Loom Report (Draft)

Johnathan Jenkins

August 18, 2011

Abstract

Loom is a domain-specific language for general-purpose graphics processing units ('gpgpus') designed to support millions of parallel execution threads. It is designed to interface with code written in a 'host' language (typically C or C++) running on the cpu. Loom is sufficiently low-level to implement efficient parallel algorithms, but includes facilities such as *parallel map*, *parallel reduce* and *parallel scan* to abstract common patterns.

Contents

Chapter 1

Introduction

1.1 Background

Loom is a language for programming massively parallel co-processors used on many modern computers. The specific devices supported are nvidia gpgpus using the cuda architecture.

Cuda devices support tens of thousands, or even millions of simultaneous execution threads, with hundreds of threads running in parallel at any time. The threads are organized into a hierarchy: groups of 32 threads make up *warps* that execute in series on single core, and do not have to be synchronized; groups of up to 512 threads make up *blocks*, which can use high-speed shared memory for inter-thread communication and user-managed caching; the blocks collectively form a *grid*, and all threads in a single grid execute the same *kernel* code. Although the kernel is the same for each thread in a grid, individual threads need not follow the same execution path.

In addition to shared memory, devices have cached *constant memory*, registers, and *global memory*. Transfer of data from the host computer's main memory to the device global memory space occurs in code on the host side (that is, not within kernels running on the gpgpu device). This simplifies loom, which is solely for compiling gpgpu kernels. In addition to memory transfer to and from the host computer, all input and output occurs on the host side.

Most cuda device programming is done in a C-like language supplied by nvidia. The target language for loom, however, is a lower-level language called ptx. Ptx looks very much like a traditional assembly language, although it runs on a device-independent virtual machine and is jit-compiled to a *cubin* binary for execution.

Extensive documentation on cuda is available from *http://developer.nvidia.com*.

1.2 Overview of Loom

1.2.1 Design Goals and Language Features

Loom attempts to abstract away many of the repetitive and error-prone details involved in writing gpgpu kernels, such as explicit array index calculations and thread barrier

synchronization, while remaining at a sufficiently low level to allow interesting parallel algorithms to be implemented (rather than merely used in a black-box library, such as the CUDA linear algebra libraries supplied by NVIDIA).

The language is statically typed. Types are indicated following a colon after a variable name: `x: Int32`. In addition to *basic types* such as Int32 and Float32, which correspond directly to PTX types, there are *records* such as `pair: {first:Int32, second:Int32}` (a pair of integers), *vectors* such as `v: Int32[10]`, and two dimensional arrays such as `a: Int32[5,5]`. Several special constants make it easy to work on large vectors and arrays in parallel.

LOOM has several standard control constructs for conditionals and looping, which are demonstrated in the sample programs below. The language also has operators designed to simplify parallel programming. There is a *parallel map* operator which applies a function of one variable to each element in a vector or array: `f // v`. The *parallel reduce* operator, `/.`, applies a function of two variables (which should be associative in those variables) repeatedly to reduce a vector or array to a single value per CUDA block (subsequent kernel calls, or code running on the host CPU, can then be used to complete the reduction). LOOM also defines left and right *parallel scan* operators, `/:` and `:/`. As with the reduction operator, scans work across blocks rather than across entire vectors.

Shared memory, which functions both as a user-managed cache and a mechanism for sharing data between threads in a single block, is allocated by declaring a variable with the `shared` keyword. CUDA programs typically use explicit barrier synchronization instructions to synchronize threads within a block; in LOOM, statements that move data between memory state spaces and alter state are synchronized by default.

Due to limitations of at least certain CUDA architectures, functions cannot be recursive. A newer architecture used on high-end graphics cards, called 'Fermi', permits recursive functions as well as a number of other powerful features. It would be an interesting exercise to extend LOOM to take advantage of some of those features.

1.2.2 Representative Programs

Finally, we show listings of a few short programs illustrating some of LOOM's features.

▷ *Find the maximum values in a two-dimensional array (by CUDA block):*

```
kernel maximum(in: Int32[X_THREADS, Y_THREADS],
    out: Int32[X_BLOCKS, Y_BLOCKS])
out <- max /. in
```

▷ *Shift the values in a vector to the left by exactly one block:*

```
kernel shiftLeft(in: Float32[THREADS], out: Float32[THREADS])
    current: Range <- block(B)   -- 'B' is the current block index
    previous: Range <- block(B-1)   -- Range type {Int32, Int32}
    out[[current]] <- in[[previous]]
```

Note that arrays indexed with double brackets are bounds-checked and padded to zero outside the defined range. Single-brackets perform unsafe array indexing. This following is a more explicit, but equivalent, implementation:

```
kernel shiftLeft1(in: Float32[THREADS], out: Float32[THREADS])
    if B > 0 then
        for i: Int32 <- B*BLOCKSIZE .. (B+1)*BLOCKSIZE
```

```
            out[i] <- in[i-BLOCKSIZE]
    else
        for i: Int32 <- B*BLOCKSIZE .. (B+1)*BLOCKSIZE
            out[i] <- 0
```

The `..` symbol in the `for` statement shows that `i` takes on successive values from `B*BLOCKSIZE` (inclusive) to `(B+1)*BLOCKSIZE` (exclusive). We could have expressed the same range of values by writing `B*BLOCKSIZE ... (B+1)*BLOCKSIZE-1`, where the `...` symbol shows that `i` goes up to `(B+1)*BLOCKSIZE-1` (inclusive).

▷ *Compute the sum of squares (by block):*

```
func sum(x: Float32, y: Float32): Float32
    return x + y

func sqr(x: Float32): Float32
    return x*x

kernel sumOfSquares(in: Int32[THREADS], out: Int32[BLOCKS])
    local current: Range <- block(B)
    shared t: Float32[BLOCKSIZE] <- in[current]
    out[B] <- sum /. (sqr // t)
```

Chapter 2

Tutorial

This tutorials shows the steps needed to get started with LOOM.

▷ *Verify that Caml is installed:*

```
calliope:src jjenkins$ ocaml -version
The Objective Caml toplevel, version 3.12.0
```

▷ *Verify that all required files are present:*

```
calliope:src jjenkins$ ls
BUGS          depend.mk         scanner.mll
Makefile      loom.ml           setup.sh
README        parser.mly        test
ast.ml        preprocessor.ml   utx.ml
compiler.ml   preprocessor.mli  utx.mli
compiler.mli  sast.ml
```

▷ *Complete dependencies in the Makefile using ocamldep and build the system:*

```
calliope:src jjenkins$ ./setup.sh
rm -f loom scanner.ml parser.ml parser.mli parser.output
rm -f *.cm[io]
ocamlc -c str.cma utx.mli
ocamlc -c str.cma utx.ml
ocamlc -c str.cma preprocessor.mli
ocamlc -c str.cma preprocessor.ml
ocamlc -c str.cma ast.ml
ocamlyacc parser.mly
ocamlc -c parser.mli
ocamlc -c str.cma parser.ml
ocamllex scanner.mll
169 states, 10614 transitions, table size 43470 bytes
ocamlc -c str.cma scanner.ml
ocamlc -c str.cma sast.ml
ocamlc -c str.cma compiler.mli
ocamlc -c str.cma compiler.ml
ocamlc -c str.cma loom.ml
ocamlc -o loom str.cma utx.cmo preprocessor.cmo scanner.cmo \
parser.cmo ast.cmo sast.cmo compiler.cmo loom.cmo
```

▷ *Check command options:*

```
calliope:src jjenkins$ ./loom -help
LOOM version 0.1
usage: ./loom [-p|-a|-s|-c] [-in <input>.<ext>] [-out <output>.<ext>]
  -p : preprocess input
  -a : parse input
  -s : typecheck input
  -c : compile to PTX code
  -in : set input filename
  -out : set output filename
  -version : print version string and exit
  -help  Display this list of options
```

When it is run with `-in <file>` or `-out <file>` arguments, LOOM will try to figure out what to do based on the file extensions (`.lm` for LOOM source, `.lmp` for preprocessed source, `.lma` for parsed intermediate format, `.lms` for a typechecked, semantically analyzed intermediate format, and `.ptx` for PTX assembly).

