0040
0041    let _fix = function
0042      | Add | Sub | Mul | Div | Mod | Exp
0043      | Eq | Lt | Gt | Neq | Leq | Geq | LogAnd | LogOr
0044      | Filter | Map -> Infix
0045      | For | Do -> Prefix
0046      | Index -> PostfixPair
0047
0048    let string_of_binop = function
0049      | Add -> " + "
0050      | Sub -> " - "
0051      | Mul -> " * "
0052      | Div -> " / "
0053      | Mod -> " %"
0054      | Exp  -> " ^ "
0055      | Eq -> " == "
0056      | Lt -> " < "
0057      | Gt -> " > "
0058      | Neq -> " != "
0059      | Leq -> " <= "
0060      | Geq -> " >= "
0061      | LogAnd -> " && "
0062      | LogOr -> " || "
0063      | Filter -> " :: "
0064      | Map -> " @ "
0065      | For -> "for "
0066      | Do -> "do "
0067      | _ -> "" (* should raise error *)
0068
0069    let string_of_binop_match = function
0070      | Index -> ("[", "]")
0071      | _ -> ("", "") (* should raise error *)
0072
0073    let string_of_binop_expr f l o r =
0074      match _fix o with
0075      | Infix -> f l ^ string_of_binop o ^ f r
0076      | Prefix -> string_of_binop o ^ f l ^ f r
0077      | PostfixPair -> match string_of_binop_match o with (o, c) -
> f l ^ o ^ f r ^ c
0078    let string_of_asn_expr f l r =
0079      f l ^ " = " ^ f r
0080
0081    let string_of_unop = function
```

```
0082      | LogNot -> "!"
0083      | Neg -> "-"
0084      | Pos -> "+"
0085
0086    let string_of_uniop_expr f o e = string_of_unop o ^ f e
0087
0088    let string_of_cond_expr f i t e =
0089      f i ^ " ? " ^ f t ^ " : " ^ f e
0090
0091    let string_of_lookback_default_expr f l e =
0092      f l ^ " : " ^ f e
0093
0094    let string_of_mut = function
0095      | Immutable -> ""
0096      | Mutable -> "var "
0097
0098    let string_of_bind = function
0099      | Bind(mut, typ, id) -> string_of_mut mut ^ _string_of_typ
typ ^ " " ^ id
0100
0101    let string_of_list f l o s c e =
0102      match (List.map f l) with
0103      | [] -> if e then o ^ c else ""
0104      | l -> o ^ String.concat s l ^ c
0105
0106    let rec string_of_struct_field = function
0107      | StructField(n, e) -> "\t." ^ n ^ " = " ^ string_of_expr e
0108
0109    and string_of_lambda = function {lformals = fs; lbody = b;
lret_expr = re } ->
0110      string_of_list string_of_bind fs "(" ", " ")" false ^
0111      string_of_list string_of_stmt b "{\n" "\n\t" "" true ^
0112      string_of_opt string_of_expr re ^ "\n}"
0113
0114    and string_of_lit = function
0115      | LitInt(l) -> string_of_int l
0116      | LitFloat(l) -> string_of_float l
0117      | LitBool(l) -> string_of_bool l
0118      | LitStr(l) -> "\"" ^ l ^ "\""
0119      | LitKn(l) -> string_of_lambda l
0120      | LitVector(l) -> string_of_list string_of_expr l "<" ", "
">" true
0121      | LitArray(l) -> string_of_list string_of_expr l "[" ", "
"]" true
0122      | LitStruct(id, l) -> string_of_id id ^ string_of_list
string_of_struct_field l "{" ";\n" "}" true
0123
0124    and string_of_expr = function
0125     | Lit l -> string_of_lit l
0126     | Id s -> string_of_id s
```