▷ *Create a simple* LOOM*test file:*

```
cat > test/foo.lm
kernel foo(a: Uint32, b: Uint32)
    sum: Int32 <- a

    for i: Int32 <- 0...9
        sum <- sum + i

    b <- sum
    return
^D
```

▷ *Preprocess to get the alternative free-form syntax:*

```
calliope:src jjenkins$ ./loom -p -in test/foo.lm
kernel foo(a: Uint32, b: Uint32) {
    sum: Int32 <- a;
    for i: Int32 <- 0...9 {
        sum <- sum + i;
    }
    b <- sum;
    return;
}
```

▷ *Check that it parses properly:*

```
calliope:src jjenkins$ ./loom -a -in test/foo.lm
Fatal error: exception Parsing.Parse_error
```

An error! The problem is that LOOM requires that variable declarations specify an address space. Let's fix this.

```
calliope:src jjenkins$ cat > test/foo.lm
kernel foo(a: Uint32, b: Uint32)
    local sum: Int32 <- a

    for i: Int32 <- 0...9
        sum <- sum + i

    b <- sum
    return
^D
```

```
calliope:src jjenkins$ ./loom -a -in test/foo.lm
local sum: Int32 <- a;
for i: Int32 <- 0 .. (1 + 9)
sum <- (sum + i);
}
b <- sum;
return ;
}
```

LOOM has parsed the input file into an (untyped) AST, and used the tree to print out the structure in a readable form so we can confirm that the input was parsed correctly.

▷ *Run the type-checker:*

```
calliope:src jjenkins$ ./loom -s -in test/foo.lm
Fatal error: exception Failure("
*** Variable sum initialization has incorrect type.")
```

Another bug in the LOOM code – this time, we forgot to convert the type of `a` to match `sum`.

```
calliope:src jjenkins$ cat > test/foo.lm
kernel foo(a: Uint32, b: Uint32)
    local sum: Int32 <- a::Int32

    for i: Int32 <- 0...9
        sum <- sum + i

    b <- sum
    return
^D
calliope:src jjenkins$ ./loom -s -in test/foo.lm
Typechecking finished with no errors.
```

▷ *Compile to* PTX*:*

```
calliope:src jjenkins$ ./loom -c -in test/foo.lm
        .version 1.4
        .target sm_10, map_f64_to_f32
        .entry foo(.param .u32 a, .param .u32 b, .param .u32 grid)
        {
        .reg .pred %rp<99>;
        .reg .s32 %rs<99>;
        .reg .s64 %rsl<99>;
        .reg .u32 %ru<99>;
        .reg .u64 %rul<99>;
        .reg .f32 %rf<99>;
        .reg .f64 %rfl<99>;
        .reg .b32 %rb<99>;
        .reg .u16 %rh<99>;
        .reg .u32 %threads;
        .reg .u32 %blocks;
        .reg .u32 %blocksize;
        ld.param.u32    %threads, [grid+0];
        ld.param.u32    %blocks, [grid+4];
        ld.param.u32    %blocksize, [grid+8];
        {
        .local .s32 sum;

  // ... Omitted
```

```
        bra     L0;
        }
L0:
        exit;
        }
```

Unfortunately, in this case the omitted section hides a number of sins – the back end code generation module is not finished at the time of this writing, so the generated PTX code contains lots of gaps.

When the back end is completed, however, it should be possible to use the PTX output with the CUDA software development kit to specify GPGPU device code just as one can do with kernels written in the CUDA (C-like) language and compiled to PTX with the command `nvcc -ptx`

▷ *Compile to* PTX*:*

```
calliope:cuda jjenkins$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2011 NVIDIA Corporation
Built on Wed_Mar_23_02:02:59_PDT_2011
Cuda compilation tools, release 4.0, V0.2.1221
calliope:cuda jjenkins$ nvcc -ptx foo_kernel.cu
calliope:cuda jjenkins$ nvcc -o foo foo.c foo_kernel.ptx
...
```

Chapter 3

Loom Reference Manual

3.1 Introduction

This reference manual gives a brief description of the LOOM language, following the model of Appendix A to Kernighan and Ritchie, *The C Programming Language* (2nd ed.). In some cases, sections headings have been taken directly from K&R.

Because LOOM is designed to closely match the target CUDA architecture, in a number of cases LOOM language features should be understood with reference to the relevant NVIDIA documentation. For example, internal floating-point formats, limitations on the number of threads in a warp, and similar information can be found in the CUDA API Reference Manual Version 4.0, the CUDA C Programming Guide Version 4.0, and the CUDA PTX: Parallel Thread Execution ISA Version 1.4, all of which are available at *http://developer.nvidia.com/nvidia-gpu-computing-documentation*.

3.2 Source Format

A program is read in as lines of ASCII text separated by newline characters. A program consists of exactly one `kernel` definition, along with `func` definitions for supporting device functions, and type declarations.

LOOM programs are typically written using line breaks and indentation to indicate block structure, rather than with explicit curly braces and semicolons as for C-syntax languages. To facilitate parsing, however, the first stage in compiling a LOOM program is running the input source code through a preprocessor that inserts braces and semicolons to mark blocks and statements.

The preprocessor goes through the input line by line, keeping a stack of indentation levels in a stack, as well as the indentation of the preceding line. For simplicity, indentation is indicated solely by spaces at the beginning of the line – behavior is undefined if there are tab characters in the input. The preprocessor applies the following rules for each line:

1. If the current line is indented further than the preceding line, assume that the preceding line introduces a block. Add a semicolon to the end of the preceding line,

push the current-line indentation level onto the stack, and add a semicolon at the end of the current line.

2. If the current line is indented the same as the preceding line, add a semicolon at the end.

3. If the line is indented less than the preceding line, pop the stack of all levels greater than the current indentation level. Insert closing braces for each element of the stack popped, and continue with steps 1–3 as appropriate.

▷ *Example: convert the following code to use explicit braces and semicolons:*

```
func foo(a: Int32): Int32
    local sum: Int32 <- 0
    for i: Int32 <- 1 .. 5
        iSqr: Int32 <- i*i
        sum <- sum + i2
    return sum   -- sum = 1 + 2*2 + 3*3 + 4*4 = 30
```

The preprocessor converts this into

```
func foo(a: Int32): Int32 {
    local sum: Int32 <- 0;
    for i: Int32 <- 1 .. 5 {
        iSqr: Int32 <- i*i;
        sum <- sum + i2;
    }
    return sum;
}
```

3.3 Lexical Elements

Tokens may be comments, keywords, operators, identifiers, constants, or thread parameters. There are also a few delimiter tokens that do not fall into these categories, such as parentheses (which are used to indicate grouping within expressions and to set off function arguments) and the two- and three-dot range symbols used within `for` statements.

After the preprocessing stage, whitespace is generally ignored except where necessary to separate adjacent tokens that would otherwise be lexically ambiguous.

3.3.1 Comments

Comments are indicated by two adjacent hyphen characters ('--'). They may begin anywhere in a line, and continue until the next newline character, which marks the end of the line. There is no special syntax for multiline comments. Comments are treated as whitespace.

3.3.2 Reserved Words

The following keywords, typenames, and predefined value symbols are reserved:

```
break       global      Bool        B
sync        kernel      Void        BLOCKS
continue    local       Float32     BLOCKSIZE
else        return      Float64     T
if          shared      Int32       THREADS
pass        then        Int64
for         type        Uint32      FALSE
func        while       Uint64      TRUE
```

In addition, to facilitate working with two-dimensional arrays, 2-D variations of the block and thread symbols in the right-hand column are also used (`X_B`, `Y_B`, `X_BLOCKS` and `Y_BLOCKS`, etc.).

3.3.3 Operators and Delimiters

LOOM recognizes the following operators and delimiters, ranked in order of precedence from highest (top row) to lowest (bottom row):

```
{   }   ;
(   )   [   ]   [[  ]]
:   ::
!
*   /   %   <<  >>  &
+   -   |   ^
=   !=  <   <=  >   >=
&&
||
//  /.  /:  :/  ..  ...
<-
```

Individual operators and delimiters are described in the following table:

`{ } ;`	statement blocks, statement termination
`[ ]`	unsafe array indexing
`[[ ]]`	safe array indexing with zero-padding
`:`	type tag,
`::`	type conversion operator
`!`	boolean negation
`* / %`	multiply, divide, modulo
`<< >> &`	shift-left, shift-right, bitwise and
`+ -`	add, subtract / unary negate
`|^`	bitwise or, xor / unary one's complement
`= != < <= > >=`	relational operators
`&& ||`	logical and, or
`// /. /: :/`	parallel map, reduce, scanl and scanr
`.. ...`	range delimiter in `for` statements
`<-`	assignment

Most binary operators group left-to-right, although assignments and expressions with unary operators group right-to-left, as do the parallel array operators.

3.3.4 Identifiers

User-defined identifiers may refer to functions, variables, types, arrays, or records. Identifiers must be accepted by the following regular expression:

```
identifier:: [a-zA-Z][a-zA-Z0-9_']*
```

3.3.5 Constants

Loom supports literal constant expressions for booleans and 32-bit integers and floating-point numbers.