```
0127      | Lookback(s, i) -> string_of_id s ^ ".." ^ string_of_int i
0128      | Access(e, i) -> string_of_expr e ^ "." ^ i
0129      | Uniop(o, e) -> string_of_uniop_expr string_of_expr o e
0130      | Assign(l, r) -> string_of_asn_expr string_of_expr l r
0131      | Binop(e1, o, e2) -> string_of_binop_expr string_of_expr e1
o e2
0132      | Call(s, el) -> string_of_opt_default "_" string_of_id s ^
0133                       string_of_list string_of_expr el "(" ", "
")" (is_some s)
0134      | LookbackDefault(l, e) -> string_of_lookback_default_expr
string_of_expr l e
0135      | Cond(i, t, e) -> string_of_cond_expr string_of_expr i t e
0136
0137    and string_of_vdecl bind expr =
0138      string_of_bind bind ^ " " ^ string_of_expr expr
0139
0140    and string_of_stmt = function
0141      | VDecl (bind, expr) -> string_of_opt (string_of_vdecl bind)
expr ^ ";\n"
0142      | Expr(expr) -> string_of_expr expr ^ ";\n"
0143
0144    let string_of_fdecl = function { fn_typ = fn; fname = id;
formals = fs;
0145                                     ret_typ = rt; body = b;
ret_expr = re } ->
0146      string_of_fn_typ fn ^ " " ^ id ^ string_of_list
string_of_bind fs "(" ", " ")" true ^
0147      string_of_typ rt ^ string_of_list string_of_stmt b "\n{\n"
"\n\t" "" true ^
0148      "\t" ^ string_of_opt string_of_expr re ^ "\n}\n"
0149
0150    let string_of_struct_def = function { sname = n; fields = f }
->
0151      "struct " ^ n ^ string_of_list string_of_bind f "{\n\t" ";
\n\t" "}" true
0152
0153    let string_of_extern = function { xalias = a; xfname = n;
xformals = f; xret_typ = r } ->
0154      "extern " ^ a ^ string_of_list string_of_bind f "(" ", " ")"
true ^ string_of_typ r ^ ";"
0155
0156    let string_of_let = function
0157      | LetDecl(bind, expr) -> string_of_bind bind ^ " " ^
string_of_expr expr ^ ";"
0158      | StructDef(s) -> string_of_struct_def s
0159      | ExternDecl(s) -> string_of_extern s
0160
0161    let rec string_of_ns = function { nname = n; nbody = b} ->
0162      "ns " ^ n ^ " {\n" ^ string_of_program b ^ "\n}"
0163
```

```
0164    and string_of_program (ns_list, let_list, fn_list) =
0165      String.concat "\n" (List.map string_of_ns ns_list) ^ "\n" ^
0166      String.concat "\n" (List.map string_of_let let_list) ^ "\n"
^
0167      String.concat "\n" (List.map string_of_fdecl fn_list)
```

```
./shux/src/frontend/scanner.mll
0001    {
0002      open Core.Std
0003      open Parser
0004      open Exceptions
0005      let lineno = ref 1
0006      let depth = ref 0
0007      let filename = ref "" (* what do with this *)
0008
0009      let unescape s =
0010        Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)
0011    }
0012
0013    (* char class regexes *)
0014    let whitespace = [' ' '\t' '\r']
0015    let newline = '\n'
0016    let alpha = ['a'-'z' 'A'-'Z']
0017    let digit = ['0'-'9']
0018    let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
0019    let escape_char = ''' (escape) '''
0020    let ascii = ([' '-'!' '#'-'[' ']'-'~'])
0021
0022    (* type regexes *)
0023    let string = '"' ( (ascii | escape)* as s) '"'
0024    (* we don't support chars
0025    let char = ''' ( ascii | digit ) '''
0026    *)
0027    let float = (digit+) ['.'] digit+
0028    let int = digit+
0029
0030    let id = alpha (alpha | digit | '_')*
0031
0032    rule token = parse
0033      | whitespace  { token lexbuf }
0034      | newline     { incr lineno; token lexbuf }
0035      | "/*"        { incr depth; comment lexbuf }
0036
0037    (* parens *)
0038      | '('     { LPAREN }
0039      | ')'     { RPAREN }
```

```
0040      | '{'     { LBRACE }
0041      | '}'     { RBRACE }
0042      | '['     { LBRACK }
0043      | ']'     { RBRACK }
0044
0045    (* strings *)
0046      | '"'     { DBL_QUOTE }
0047      | '\''    { SNG_QUOTE }
0048
0049    (* separators *)
0050      | ';'     { SEMI }
0051      | ','     { COMMA }
0052      | ".."    { DOTDOT }
0053      | '.'     { DOT }
0054
0055    (* arithmetic operators *)
0056      | '+'     { PLUS }
0057      | '-'     { MINUS }
0058      | '*'     { TIMES }
0059      | '/'     { DIVIDE }
0060      | '%'     { MOD }
0061      | '^'     { EXPONENT }
0062
0063    (* assignment operators *)
0064      | '='     { ASSIGN }
0065      | "+="    { ADD_ASN }
0066      | "-="    { SUB_ASN }
0067      | "*="    { MUL_ASN }
0068      | "/="    { DIV_ASN }
0069      | "%="    { MOD_ASN }
0070      | "^="    { EXP_ASN }
0071
0072    (* logical operators *)
0073      | "&&"    { LOG_AND }
0074      | "||"    { LOG_OR }
0075      | "!"     { LOG_NOT }
0076
0077    (* comparison operators *)
0078      | '<'     { LT }
0079      | '>'     { GT }
0080      | "=="    { EQ }
0081      | "!="    { NEQ }
0082      | "<="    { LEQ }
0083      | ">="    { GEQ }
0084
0085    (* shux *)
0086      | '?'     { QUES }
0087      | ':'     { COLON }
0088      | "::"    { FILTER }
0089      | '@'     { MAP }
```

```
0090      | "->"    { ARROW }
0091
0092    (* control keywords *)
0093      | "if"      { IF }
0094      | "then"    { THEN }
0095      | "elif"    { ELIF }
0096      | "else"    { ELSE }
0097      | "for"     { FOR }
0098      | "while"   { WHILE }
0099      | "do"      { DO }
0100      | "noop"    { PASS }
0101    (* declarations *)
0102      | "ns"      { NS }
0103      | "gn"      { GN }
0104      | "kn"      { KN }
0105      | "struct"  { STRUCT }
0106      | "let"     { LET }
0107      | "var"     { VAR }
0108      | "extern"  { EXTERN }
0109
0110    (* types *)
0111      | "int"               { INT_T }
0112      | "scalar" | "float"  { FLOAT_T }
0113      | "string"            { STRING_T }
0114      | "bool"              { BOOL_T }
0115      | "vector"            { VECTOR_T }
0116      | "_ptr"              { PTR_T }
0117
0118    (* literals *)
0119      | "true" | "false" as tf  { BOOL_LIT(bool_of_string tf) }
0120      | int as i                { INT_LIT(int_of_string i) }
0121      | float as f              { FLOAT_LIT(Float.of_string f) }
0122      | string                  { STRING_LIT(unescape s)}
0123      | id as n                 { ID(n) }
0124      | '_'                     { UNDERSCORE }
0125    (* ye good olde *)
0126      | eof       { EOF }
0127
0128    (* The Reign of Error *)
0129      | '"'       { raise (Exceptions.UnmatchedQuotation(!
lineno)) }
0130      | _ as e    { raise (Exceptions.IllegalCharacter(!filename,
e, !lineno)) }
0131
0132    (* comments *)
0133    and comment = parse
0134      | "/*"      { incr depth; comment lexbuf }
0135      | "*/"      { decr depth; if !depth > 0 then comment lexbuf
else token lexbuf }
0136      | newline   { incr lineno; comment lexbuf }
```

```
0137      | _          { comment lexbuf }
```

```
./shux/src/frontend/parser.mly
0001    %{
0002      open Ast
0003    %}
0004
0005    %token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK DBL_QUOTE
SNG_QUOTE
0006    %token SEMI COMMA DOTDOT DOT PLUS MINUS TIMES DIVIDE MOD
EXPONENT
0007    %token ASSIGN ADD_ASN SUB_ASN MUL_ASN DIV_ASN MOD_ASN EXP_ASN
0008    %token LOG_AND LOG_OR LOG_NOT LT GT EQ NEQ LEQ GEQ
0009    %token QUES COLON FILTER MAP ARROW IF THEN ELIF ELSE FOR WHILE
DO UNDERSCORE
0010    %token NS GN KN STRUCT LET EXTERN VAR INT_T FLOAT_T STRING_T
BOOL_T VECTOR_T PTR_T PASS
0011
0012    %token <bool> BOOL_LIT
0013    %token <int> INT_LIT
0014    %token <float> FLOAT_LIT
0015    %token <string> STRING_LIT
0016    %token <string> ID
0017    %token EOF
0018
0019    %start program
0020
0021    %type <Ast.program> program
0022
0023    %%
0024
0025    program:
0026      | ns EOF                                  { $1 }
0027
0028    ns:
0029      | ns_section let_section fn_section       { ($1, $2, $3) }
0030
0031
0032    /* namespace declaration rules */
0033    ns_section:
0034      | ns_decls                                { List.rev $1 }
0035      | /* nothing */                           { [] }
0036
0037    ns_decls:
0038      | ns_decls ns_decl                        { $2::$1 }
0039      | ns_decl                                 { [$1] }
```