Floating-point literals are written with an optional decimal point, and an optional signed exponent following `[eE]`.

Constants may be followed by a type conversion operator to force their type to be different from the default (e.g., `123::Float32` will be read in as a `Float32`).

Constants may be combined into *constant expressions* using the evaluation rules for expressions set forth below.

3.3.6 Boolean Constants

TRUE and FALSE are predefined symbolic constants of type `Bool`.

3.3.7 Thread Parameters

A number of symbolic parameters are defined at the time the kernel is invoked by host code running on the CPU. These parameters allow the GPGPU device thread to know about its execution context.

`B`	Index of the current thread block
`BLOCKS`	Number of thread blocks in the grid
`BLOCKSIZE`	Number of threads per block
`T`	Index of the current thread
`THREADS`	Number of threads in the grid

The parameters in the table correspond to grids laid out in one dimension (i.e., laid out to easily accommodate one-dimensional data structures for inputs and outputs to and from the GPGPU). Loom also supports two-dimensional layouts, in which case analogues to the parameters above are defined at the time of kernel invocation, prefaced by `X_` and `Y_`.

3.4 Data Types

Each variable in Loom has an associated identifier (its name) and a data type. The basic Loom data types generally correspond their CUDA/PTX equivalents, except for `Bool`s, which are primarily used to control program execution. In addition to the basic types, Loom supports (one- and two-dimensional) array and record container types, as well as new types defined in a `type` statement.

3.4.1 Type Declarations

Variables must be declared before they are used (although in the special case of `for` statements, the declaration can occur within the statement itself). Declaration statements take an optional initialization assignment. If no explicit initialization is present, the variable is set to zero (in the case of numerical types) or `FALSE` (in the case of `Bool`s).

```
local x: Int32  -- Initialized to 0
local e: Float32 <- 2.718281828
for i: Int32 <- 0 .. 10
    x <- x + i
```

All variable objects exist in one of three memory spaces: global memory (the main memory on the GPGPU device), shared memory (a much smaller amount of high-speed memory simultaneously accessible to the threads in a single CUDA block) or local memory (memory local to a single thread, often mapped to registers).

```
variable-decl:  memory-space identifier ':' type initializer
memory-space:   'local' | 'shared' | 'global'
type:           identifier | basic-type | array-type | record-type
basic-type:     'Bool' | 'Byte' | 'Float32' | ...
array-type:     type '[' constant-expression ']'
record-type:    '{' identifier ':' type maybe-more '}'
maybe-more:     '' | ',' identifier ':' type maybe-more
initializer:    '' | '<-' constant-expression
```

3.4.2 Defining New Types

Users may define their own types to supplement the built-ins.

```
type-definition:  'type' identifier type
```

3.5 Expressions

Expressions are combinations of variables, constants, function calls and operators that have a value, and therefore a type. Instead of specifying an unambiguous formal grammar for expressions (as in K&R Appendix A), ambiguities in the following grammar are resolved with reference to the operator precedence and grouping rules provided above.

```
expression:       identifier
                | <literal>
                |  '(' expression ')'
                | binop-expr | unary-expr
                | reference
                | assignment-expr
                | parallel-expr
                | function-call
binop-expr:       expression binop expression
binop:            + | - | * | / | ...
unary-expr:       unary-op expression
unary-op:         ! | - | ^
reference:        array-unsafe-ref | array-safe-ref | record-ref
array-unsafe-ref: identifier '[' expression second-index ']'
```

```
array-safe-ref:   identifier '[[' expression second-index ']]'
second-index:     '' | ',' expression
record-ref:       identifier '.' identifier
assignment-expr:  lhs '<-' expression
lhs:              identifier | reference
parallel-expr:    map-expr | reduce-expr | scanl-expr | scanr-expr
map-expr:         function-name // array-expr
reduce-expr:      function-name /. array-expr
scanl-expr:       function-name /: array-expr
scanr-expr:       function-name :/ array-expr
function-name:    identifier
array-expr:       identifier | map-expr
function-call:    identifier '(' expression maybe-others ')'
maybe-others:     '' | ',' expression maybe-others
```

Note that an `lhs` may be the name of an (unindexed) array, in which case the corresponding assignment expression is compiled into a loop over the array, where each iteration performs a sub-assignment to one element of the array.

3.6 Control Structures

Most LOOM control structures are similar to those in traditional imperative languages. The exceptions are the *parallel operators*. Strictly speaking, LOOM programs run inside a single thread on a CUDA device, so in a sense these operators are not parallel by themselves. Rather, they are designed to encapsulate common patters for coordinating many threads to do work and share data in parallel. Each thread knows certain characteristic information about its context (such as the index numbers of its thread and block), which is used to differentiate thread behavior. For example, each thread may operate on a different element of an array, where the array is indexed by the thread index number.

3.6.1 Parallel Operators

Certain type restrictions apply to `parallel-expr` expressions. For each of the four operators, the function on the left must accept a value having the same type as the type contained within the array on the right, and the array's dimension should match the number of CUDA threads in the grid (in either one or two dimensions).

The map operator, `f // v`, produces an array with the same dimensions as the starting array (`THREADS` or `X_THREADS` × `Y_THREADS`). Each element of the array gets mapped by `f` (a function of one variable) to a new value.

The reduce operator, `f /. v`, produces a scalar. `f` must be a function of two arguments, and also be associative in those arguments. If `f` is defined to add its two arguments, for example, then `f /. 0, 1, 2, 3` would yield the scalar value `6`.

The two scan operators, `/:` and `:/`, produce arrays with dimensions `BLOCKS` or `X_BLOCKS` × `Y_THREADS` (in the case of two-dimensional arrays, the reduce and scan operators work over the first array index). If `f` is defined to add its two arguments, as above, then `f /: 0, 1, 2, 3` would yield the array value `0, 1, 3, 6`, and `f :/ 0, 1, 2, 3` would yield the array value `6, 6, 5, 3`.

If there are n parallel threads, `//` operates in one step, whereas `/.`, `/:`, and `:/` operate in $\log n$ steps.

The foregoing operators do implicit barrier synchronization among threads in a block. However, lower-level parallel code may need to do explicit synchronization. For this, use the `sync` statement.

3.6.2 Statements and Blocks

```
statement:     expression ';'
             | block
             | if-stmt | for-stmt | while-stmt
             | 'break' ';'
             | 'continue' ';'
             | 'return' return-val ';'
             | 'sync'
block:         '{' statement other-stmts '}'
other-stmts:   '' | other-stmts
return-val:    '' | expression
```

Blocks are significant not only for grouping statements within control structures, but also because local variables have block scope.

The loop control statements `break` and `continue` act in the usual way on the innermost surrounding loop.

3.6.3 Conditionals

```
if-stmt:     'if' expression 'then' block maybe-else
maybe-else:  '' | 'else' block
```

LOOM follows the C convention for resolving `else` ambiguity: the `else` connects with the last-encountered `else`-less `if`.

3.6.4 Loops

```
while-stmt:  'while' expression block
for-stmt:    'for' loop-var '<-' expression dots expression block
loop-var:    identifier maybe-type
maybe-type:  '' | ':' type
```

If a loop variable is declared within the `for` statement, its scope is the surrounding block rather than the block in the body of the statement.

3.7 Functions

LOOM has two kinds of functions: kernel functions (of which there is exactly one per program, and which is called from code running on the host CPU) and device functions (which can be called by the kernel function or by other device functions). Due to limitations of the CUDA architecture, recursion is not permitted.

3.7.1 Kernels

```
kernel-func:  'kernel' identifier '(' parameters ')' block
parameters:   parm maybe-parms
parm:         identifier ':' type
maybe-parms:  '' | ',' maybe-parms
```

The kernel function is the only function that does not have a return type – it can affect the world only by altering the global variables passed in as arguments.

The `block` in the body of the kernel function must contain at least one `return` statement.

3.7.2 Device Functions

```
device-func:  'func' identifier '(' parameters ')' ':' type block
```

Device function definitions are similar to kernel functions, except that they have a return type and are introduced by a different keyword.

Arguments are generally passed by value except for arrays and records, which are passed by reference. In a function-call statement such as `f(1+1,2+2,a)`, expressions in the argument list are evaluated from left to right before the function call. Within the function body, arguments exist in the register memory space (for arrays and records, the corresponding pointers are stored in register memory space).

In fact, unlike a traditional assembly language, PTX includes support for passing variables in a C-like syntax, and LOOM function calls compile to this form in a straightforward way. Accordingly, further details about functions can be found in the PTXdocumentation supplied by NVIDIA.