```
0040
0041    ns_decl:
0042      | NS ID ASSIGN LBRACE ns RBRACE           { {nname = $2;
nbody = $5} }
0043
0044
0045    /* let declaration rules */
0046    let_section:
0047      | let_decls                                { List.rev $1 }
0048      | /* nothing */                            { [] }
0049
0050    let_decls:
0051      | let_decls let_decl                       { $2::$1 }
0052      | let_decl                                 { [$1] }
0053
0054    let_decl:
0055      | LET typ ID ASSIGN expr SEMI
{ LetDecl((Bind(Immutable, $2, $3), $5)) }
0056      | STRUCT ID LBRACE struct_def RBRACE       { StructDef({sname
= $2; fields = List.rev $4}) }
0057      | EXTERN ID LPAREN formals RPAREN
0058        ret_typ SEMI
{ ExternDecl({xalias = $2; xfname = $2;
0059                                                   xret_typ = $6;
xformals = $4;}) }
0060
0061    struct_def:
0062      | struct_def val_decl SEMI                 { $2::$1 }
0063      | val_decl SEMI                            { [$1] }
0064
0065
0066    /* function declaration rules */
0067    fn_section:
0068      | fn_decls                                 { List.rev $1 }
0069      | /* nothing */                            { [] }
0070
0071    fn_decls:
0072      | fn_decls fn_decl                         { $2::$1 }
0073      | fn_decl                                  { [$1] }
0074
0075    fn_decl:
0076      | fn_type ID LPAREN formals RPAREN ret_typ
0077        LBRACE statements ret_expr RBRACE        { {fname = $2;
fn_typ = $1; ret_typ = $6;
0078                                                   formals = $4;
body = List.rev $8; ret_expr = $9} }
0079      | fn_type ID LPAREN formals RPAREN ret_typ
0080        LBRACE ret_expr RBRACE                   { {fname = $2;
fn_typ = $1; ret_typ = $6;
0081                                                   formals = $4;
```

```
body = []; ret_expr = $8} }
0082    ret_typ:
0083      | typ                                          { Some($1) }
0084      | /* void return type */                       { None }
0085
0086    ret_expr:
0087      | expr                                         { Some($1) }
0088      | /* nothing */                                { None }
0089
0090
0091    /* function body rules */
0092    statements:
0093      | statements statement SEMI                    { $2::$1 }
0094      | statement SEMI                               { [$1] }
0095
0096    statement:
0097      | decl                                         { $1 }
0098      | expr                                         { Expr($1) }
0099
0100    decl:
0101      | decl_mod typ ID                              { VDecl(Bind($1,
$2, $3), None) }
0102      | decl_mod typ ID ASSIGN expr                  { VDecl(Bind($1,
$2, $3), Some($5)) }
0103
0104    expr:
0105      | asn_expr                                     { $1 }
0106
0107    asn_expr:
0108      | unary_expr ASSIGN asn_expr                   { Assign($1, $3) }
0109      | unary_expr asn_op asn_expr                   { Assign($1, Binop
($1, $2, $3)) }
0110      | if_expr                                      { $1 }
0111
0112    if_expr:
0113      | id_expr COLON if_expr
{ LookbackDefault($1, $3) }
0114      | bool_expr QUES fn_expr COLON if_expr         { Cond($1, $3,
$5) }
0115      | IF bool_expr THEN fn_expr ELSE if_expr       { Cond($2, $4,
$6) }
0116      | fn_expr                                      { $1 }
0117
0118    fn_expr:
0119      | iter_expr fn_op kns                          { Binop($1, $2,
$3) }
0120      | iter_expr                                    { $1 }
0121      | PASS                                         { Call(Some
["nop"],[])  }
0122
```

```
0123    kns:
0124      | kn fn_op kns                                { Binop($1, $2,
$3) }
0125      | kn                                         { $1 }
0126
0127    kn:
0128      | id_ns                                        { Id($1) }
0129      | lambda                                     { Lit($1) }
0130
0131    lambda:
0132      | LPAREN formals RPAREN ARROW
0133        LBRACE statements ret_expr RBRACE          { LitKn({lformals
= $2; lbody = List.rev $6; lret_expr = $7}) }
0134      | LPAREN formals RPAREN ARROW
0135        LBRACE ret_expr RBRACE                     { LitKn({lformals
= $2; lbody = []; lret_expr = $6}) }
0136
0137    iter_expr:
0138      | iter_type unit_expr gn_call                { Binop($2, $1,
$3) }
0139      | unit_expr                                  { $1 }
0140
0141    gn_call:
0142      | id_ns LPAREN actuals RPAREN                { Call(Some($1),
$3) }
0143      | UNDERSCORE                                 { Call(None, []) }
0144
0145    unit_expr:
0146      | bool_expr                                  { $1 }
0147
0148    bool_expr:
0149      | bool_or_expr                               { $1 }
0150
0151    bool_or_expr:
0152      | bool_or_expr LOG_OR bool_and_expr          { Binop($1, LogOr,
$3) }
0153      | bool_and_expr                              { $1 }
0154
0155    bool_and_expr:
0156      | bool_and_expr LOG_AND cmp_expr             { Binop($1,
LogAnd, $3) }
0157      | cmp_expr                                   { $1 }
0158
0159    cmp_expr:
0160      | eq_expr                                    { $1 }
0161
0162    eq_expr:
0163      | eq_expr eq_op relat_expr                   { Binop($1, $2,
$3) }
0164      | relat_expr                                 { $1 }
```

```
0165
0166    relat_expr:
0167      | shift_expr relat_op shift_expr          { Binop($1, $2,
$3) }
0168      | shift_expr                              { $1 }
0169
0170    /* unused, bit-wise operations not implemented */
0171    shift_expr:
0172      | arithmetic_expr                         { $1 }
0173
0174    arithmetic_expr:
0175      | add_expr                                { $1 }
0176
0177    add_expr:
0178      | add_expr add_op mult_expr               { Binop($1, $2,
$3) }
0179      | mult_expr                               { $1 }
0180
0181    mult_expr:
0182      | mult_expr mult_op unary_expr            { Binop($1, $2,
$3) }
0183      | unary_expr                              { $1 }
0184
0185    unary_expr:
0186      | prefix_op unary_expr                    { Uniop($1, $2) }
0187      | postfix_expr                            { $1 }
0188
0189    postfix_expr:
0190      | postfix_expr LBRACK expr RBRACK         { Binop($1, Index,
$3) }
0191      | id_ns LPAREN actuals RPAREN             { Call(Some($1),
$3) }
0192      | postfix_expr DOT ID                     { Access($1, $3) }
0193      | primary_expr                            { $1 }
0194
0195    primary_expr:
0196      | LPAREN expr RPAREN                      { $2 }
0197      | lit                                     { Lit($1) }
0198      | id_expr                                 { $1 }
0199
0200    id_expr:
0201      | id_ns DOTDOT INT_LIT                      { Lookback($1,
$3) }
0202      | id_ns                                     { Id($1) }
0203
0204
0205    /* function argument rules */
0206    formals:
0207      | formal_list                             { $1 }
0208      | /* nothing */                           { [] }
```

```
0209
0210    formal_list:
0211      | val_decl COMMA formal_list              { $1::$3 }
0212      | val_decl                                { [$1] }
0213
0214    actuals:
0215      | actual_list                             { $1 }
0216      | /* nothing */                           { [] }
0217
0218    actual_list:
0219      | expr COMMA actual_list                  { $1::$3 }
0220      | expr                                    { [$1] }
0221
0222
0223    /* operator rules */
0224    asn_op:
0225      | ADD_ASN                                 { Add }
0226      | SUB_ASN                                 { Sub }
0227      | MUL_ASN                                 { Mul }
0228      | DIV_ASN                                 { Div }
0229      | MOD_ASN                                 { Mod }
0230      | EXP_ASN                                 { Exp }
0231
0232    eq_op:
0233      | EQ                                      { Eq }
0234      | NEQ                                     { Neq }
0235
0236    relat_op:
0237      | LT                                      { Lt }
0238      | GT                                      { Gt }
0239      | LEQ                                     { Leq }
0240      | GEQ                                     { Geq }
0241
0242    add_op:
0243      | PLUS                                    { Add }
0244      | MINUS                                   { Sub }
0245
0246    mult_op:
0247      | TIMES                                   { Mul }
0248      | DIVIDE                                  { Div }
0249      | MOD                                     { Mod }
0250      | EXPONENT                                { Exp }
0251
0252    prefix_op:
0253      | PLUS                                    { Pos } /* can't
return nothing */
0254      | MINUS                                   { Neg }
0255      | LOG_NOT                                 { LogNot }
0256
0257    fn_op:
```