Chapter 4

Architecture

The LOOM program is implemented in Caml, and divided into several modules that operate in a pipeline to translate input source code through various intermediate stages, and finally into PTX assembly. In addition to the modules described below, there is also a module of utility functions (see `utx.mli`), primarily for file input and output.

4.1 Top Level

The top level of the system, defined in `loom.ml`, contains a command line argument processor using the Caml `Arg` package, and uses pattern matching on the following variant types to provide a flexible interface to the various translation stages.

```
type filetype = Source | Pre | Parsed | Checked | Ptx
```

The top level also contains code to infer filetypes based on extension, and to print error messages for unsupported translations (e.g., there are currently no serializer or reader implemented for the typechecked ADT, so it is not possible to start from that intermediate format). These features proved useful for testing and debugging the compiler front end.

4.2 Preprocessor

The preprocessor has a very simple interface, consisting of a function of the following type:

```
val embrace : string list -> string
```

This function adds curly brackets and semicolons to LOOM code based on indentation level only, and does not need to know anything about the LOOM language. Indeed, the same module could be used to add C-like block structure to a completely different indentation-structure language.

4.3 Scanner and Parser

Loom uses `ocamllex` and `ocamlyacc` to tokenize the input and do a syntax-directed translation into an abstract syntax tree (AST), respectively. The types implementing the AST, contained in the `Ast` module, are as follows.

```
type memspace = Local | Shared | Global

type binop =
    Mult    | Div     | Plus    | Minus   | Mod
  | Lshift  | Rshift  | BitOr   | BitAnd  | BitXor
  | Equal   | NotEq   | LessEq  | Less    | GreaterEq | Greater
  | Map     | Reduce  | Scanl   | Scanr   | Or        | And

type unop =
    Not | Negative | BitComplement

type loom_type =
    Bool | Void | Int32 | Int64 | Uint32 | Uint64 | Float32 | Float64
  | Array   of loom_type * expr
  | Record  of (string * loom_type) list
  | Type    of string  (* to be associated with a loom_type later *)

and expr =
    IntLiteral    of int
  | FloatLiteral  of float
  | BoolLiteral   of bool
  | Convert       of expr * loom_type
  | Place         of place
  | Binop         of expr * binop * expr
  | Unop          of unop * expr
  | Call          of string * expr list
  | NullExpr

and place =
    Aref          of string * expr
  | Saref         of string * expr
  | Rref          of string * string
  | Id            of string

type stmt =
    Block   of stmt list
  | Vdecl   of memspace * string * loom_type * expr option
  | Tdecl   of string * loom_type
  | Expr    of expr
  | Return  of expr
  | Assign  of place * expr
  | If      of expr * stmt list * stmt list
  | For     of string * loom_type * expr * expr * stmt list
  | While   of expr * stmt list
  | Break
  | Continue
  | Sync
  | NullStmt

type func = {
  fname    : string;
```

```
  formals  : (string * loom_type) list;
  ret_type : loom_type;
  body     : stmt list;
}

type kernel = {
  kname    : string;
  kformals : (string * loom_type) list;
  kbody    : stmt list;
}

type program = kernel option * func list
```

For debugging purposes, the `Ast` module contains a serializer to produce string output from the AST.

4.4 Semantic Analyzer

The AST corresponds to correct syntax, but contains little type or other semantic information. The semantic analyzer (`Sast`) walks the AST recursively to verify that types are consistent and that variables and other identifiers are properly in scope, and produces a semantically-checked AST (SAST) that is significantly closer to a form that can be used to generate code. For example, since each operator-expression node of the SAST has a type stored in the node, the back end translator knows that it has to issue an instruction putting the result of the operator into a register of that particular type (the PTX assembly language, which is actually a machine-independent language for a low-level VM, has typed virtual registers and instructions).

The SAST is implemented with the following types.

```
type memspace = Local | Shared | Global | Param

type binop =
    Mult    | Div     | Plus    | Minus   | Mod
  | Lshift  | Rshift  | BitOr   | BitAnd  | BitXor
  | Equal   | NotEq   | LessEq  | Less    | GreaterEq | Greater
  | Map     | Reduce  | Scanl   | Scanr   | And       | Or

type unop =
    Not | Negative | BitComplement

type loom_type =
    Bool | Void | Int32 | Int64 | Uint32 | Uint64 | Float32 | Float64
  | Array     of loom_type * expr
  | Record    of (string * loom_type) list
  | Function  of (loom_type list) * loom_type

and expr =
    IntLiteral    of int
  | FloatLiteral  of float
  | BoolLiteral   of bool
  | Convert       of expr_t * loom_type
  | Place         of place
  | Binop         of expr_t * binop * expr_t
  | Unop          of unop * expr_t
```

```
  | Call          of fdecl * expr_t list
  | NullExpr

and place =
  | Aref          of vdecl * expr_t
  | Saref         of vdecl * expr_t
  | Rref          of vdecl * string
  | Id            of decl

and expr_t = expr * loom_type
and place_t = place * loom_type

and tdecl = string * loom_type
and vdecl = string * loom_type
and fdecl = string * (loom_type list) * loom_type
and decl =    Var   of vdecl
            | Func  of fdecl
            | Const of vdecl

type stmt =
    Block   of stmt list
  | Vdecl   of memspace * vdecl * expr_t option
  | Tdecl   of tdecl
  | Expr    of expr_t
  | Return  of expr_t
  | Assign  of place_t * expr_t
  | If      of expr_t * stmt list * stmt list
  | For     of vdecl * expr_t * expr_t * stmt list
  | While   of expr_t * stmt list
  | Break
  | Continue
  | Sync
  | NullStmt

type func = {
  fname     : string;
  formals   : vdecl list;
  ret_type  : loom_type;
  body      : stmt list;
}

type kernel = {
  kname    : string;
  kformals : vdecl list;
  kbody    : stmt list;
}

type program = kernel * func list
```

A noteworthy aspect of the SAST is that it carries a great deal of additional type information in the `expr_t` subtypes scattered throughout.

4.5 Code Generator

The code generator (`compiler.mli` and `compiler.ml`) is currently incomplete, as described in the 'BUGS' file in the source directory. The essential idea, however, is to walk the SAST and convert it into a final low-level intermediate form that can be trivially printed out to text. The intermediate form consists of a list of directives, declarations and instructions that are assembled with reference to an environment structure that accompanies the traversal of the SAST. The intermediate form has the following interface:

```
type ptx_t = Pred | Void | S32 | S64 | U32 | U64
           | F32 | F64 | B32 | B64 | U16

type memspace = Local | Shared | Global | Param | Reg

type ptx_line =
    Direct      of  string
  | Entry       of  string * (ptx_t * string) list
  | Func        of  ptx_t * string * (ptx_t * string) list
  | Decl        of  memspace * ptx_t * string
  | Label       of  string
  | Branch      of  string option * string
  | Op2         of  string * ptx_t * string * string
  | Op3         of  string * ptx_t * string * string * string
  | Convert     of  ptx_t * ptx_t * string * string
  | Load        of  memspace * ptx_t * string * string * int
  | Store       of  memspace * ptx_t * string * int * string

type ptx = ptx_line list
```

Chapter 5

Planning, Testing and Lessons Learned

5.1 General Comments

As a general matter plan, I implemented the `ocamllex` language and `ocamlyacc` language first to test the parsability of the syntax, sketched out the intermediate form data structures, implemented a simple top-level interface to facilitate testing, and then proceeded through the compilation modules more or less in order.

With more time I would have set up one of the standard Caml unit-testing frameworks, but given the limited time available, I'm not sure the overhead of extensive unit testing would have been justified. Instead, I had to rely on the Caml typechecker and incremental testing with small examples, including extensive use of the Caml REPL.

It had originally been my plan to implement a minimal, stripped-down language first and then built features gradually, and that is indeed how I am developing the back end.

A lot could be said about the advantages of schedules, milestones, and deadlines for projects, but the reality is that they tend not to be compatible with having an unrelated job.

5.2 GPU Device Testing

The back end is one area where more formal testing will be essential. Testing GPGPU kernels is typically done by implementing routines in (single-threaded) C on the CPU, using these routines to compute 'gold standard' data output, and then comparing the results to results computed on the GPGPU device. Doing so allows for a greater range of testing possibilities (including testing processing of large or randomly-generated arrays), but does depend on the correctness of the serial version.

5.3 Lessons Learned

1. Caml works well as a compiler implementation language, although the syntax seems off in some respects. For example, nested pattern matching is awkward.

2. The LOOM language seemed simple enough when I designed it, so I was surprised at the steady pace at which complications and special cases came up. I would definitely have chosen an even simpler language if I were to start over – the bare minimum to make the point, with fewer types, operators and control structures. I would certainly not have included record types, for example, which add significantly to the complexity of the code but do not really get at the core ideas of parallelism.

3. I'm not convinced that a language like LOOM is the best solution to the problem of simplifying parallel programming for the GPGPU. Even with a higher-level language for the kernel, there is still a great deal of complexity remaining in C code on the CPU. The best approach would be a single language for solving computational problems with separate back end translators for the CPU and GPGPU, in which the two parts would work together transparently. For example, a reduce operation might be executed on the GPGPU to reduce an array by a factor of 512 (the maximum size of a cooperative thread unit), with subsequent reduction handled by the CPU (and with no need to write two pieces of code in two different languages to do essentially the same thing).

Chapter 6

Source Listings

BUGS

```
*   Preprocessor doesn't signal error in the case of inconsistent indentation --
    in particular, if a statement is indented less than the previous statement
    (but not all the way back to the last proper indentation level), the
    enclosing block is not terminated and no error is signaled.

*   Parser and typechecker / semantic analyzer work, but
    - currently support only one-dimensional arrays and
    - do not recognize array indexing by array range types.

*   Literals are of Ocaml type int (signed 31-bit) and float (32-bit). This
    means that if a large constant is read into a LOOM Int64, for example,
    it may overflow.  Floating-point literals cannot be in exponent format.

*   Compiler back-end only partially implemented.  Still needs a fair amount
    of work to create a minimal working subset of the LOOM language.  Some
    fragments implemented, and most datatypes and utility functions in place.

*   Testing framework not complete (needs at least a minimal subset of
    the working compiler back-end for full LOOM -> PTX -> executable testing).

*   Command-line interface supports input in LOOM or preprocessed code, and
    output in preprocessed code, pretty-printed AST, and (incomplete) PTX.
    Some other potentially useful combinations involving intermediate forms
    are not implemented.  E.g., the program cannot currently read in the
    AST or typed AST directly, or write out the typed AST except by passing
    to the compiler back-end.
```

README

```
If the usual OCaml binaries are installed and visible in the path, it should
be possible to compile LOOM by typing './setup.sh' in this directory.

Once that's done, type './loom -help' for options.
```

setup.sh

```
#!/bin/sh
make clean
ocamldep *.mli *.ml > depend.mk
make
```

Makefile

```
OBJS = utx.cmo preprocessor.cmo scanner.cmo parser.cmo \
       ast.cmo sast.cmo compiler.cmo loom.cmo
EXTRAS = str.cma

loom: $(OBJS)
       ocamlc -o loom $(EXTRAS) $(OBJS)

scanner.ml: scanner.mll parser.cmo
       ocamllex scanner.mll

parser.ml: parser.mly ast.cmo
       ocamlyacc parser.mly
       ocamlc -c parser.mli

%.cmo: %.ml
       ocamlc -c $(EXTRAS) $<

%.cmi: %.mli
       ocamlc -c $(EXTRAS) $<

clean:
       rm -f loom scanner.ml parser.ml parser.mli parser.output
       rm -f *.cm[io]

# Need to run 'ocamldep *.mli *.ml > depend.mk'

include depend.mk
```

loom.ml

```
(** loom.ml -- top-level command-line interface *)

let version = "LOOM␣version␣0.1"

type filetype = Source | Pre | Parsed | Checked | Ptx

(** Attempt to determine a file's type based on its extension *)
let infer_filetype filename =
  let parts = Str.split (Str.regexp "[.]") filename in
  let extension = if List.length parts > 1 then
                    List.hd (List.rev parts)
                  else failwith "No␣file␣extension␣--␣can't␣determine␣type."
    in
    match extension with
      "lm"  -> Source
```

```
    | "lmp" -> Pre
    | "lma" -> Parsed
    | "lms" -> Checked
    | "ptx" -> Ptx
    | _     -> failwith ("Unknown␣extension:␣" ^ extension)

(** Print string corresponding to file type *)
let string_of_filetype = function
  Source  -> "LOOM␣source␣code"
| Pre     -> "preprocessed␣LOOM␣code"
| Parsed  -> "parsed␣LOOM␣code"
| Checked -> "semantically␣analyzed,␣typechecked␣LOOM␣code"
| Ptx     -> "PTX␣code"

(* Parse arguments *)

let usage = version ^ "\nusage:␣" ^ Sys.argv.(0)
            ^ "␣[-p␣|␣-a␣|␣-s␣|␣-c]␣[-in␣<input>.<ext>]␣[-out␣<output>.<ext>]"

(* Default values *)
let in_fname     = ref ""   (* stdin *)
let out_fname    = ref ""   (* stdout *)
let in_filetype  = ref Source
let out_filetype = ref Ptx

let arguments = [
  ("-p",   Arg.Unit (fun () -> out_filetype := Pre),     ":␣preprocess␣input"   )
; ("-a",   Arg.Unit (fun () -> out_filetype := Parsed),  ":␣parse␣input"        )
; ("-s",   Arg.Unit (fun () -> out_filetype := Checked), ":␣typecheck␣input"    )
; ("-c",   Arg.Unit (fun () -> out_filetype := Ptx),     ":␣compile␣to␣PTX␣code")
; ("-in",  Arg.String (fun s -> in_fname  := s),         ":␣set␣input␣filename" )
; ("-out", Arg.String (fun s -> out_fname := s),         ":␣set␣output␣filename")
; ("-version",
        Arg.Unit (fun () -> print_endline version; exit 0),
        ":␣print␣version␣string␣and␣exit")
]

(* main *)

let _ =

  let onerr = fun arg -> raise (Arg.Bad ("Unknown␣argument:␣" ^ arg)) in
    Arg.parse arguments onerr usage;

  if !in_fname <> ""  then in_filetype  := infer_filetype !in_fname;
  if !out_fname <> "" then out_filetype := infer_filetype !out_fname;

  (* take appropriate action depending on the input and output filetypes given *)
  match (!in_filetype, !out_filetype) with
    Source, Pre     ->  let lines = Utx.read_lines !in_fname in
                          Utx.write_string !out_fname (Preprocessor.embrace lines)

  | Source, Parsed  ->  let lines   = Utx.read_lines !in_fname in
                        let pre     = Preprocessor.embrace lines in
                        let lexbuf  = Lexing.from_string pre in
                        let parsed  = Parser.program Scanner.token lexbuf in
                          Utx.write_string !out_fname (Ast.to_string parsed)
```

```
  | Source, Checked ->  let lines   = Utx.read_lines !in_fname in
                        let pre     = Preprocessor.embrace lines in
                        let lexbuf  = Lexing.from_string pre in
                        let parsed  = Parser.program Scanner.token lexbuf in
                        let checked = Sast.analyze parsed in
                          Utx.write_string !out_fname (Sast.to_string checked)

  | Source, Ptx     ->  let lines   = Utx.read_lines !in_fname in
                        let pre     = Preprocessor.embrace lines in
                        let lexbuf  = Lexing.from_string pre in
                        let parsed  = Parser.program Scanner.token lexbuf in
                        let checked = Sast.analyze parsed in
                        let ptx     = Compiler.translate checked in
                          Utx.write_string !out_fname (Compiler.to_string ptx)

  | Pre, Parsed     ->  let pre     = Utx.read_string !in_fname in
                        let lexbuf  = Lexing.from_string pre in
                        let parsed  = Parser.program Scanner.token lexbuf in
                          Utx.write_string !out_fname (Ast.to_string parsed)

  | Pre, Checked    ->  let pre     = Utx.read_string !in_fname in
                        let lexbuf  = Lexing.from_string pre in
                        let parsed  = Parser.program Scanner.token lexbuf in
                        let checked = Sast.analyze parsed in
                          Utx.write_string !out_fname (Sast.to_string checked)

  | Pre, Ptx        ->  let pre     = Utx.read_string !in_fname in
                        let lexbuf  = Lexing.from_string pre in
                        let parsed  = Parser.program Scanner.token lexbuf in
                        let checked = Sast.analyze parsed in
                        let ptx     = Compiler.translate checked in
                          Utx.write_string !out_fname (Compiler.to_string ptx)

  | _               ->  failwith ("*** Don't know how to translate "
                                  ^ string_of_filetype !in_filetype ^ "into"
                                  ^ string_of_filetype !out_filetype ^ ".\n")
```