```
0258      | MAP                                       { Map }
0259      | FILTER                                    { Filter }
0260
0261    iter_type:
0262      | FOR                                       { For }
0263      | DO                                        { Do }
0264
0265
0266    /* type rules */
0267    val_decl:
0268      | typ ID                                    { Bind(Immutable,
$1, $2) }
0269
0270    typ:
0271      | typ LBRACK RBRACK                         { Array($1,
None) }
0272      | typ LBRACK INT_LIT RBRACK                 { Array($1,
Some($3)) }
0273      | unit_t                                    { $1 }
0274
0275    unit_t:
0276      | STRUCT id_ns                              { Struct($2) } /*
user defined structs */
0277      | primitive_t                               { $1 }
0278
0279    primitive_t:
0280      | INT_T                                     { Int }
0281      | FLOAT_T                                   { Float }
0282      | STRING_T                                  { String }
0283      | BOOL_T                                    { Bool }
0284      | PTR_T                                     { Ptr }
0285      | vector_t                                  { $1 }
0286
0287    vector_t:
0288      | VECTOR_T LT INT_LIT GT                    { Vector($3) }
0289
0290    fn_type:
0291      | GN                                        { Gn }
0292      | KN                                        { Kn }
0293
0294    decl_mod:
0295      | VAR                                       { Mutable }
0296      | /* nothing, val */                        { Immutable }
0297
0298
0299    /* literal syntax rules */
0300    lit:
0301      | struct_lit                                { $1 }
0302      | array_lit                                 { $1 }
0303      | vector_lit                                { $1 }
```

```
0304      | STRING_LIT                                { LitStr($1) }
0305      | BOOL_LIT                                  { LitBool($1) }
0306      | FLOAT_LIT                                 { LitFloat($1) }
0307      | INT_LIT                                   { LitInt($1) }
0308
0309    struct_lit:
0310      | id_ns LBRACE struct_lit_fields RBRACE   { LitStruct($1,
$3) }
0311      | id_ns LBRACE RBRACE                     { LitStruct($1,
[]) }
0312
0313    struct_lit_fields:
0314      | struct_lit_field SEMI struct_lit_fields { $1::$3 }
0315      | struct_lit_field                        { [$1] }
0316
0317    struct_lit_field:
0318      | DOT ID ASSIGN expr                      { StructField($2,
$4) }
0319
0320    array_lit:
0321      | LBRACK lit_elements RBRACK
{ LitArray(List.rev $2) }
0322      | LBRACK RBRACK                           { LitArray([]) }
0323
0324    vector_lit:
0325      | LPAREN lit_elements COMMA expr RPAREN
{ LitVector(List.rev ($4 :: $2)) }
0326
0327    lit_elements:
0328      | lit_elements COMMA expr                 { $3::$1 }
0329      | expr                                    { [$1] }
0330
0331    id_ns:
0332      | ID ARROW id_ns                          { $1::$3 }
0333      | ID                                      { [$1] }
```

./shux/src/frontend/ast.mli

```
0001    open Core.Std
0002
0003    type typ =
0004      | Int
0005      | Float
0006      | String
0007      | Bool
0008      | Struct of string list (* struct identifier *)
0009      | Array of typ * int option
```

```
0010      | Vector of int (* number of elements in vector *)
0011      | Ptr
0012      | Void
0013
0014    type mut =
0015      | Mutable
0016      | Immutable
0017
0018    type bind = Bind of mut * typ * string
0019
0020    type fn_typ =
0021      | Kn
0022      | Gn
0023
0024    type bin_op =
0025      | Add | Sub | Mul | Div | Mod | Exp
0026      | Eq | Lt | Gt | Neq | Leq | Geq
0027      | LogAnd | LogOr
0028      | Filter | Map
0029      | Index
0030      | For | Do
0031
0032    type un_op =
0033      | LogNot | Neg | Pos
0034
0035    type lambda = {
0036      lformals  : bind list;
0037      lbody     : stmt list;
0038      lret_expr : expr option;
0039    }
0040
0041    and lit =
0042      | LitInt of int
0043      | LitFloat of float
0044      | LitBool of bool
0045      | LitStr of string
0046      | LitKn of lambda
0047      | LitVector of expr list
0048      | LitArray of expr list (* include optional type annotation
here? *)
0049      | LitStruct of string list * struct_field list (* should
this be more sophisticated? *)
0050
0051    and struct_field = StructField of string * expr
0052
0053    and expr =
0054      | Lit of lit
0055      | Id of string list
0056      | Lookback of string list * int
0057      | Binop of expr * bin_op * expr
```