utx.mli

```
(** Module Utx
    Utility functions used throughout LOOM compiler *)

(** Return a list of strings corresponding to lines read in from an input
    channel. Example: to read from stdin, use 'read_lines stdin'. *)
val read_lines_from_channel : in_channel -> string list

(** Write a list of strings to output channel. *)
val write_lines_to_channel : out_channel -> string list -> unit

(** Return a list of strings corresponding to lines read in from an input
    file. Examples: to read from stdin, use 'read_lines ""'. To read from
    a file called "foo.txt", use 'read_lines "foo.txt"'. *)
val read_lines : string -> string list

(** Given a filename, read in a string.  If argument is "", use stdin. *)
val read_string : string -> string
```

```
(** Given a filename, write a list of strings to the file.  If argument is "",
    use stdout. *)
val write_lines : string -> string list -> unit

(** Given a filename, write string to the file.  If argument is "", use stdout. *)
val write_string : string -> string -> unit

(** split3: analog of List.split for lists of triples *)
val split3 : ('a * 'b  * 'c) list -> 'a list * 'b list * 'c list
```

utx.ml

```
let read_lines_from_channel chan =
  let rec read_lines' lines =
    try
      let line = input_line chan in
        read_lines' (line :: lines)
    with
      End_of_file -> List.rev lines
    in
    read_lines' []
;;

let write_lines_to_channel chan lines =
  List.iter (function line -> output_string chan (line ^ "\n")) lines
;;

let read_lines filename =
  if filename = "" then
    read_lines_from_channel stdin
  else
    let inchan = open_in filename in
    let lines = read_lines_from_channel inchan in
      close_in inchan;
      lines
;;

let read_string filename =
  let lines = read_lines filename in
    String.concat "\n" lines
;;

let write_lines filename lines =
  if filename = "" then
    write_lines_to_channel stdout lines
  else
    let outchan = open_out filename in
      write_lines_to_channel outchan lines;
      close_out outchan
;;

let write_string filename str =
  if filename = "" then
    output_string stdout str
  else
```

```
    let outchan = open_out filename in
      output_string outchan str;
      close_out outchan
;;

let split3 list_of_triples =
  let f = (fun (x,y,z) (xs,ys,zs) -> (x::xs, y::ys, z::zs)) in
    List.fold_right f list_of_triples ([],[],[])
;;
```

preprocessor.mli

```
(** Module Preprocessor *)

(** Add braces and semicolons to a list of strings holding LOOM lines of code. *)
val embrace : string list -> string
```

preprocessor.ml

```
(** preprocessor.ml
    Convert a source file from pseudocode style, where block structure is
    indicated by indentation level, to braces-and-semicolon style suitable
    for input to the compiler front-end. Also strip out comments and blank lines.
    NB - inserts a (harmless) empty statement on the last line. *)

(** Count the number of spaces before the first non-space character in str.
    (NB - Caml doesn't seem to encourage functional-style string manipulation,
    so fall back on refs and while-loops.) *)
let leading_spaces str =
  let count = ref 0 in
    while str.[!count] = ' ' do
      count := !count + 1
    done;
    !count;;

(** Strip off trailing spaces from str. *)
let right_strip str =
  let i = ref (String.length str) in
    while !i > 0 && str.[!i - 1] = ' ' do
      i := !i - 1
    done;
    String.sub str 0 !i

(** Given a list of indentation levels (typically just been popped from the
    indentation stack), terminate the preceding statement with a semicolon
    and close out open blocks at each level specified. *)
let close_braces indents =
  let braces =
    List.fold_left (fun str n -> str ^ "\n" ^ String.make n ' ' ^ "}") "" indents in
  ";" ^ braces ^ "\n";;

(** Given a state object representing prior lines, add braces and semicolons
    to the next line. *)
let embrace_line (stack, acc) line =
  let line' =                       (* line' is line sans comments *)
```

```
    match Str.bounded_split (Str.regexp "--") line 2 with
      str :: _  -> right_strip str
    | _         -> ""
  in
  if line' = "" then
    (stack, acc)    (* ignore blank lines - pass through input state *)
  else
    let indent = leading_spaces line' in
      match stack with
        []                        -> ([0], line' :: acc)
      | (s :: _) when indent = s  -> (stack, (";\n" ^ line') :: acc)
      | (s :: _) when indent > s  -> (indent :: stack, ("␣{\n" ^ line') :: acc)
      | (s :: rest)               ->
        let (to_pop, reduced) = List.partition (fun x -> x >= indent) rest in
        let closes = close_braces to_pop in
          (indent :: reduced, (closes ^ line') :: acc);;

let embrace lines =
  let lines' = lines @ ["X\n"] in  (* HACK: add a placeholder at the end *)
  let (_, backward_result) = List.fold_left embrace_line ([], []) lines' in
  let s = String.concat "" (List.rev backward_result) in
    String.sub s 0 ((String.length s) - 2)  (* HACK: delete the placeholder *)
```

scanner.mll

```
{ open Parser }

let letter = ['a'-'z' 'A'-'Z']
let digit  = ['0'-'9']
let idchar = ['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']

rule token = parse
    [' ' '\t' '\n']       { token lexbuf }      (* whitespace *)
  | "--" [^ '\n']* '\n'   { token lexbuf }      (* comment *)
  | '('                   { LPAREN }
  | ')'                   { RPAREN }
  | "[["                  { LDOUBLE }
  | "]]"                  { RDOUBLE }
  | '['                   { LSQUARE }
  | ']'                   { RSQUARE }
  | '{'                   { LCURLY }
  | '}'                   { RCURLY }
  | ';'                   { SEMI }
  | "<-"                  { ASSIGN }
  | "<<"                  { LSHIFT }
  | ">>"                  { RSHIFT }
  | '*'                   { STAR }
  | '/'                   { SLASH }
  | '%'                   { MOD }
  | '+'                   { PLUS }
  | '-'                   { MINUS }
  | '|'                   { VBAR }
  | '^'                   { CARET }
  | '~'                   { TILDE }
  | '&'                   { AMP }
  | '='                   { EQ }
  | "!="                  { NE }
```

```
  | "<="                  { LE }
  | ">="                  { GT }
  | '<'                   { LT }
  | '>'                   { GT }
  | ':'                   { COLON }
  | ','                   { COMMA }
  | "::"                  { CONVERT }
  | '.'                   { DOT }
  | ".."                  { TWODOTS }
  | "..."                 { THREEDOTS }
  | "//"                  { MAP }
  | "/."                  { REDUCE }
  | "/:"                  { SCANL }
  | ":/"                  { SCANR }
  | "and"                 { AND }
  | "break"               { BREAK }
  | "continue"            { CONTINUE }
  | "else"                { ELSE }
  | "if"                  { IF }
  | "for"                 { FOR }
  | "func"                { FUNC }
  | "global"              { GLOBAL }
  | "kernel"              { KERNEL }
  | "local"               { LOCAL }
  | "not"                 { NOT }
  | "or"                  { OR }
  | "pass"                { PASS }
  | "return"              { RETURN }
  | "shared"              { SHARED }
  | "sync"                { SYNC }
  | "then"                { THEN }
  | "type"                { TYPE }
  | "while"               { WHILE }
  (* Basic types and constant literals *)
  | "Bool"                { BOOL }
  | "Void"                { VOID }
  | "Int32"               { INT32 }
  | "Int64"               { INT64 }
  | "Uint32"              { UINT32 }
  | "Uint64"              { UINT64 }
  | "Float32"             { FLOAT32 }
  | "Float64"             { FLOAT64 }
  | "TRUE"                { TRUE }
  | "FALSE"               { FALSE }
  | eof                   { EOF }
  | digit+ as inum        { INTLITERAL(int_of_string inum) } (* pos. integers *)
  | digit+ '.' digit+ as fnum { FLOATLITERAL(float_of_string fnum) }
  | letter idchar* as id  { ID(id) }
  | _ as bogus            { raise (Failure("Bad token: " ^ Char.escaped bogus)) }
```

parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN LDOUBLE RDOUBLE LSQUARE RSQUARE LCURLY RCURLY
%token SEMI COMMA DOT TWODOTS THREEDOTS COLON TYPE
%token ASSIGN MAP REDUCE SCANL SCANR
```