```
0058      | Assign of expr * expr
0059      | Call of string list option * expr list
0060      | Uniop of un_op * expr
0061      | LookbackDefault of expr * expr
0062      | Cond of expr * expr * expr (* technically Ternop *)
0063      | Access of expr * string
0064
0065    and stmt =
0066      | VDecl of bind * expr option
0067      | Expr of expr
0068
0069    type fn_decl = {
0070      fname     : string;
0071      fn_typ    : fn_typ;
0072      ret_typ   : typ option;
0073      formals   : bind list;
0074      body      : stmt list;
0075      ret_expr  : expr option;
0076    }
0077
0078    type struct_def = {
0079      sname     : string;
0080      fields    : bind list;
0081    }
0082
0083    type extern_decl = {
0084      xalias    : string; (* what we call inside shux *)
0085      xfname    : string; (* what we link to outside shux *)
0086      xret_typ  : typ option;
0087      xformals  : bind list;
0088    }
0089
0090    type let_decl =
0091      | LetDecl of bind * expr
0092      | StructDef of struct_def
0093      | ExternDecl of extern_decl
0094
0095    type ns_decl = {
0096      nname     : string;
0097      nbody     : program;
0098    }
0099
0100    and program = ns_decl list * let_decl list * fn_decl list
```

./shux/src/frontend/exceptions.ml

```
0001    (* Scanner Exceptions *)
```

```
0002    exception IllegalCharacter of string * char * int
0003    exception UnmatchedQuotation of int
```

./shux/.gitignore

```
0001    #OCaml
0002    _log
0003    _build
0004    *.cmi
0005    *.cmo
0006    *.native
0007    *.byte
0008    *.out
0009    *.ll
0010    *.error
0011    scanner.ml
0012    parser.output
0013    parser.ml
0014    parser.mli
0015    shuxc
0016    build/
0017
0018    #nhc
0019    /src/version.ml
0020
0021    #Mac
0022    .DS_Store
0023    ._*
0024
0025    #Windows
0026    Thumbs.db
```

./shux/build/README.txt

```
0001    # How to run
0002
0003    make
0004    ./shuxc test.shux > test.ll
0005    lli test.ll
```

./shux/build/build_example.shux
```
0001    extern print(int x);
0002
0003    kn main() int{
0004       int x = 12;
0005       int y = 81;
0006       print(3>2 ? 18: 3);
0007
0008       print(y%x);
0009       0
0010    }
```

./shux/build/README.txt~
```
0001    #How to run
0002
0003    make
0004    ./shuxc test.shux > test.ll
0005    lli test.ll
```

./shux/build/Makefile
```
0001    CAMLC=ocamlc
0002    OCAML=ocamlfind $(CAMLC) -linkpkg -thread -package core -
package llvm -package llvm.analysis
0003    OBJS=exceptions.cmo sast.cmo ast.cmo astprint.cmo scanner.cmo
parser.cmo semant.cmo ast_sast.cmo sast_cast.cmo llast.cmo
lltranslate.cmo cast_llast.cmo
0004    INCLUDE=-I ../src/include -g
0005    OCAML_FLAGS=$(INCLUDE) $(OBJS)
0006    EXAMPLES=../examples
0007    SYMLINK=ln -s -f
0008    PREREQS=symlink
0009    COMPILER=shuxc
0010    TEST_DIR=../tests
0011    TEST=run_tests.sh
0012    TARGETS=$(PREREQS) $(COMPILER)
0013
0014    .PHONY: default symlink hello
0015    default: $(TARGETS)
0016
0017    symlink:
0018            $(SYMLINK) ../src/frontend/exceptions.ml
exceptions.ml
```