```
%token OR AND EQ NE LE GE LT GT
%token PLUS MINUS VBAR CARET
%token STAR SLASH MOD LSHIFT RSHIFT AMP TILDE NOT CONVERT
%token ASSIGN LSHIFT RSHIFT STAR SLASH MOD PLUS MINUS VBAR CARET
%token FOR WHILE BREAK CONTINUE IF THEN ELSE PASS
%token FUNC KERNEL RETURN GLOBAL LOCAL SHARED SYNC
%token BOOL VOID INT32 INT64 UINT32 UINT64 FLOAT32 FLOAT64
%token TRUE FALSE EOF
%token <int> INTLITERAL
%token <float> FLOATLITERAL
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc UMINUS
%right ASSIGN
%right MAP REDUCE SCANL SCANR
%left OR
%left AND
%left EQ NE LE GE LT GT
%left PLUS MINUS VBAR CARET
%left STAR SLASH MOD LSHIFT RSHIFT AMP
%nonassoc TILDE NOT
%left CONVERT

%start program
%type <Ast.program> program

%%

program:
    /* empty program */   { None, [] }
  | program kernel        { Some($2), snd $1 }
  | program func          { fst $1, ($2 :: snd $1) }

kernel:  KERNEL ID LPAREN args_opt RPAREN block
    { { kname     = $2;
        kformals  = $4;
        kbody     = $6 } }

func:  FUNC ID LPAREN args_opt RPAREN COLON ltype block
    { { fname     = $2;
        formals   = $4;
        ret_type  = $7;
        body      = $8 } }

args_opt:
    /* nothing */     { [] }
  | args_list { List.rev $1 }

args_list:
    ID COLON ltype                 { [($1, $3)] }
  | args_list COMMA ID COLON ltype { ($3, $5) :: $1 }

block:  LCURLY stmt_list RCURLY   { List.rev $2 }

stmt_list:
```

```
    /* nothing */     { [] }
  | stmt_list stmt    { $2 :: $1 }

stmt:
    block                                   { Block($1) }
  | expr SEMI                               { Expr($1) }
  | memspace ID COLON ltype ASSIGN expr SEMI { Vdecl($1, $2, $4, Some($6)) }
  | memspace ID COLON ltype SEMI            { Vdecl($1, $2, $4, None) }
  | TYPE ID ltype SEMI                      { Tdecl($2, $3) }
  | place ASSIGN expr SEMI                  { Assign($1, $3) }
  | IF expr THEN block                      { If($2, $4, []) }
  | IF expr THEN block ELSE block           { If($2, $4, $6) }
  | FOR ID COLON ltype ASSIGN expr TWODOTS expr block
             { For($2, $4, $6, $8, $9) }
  | FOR ID COLON ltype ASSIGN expr THREEDOTS expr block
             { For($2, $4, $6, Binop(IntLiteral(1),Plus,$8), $9) }
  | WHILE expr block                        { While ($2, $3) }
  | BREAK SEMI                              { Break }
  | CONTINUE SEMI                           { Continue }
  | RETURN expr SEMI                        { Return($2) }
  | RETURN SEMI                             { Return(NullExpr) }
  | SYNC SEMI                               { Sync }
  | PASS SEMI                               { NullStmt }

place:
    ID LSQUARE expr RSQUARE       { Aref($1, $3) }
  | ID LDOUBLE expr RDOUBLE       { Saref($1, $3) }
  | ID DOT ID                     { Rref($1, $3) }
  | ID                            { Id($1) }

expr:
    INTLITERAL                    { IntLiteral($1) }
  | FLOATLITERAL                  { FloatLiteral($1) }
  | TRUE                          { BoolLiteral(true) }
  | FALSE                         { BoolLiteral(false) }
  | LPAREN expr RPAREN            { $2 }
  | place                         { Place($1) }
  | expr CONVERT ltype            { Convert($1, $3) }
  | expr STAR expr                { Binop($1, Mult, $3) }
  | expr SLASH expr               { Binop($1, Div, $3) }
  | expr PLUS expr                { Binop($1, Plus, $3) }
  | expr MINUS expr               { Binop($1, Minus, $3) }
  | expr MOD expr                 { Binop($1, Mod, $3) }
  | expr LSHIFT expr              { Binop($1, Lshift, $3) }
  | expr RSHIFT expr              { Binop($1, Rshift, $3) }
  | expr VBAR expr                { Binop($1, BitOr, $3) }
  | expr AMP expr                 { Binop($1, BitAnd, $3) }
  | expr CARET expr               { Binop($1, BitXor, $3) }
  | expr EQ expr                  { Binop($1, Equal, $3) }
  | expr NE expr                  { Binop($1, NotEq, $3) }
  | expr LE expr                  { Binop($1, LessEq, $3) }
  | expr LT expr                  { Binop($1, Less, $3) }
  | expr GE expr                  { Binop($1, GreaterEq, $3) }
  | expr GT expr                  { Binop($1, Greater, $3) }
  | expr MAP expr                 { Binop($1, Map, $3) }
  | expr REDUCE expr              { Binop($1, Reduce, $3) }
  | expr SCANR expr               { Binop($1, Scanr, $3) }
```

```
  | expr SCANL expr               { Binop($1, Scanl, $3) }
  | NOT expr                      { Unop(Not, $2) }
  | MINUS expr %prec UMINUS       { Unop(Negative, $2) }
  | TILDE expr                    { Unop(BitComplement, $2) }
  | ID LPAREN actuals_opt RPAREN  { Call($1, $3) }

actuals_opt:
    /* nothing */   { [] }
  | actuals_list    { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }

memspace:
    LOCAL   { Local }
  | GLOBAL  { Global }
  | SHARED  { Shared }

ltype:
    BOOL    { Bool }
  | VOID    { Void }
  | INT32   { Int32 }
  | INT64   { Int64 }
  | UINT32  { Uint32 }
  | UINT64  { Uint64 }
  | FLOAT32 { Float32 }
  | FLOAT64 { Float64 }
  | ltype LSQUARE expr RSQUARE  { Array($1, $3) }
  | LCURLY fields_opt RCURLY    { Record($2) }
  | ID                          { Type($1) }

fields_opt:
    /* nothing */     { [] }
  | fields_list { List.rev $1 }

fields_list:
    ID COLON ltype                    { [($1, $3)] }
  | fields_list COMMA ID COLON ltype { ($3, $5) :: $1 }
```

ast.ml

```
(** Module Ast *)

type memspace = Local | Shared | Global

type binop =
    Mult    | Div     | Plus    | Minus   | Mod
  | Lshift  | Rshift  | BitOr   | BitAnd  | BitXor
  | Equal   | NotEq   | LessEq  | Less    | GreaterEq | Greater
  | Map     | Reduce  | Scanl   | Scanr   | Or        | And

type unop =
    Not | Negative | BitComplement

type loom_type =
    Bool | Void | Int32 | Int64 | Uint32 | Uint64 | Float32 | Float64
```

```
  | Array   of loom_type * expr
  | Record  of (string * loom_type) list
  | Type    of string  (* to be associated with a loom_type later *)

and expr =
    IntLiteral    of int
  | FloatLiteral  of float
  | BoolLiteral   of bool
  | Convert       of expr * loom_type
  | Place         of place
  | Binop         of expr * binop * expr
  | Unop          of unop * expr
  | Call          of string * expr list
  | NullExpr

and place =
    Aref          of string * expr
  | Saref         of string * expr
  | Rref          of string * string
  | Id            of string

type stmt =
    Block   of stmt list
  | Vdecl   of memspace * string * loom_type * expr option
  | Tdecl   of string * loom_type
  | Expr    of expr
  | Return  of expr
  | Assign  of place * expr
  | If      of expr * stmt list * stmt list
  | For     of string * loom_type * expr * expr * stmt list
  | While   of expr * stmt list
  | Break
  | Continue
  | Sync
  | NullStmt

type func = {
  fname    : string;
  formals  : (string * loom_type) list;
  ret_type : loom_type;
  body     : stmt list;
}

type kernel = {
  kname    : string;
  kformals : (string * loom_type) list;
  kbody    : stmt list;
}

type program = kernel option * func list

(* -- String conversion functions -- *)

let string_of_memspace =
  function
    Local -> "local" | Global -> "global" | Shared -> "shared"
```