```
0019            $(SYMLINK) ../src/frontend/ast.mli ast.ml
0020            $(SYMLINK) ../src/backend/sast.mli sast.ml
0021            $(SYMLINK) ../src/backend/cast.mli cast.ml
0022            $(SYMLINK) ../src/backend/codegen.ml codegen.ml
0023            $(SYMLINK) ../src/frontend/parser.mly parser.mly
0024            $(SYMLINK) ../src/frontend/scanner.mll scanner.mll
0025            $(SYMLINK) ../src/backend/semant.ml semant.ml
0026            $(SYMLINK) ../src/backend/sast_cast.ml sast_cast.ml
0027            $(SYMLINK) ../src/backend/ast_sast.ml ast_sast.ml
0028            $(SYMLINK) ../src/shuxc.ml shuxc.ml
0029            $(SYMLINK) ../src/frontend/astprint.ml astprint.ml
0030            $(SYMLINK) ../src/backend/llast.mli llast.ml
0031            $(SYMLINK) ../src/backend/cast_llast.ml cast_llast.ml
0032            $(SYMLINK) ../src/backend/lltranslate.ml
lltranslate.ml
0033
0034    shuxc.native:
0035            ocamlbuild -use-ocamlfind -tag thread -pkgs
llvm,llvm.analysis,core -cflags -w,+a-4 shuxc.native
0036
0037    scanner.ml: scanner.mll exceptions.ml
0038            ocamllex scanner.mll
0039
0040    parser.ml parser.mli: parser.mly ast.ml
0041            ocamlyacc parser.mly
0042
0043    shuxc: shuxc.ml $(OBJS)
0044            $(OCAML) $(OCAML_FLAGS) -o shuxc $<
0045
0046    %.cmo: %.ml
0047            $(OCAML) $(OCAML_FLAGS) -c $<
0048    %.cmi: %.mli
0049            $(OCAML) $(OCAML_FLAGS) -c $<
0050
0051    #generated by ocamldep *.ml *.mli
0052    ast.cmo :
0053    ast.cmx :
0054    astprint.cmo : ast.cmo
0055    astprint.cmx : ast.cmx
0056    ast_sast.cmo : sast.cmo ast.cmo
0057    ast_sast.cmx : sast.cmx ast.cmx
0058    cast.cmo : sast.cmo
0059    cast.cmx : sast.cmx
0060    exceptions.cmo :
0061    exceptions.cmx :
0062    parser.cmo : ast.cmo parser.cmi
0063    parser.cmx : ast.cmx parser.cmi
0064    sast_cast.cmo : sast.cmo cast.cmo
0065    sast_cast.cmx : sast.cmx cast.cmx
0066    cast_llast.cmo: cast.cmo llast.cmo
```

```
0067    cast_llast.cmx: cast.cmx llast.cmx
0068    sast.cmo :
0069    sast.cmx :
0070    llast.cmo:
0071    llast.cmx:
0072    lltranslate.cmo: llast.cmo
0073    lltranslate.cmx: llast.cmx
0074    scanner.cmo : parser.cmi exceptions.cmo
0075    scanner.cmx : parser.cmx exceptions.cmx
0076    semant.cmo : sast.cmo astprint.cmo ast.cmo
0077    semant.cmx : sast.cmx astprint.cmx ast.cmx
0078    shuxc.cmo : semant.cmo scanner.cmo sast_cast.cmo parser.cmi
codegen.cmo \
0079        astprint.cmo ast_sast.cmo cast_llast.cmo
0080    shuxc.cmx : semant.cmx scanner.cmx sast_cast.cmx parser.cmx
codegen.cmx \
0081        astprint.cmx ast_sast.cmx cast_llast.cmx
0082    parser.cmi : ast.cmo
0083    .PHONY: all clean cleanall autoclean hello test
0084
0085    all: cleanall symlink $(OBJS) $(TARGETS)
0086
0087    cleanall: cleantests
0088            rm -rf *.* shuxc
0089
0090    clean: cleantests
0091            rm -rf *.cmx *.cmi *.cmo *.cmx *.o parser.ml
parser.mli scanner.ml shuxc
0092
0093    cleantests:
0094            cd $(TEST_DIR) && rm -rf *.ll && rm -rf *.out
0095
0096    autoclean:
0097            ocamlbuild -clean
0098
0099    hello: $(COMPILER) $(EXAMPLES)/hello_world.shux
0100            ./shuxc $(EXAMPLES)/nstest.shux > nstest.ll
0101            cat nstest.ll
0102            lli nstest.ll
0103
0104    tests: cleantests
0105            @cd $(TEST_DIR) && $(SYMLINK) ../build/shuxc shuxc
&& ./$(TEST)
0106
0107    tests_front: cleantests
0108            @cd $(TEST_DIR) && $(SYMLINK) ../build/shuxc shuxc
&& ./$(TEST) COMPILE
```