```
let string_of_binop = function
    Mult -> "*" | Div -> "/" | Plus -> "+" | Minus -> "-" | Mod -> "%"
  | Lshift -> "<<" | Rshift -> ">>" | BitOr -> "|" | BitAnd -> "&" | BitXor -> "^"
  | Equal -> "=" | NotEq -> "!=" | LessEq -> "<=" | Less -> "<" | GreaterEq -> ">="
  | Greater -> ">" | Map -> "//" | Reduce -> "/." | Scanl -> "/:" | Scanr -> ":/"
  | And -> "&&" | Or -> "||"

let string_of_unop = function
  Not -> "!" | Negative -> "-" | BitComplement -> "~"

let rec string_of_type =
  let str_field (s,t) = s ^ ":␣" ^ string_of_type t in
    function
      Bool -> "Bool" | Void -> "Void" | Int32 -> "Int32" | Int64 -> "Int64"
    | Uint32 -> "Uint32" | Uint64 -> "Uint64"
    | Float32 -> "Float32"| Float64 -> "Float64"
    | Array(t,e)      ->  string_of_type t ^ "[" ^ string_of_expr e ^ "]"
    | Record(fields)  ->  "{" ^ String.concat ",␣" (List.map str_field fields)
                          ^ "}"
    | Type(name)      ->  name

and string_of_expr = function
    IntLiteral(n)   ->  string_of_int n
  | FloatLiteral(x) ->  string_of_float x
  | BoolLiteral(p)  ->  if p then "true" else "false"
  | Convert(e,t)    ->  "(" ^ string_of_expr e ^ ")␣::␣" ^ string_of_type t
  | Place(p)        ->  string_of_place p
  | Binop(e1,op,e2) ->  "(" ^ string_of_expr e1 ^ "␣"
                        ^ string_of_binop op ^ "␣"
                        ^ string_of_expr e2 ^ ")"
  | Unop(op,e)      ->  "(" ^ string_of_unop op ^ "␣" ^ string_of_expr e ^ ")"
  | Call(f,args)    ->  f ^ "("
                        ^ String.concat ",␣" (List.map string_of_expr args)
                        ^ ")"
  | NullExpr        ->  ""

and string_of_place = function
    Aref(a,e)       ->  a ^ "[" ^ string_of_expr e ^ "]"
  | Saref(a,e)      ->  a ^ "[[" ^ string_of_expr e ^ "]]"
  | Rref(v,x)       ->  v ^ "." ^ x
  | Id(s)           ->  s

(* FIXME - should add hierarchical indentation for blocks -- would need to
   pass around indentation parameter everywhere, though *)

let rec string_of_stmt = function
    Block(stmts)      ->  "{\n"
                          ^ String.concat "" (List.map string_of_stmt stmts)
                          ^ "}\n"
  | Vdecl(m,v,t,e)    ->  string_of_memspace m ^ "␣" ^ v ^ ":␣" ^ string_of_type t
                          ^ (match e with
                               Some(e') -> "␣<-␣" ^ string_of_expr e'
                             | None     -> "")
                          ^ ";\n"
  | Tdecl(name,t)     ->  "type␣" ^ name ^ string_of_type t ^ ";\n"
  | Expr(e)           ->  string_of_expr(e) ^ ";\n"
  | Return(e)         ->  "return␣" ^ string_of_expr e ^ ";\n"
```

```
  | Assign(p,e)       ->  string_of_place p ^ "␣<-␣" ^ string_of_expr e ^ ";\n"
  | If(e,ss,[])       ->  "if␣" ^ string_of_expr e ^ "␣then␣{\n"
                          ^ String.concat "" (List.map string_of_stmt ss)
                          ^ "}\n"
  | If(e,ss,ss')      ->  "if␣" ^ string_of_expr e ^ "␣then␣{\n"
                          ^ String.concat "" (List.map string_of_stmt ss)
                          ^ "}\nelse␣{\n"
                          ^ String.concat "" (List.map string_of_stmt ss')
                          ^ "}\n"
  | For(v,t,e1,e2,ss) ->  "for␣" ^ v ^ ":␣" ^ string_of_type t ^ "␣<-␣"
                          ^ string_of_expr e1 ^ "␣..␣" ^ string_of_expr e2 ^ "\n"
                          ^ String.concat "" (List.map string_of_stmt ss)
                          ^ "}\n"
  | While(e,ss)       ->  "while␣" ^ string_of_expr e
                          ^ String.concat "" (List.map string_of_stmt ss)
                          ^ "}\n"
  | Break             ->  "break;\n"
  | Continue          ->  "continue;\n"
  | Sync              ->  "sync;\n"
  | NullStmt          ->  "\n"
;;

let string_of_formal (name, t) =
  name ^ ":␣" ^ string_of_type t

let string_of_func f =
  "func␣" ^ f.fname ^ "␣("
  ^ String.concat ",␣" (List.map string_of_formal f.formals)
  ^ "):␣" ^ string_of_type f.ret_type ^ "␣{\n"
  ^ String.concat "" (List.map string_of_stmt f.body)
  ^ "}\n"

let string_of_kernel k =
  "kernel␣" ^ k.kname ^ "␣("
  ^ String.concat ",␣" (List.map string_of_formal k.kformals)
  ^ ")␣{\n"
  ^ String.concat "" (List.map string_of_stmt k.kbody)
  ^ "}\n"

let to_string (k,fs) =
  String.concat "" (List.map string_of_func fs)
  ^ match k with
      Some(k')  ->  string_of_kernel k'
    | None      ->  "--␣Warning:␣no␣kernel␣supplied\n"
```

sast.ml

```
(** Module Sast -- semantic analyzer / typechecker *)

type memspace = Local | Shared | Global | Param

type binop =
    Mult    | Div     | Plus    | Minus   | Mod
  | Lshift  | Rshift  | BitOr   | BitAnd  | BitXor
  | Equal   | NotEq   | LessEq  | Less    | GreaterEq | Greater
  | Map     | Reduce  | Scanl   | Scanr   | And       | Or
```

```
type unop =
    Not | Negative | BitComplement

type loom_type =
    Bool | Void | Int32 | Int64 | Uint32 | Uint64 | Float32 | Float64
  | Array     of loom_type * expr
  | Record    of (string * loom_type) list
  | Function  of (loom_type list) * loom_type

and expr =
    IntLiteral    of int
  | FloatLiteral  of float
  | BoolLiteral   of bool
  | Convert       of expr_t * loom_type  (* in Sast, holds the SOURCE type *)
  | Place         of place
  | Binop         of expr_t * binop * expr_t
  | Unop          of unop * expr_t
  | Call          of fdecl * expr_t list
  | NullExpr

and place =
  | Aref          of vdecl * expr_t
  | Saref         of vdecl * expr_t
  | Rref          of vdecl * string
  | Id            of decl

and expr_t = expr * loom_type
and place_t = place * loom_type

and tdecl = string * loom_type    (* type name, type *)
and vdecl = string * loom_type    (* var name, type *)
and fdecl = string * (loom_type list) * loom_type  (* name, args, return type *)
and decl =    Var   of vdecl
            | Func  of fdecl
            | Const of vdecl  (* used from thread constants *)

type stmt =
    Block   of stmt list
  | Vdecl   of memspace * vdecl * expr_t option
  | Tdecl   of tdecl
  | Expr    of expr_t
  | Return  of expr_t
  | Assign  of place_t * expr_t
  | If      of expr_t * stmt list * stmt list
  | For     of vdecl * expr_t * expr_t * stmt list
  | While   of expr_t * stmt list
  | Break
  | Continue
  | Sync
  | NullStmt

type func = {
  fname     : string;
  formals   : vdecl list;
  ret_type  : loom_type;
  body      : stmt list;
}
```

```
type kernel = {
  kname    : string;
  kformals : vdecl list;
  kbody    : stmt list;
}

type program = kernel * func list

let thread_constants =
    [     "I" ;   "THREADS" ;     "B" ;   "BLOCKS"  ;   "BLOCKSIZE"
    ;   "X_I" ; "X_THREADS" ;   "X_B" ; "X_BLOCKS"  ; "X_BLOCKSIZE"
    ;   "Y_I" ; "Y_THREADS" ;   "Y_B" ; "Y_BLOCKS"  ; "Y_BLOCKSIZE" ]

(* -- environment types and utility functions -- *)

type symbols = {
  parent          : symbols option
; mutable vars    : (vdecl * memspace) list
; mutable types   : tdecl list
; mutable funcs   : fdecl list
}

type environment = {
  func_name       : string
; return_type     : loom_type
; loop_level      : int
; scope           : symbols
}

(** Look up a variable name in a symbol table, and return (vdecl, memspace) if
    found.  Otherwise, follow links upward in the scope chain.  If the variable
    name does not appear anywhere in the chain, raise an exception. *)
let rec lookup_var_full scope vname =
  try
    List.find (fun ((s, _),_) -> s = vname) scope.vars
  with Not_found ->
    match scope.parent with
      Some(p) -> lookup_var_full p vname
    | None    -> raise Not_found
;;

(** Look up a variable name in a symbol table, and return vdecl if found. *)
let lookup_var scope vname =
  let (vd, _) = lookup_var_full scope vname in
    vd
;;

(** Look up a function name, and return fdecl if found. *)
(* FIXME: because LOOM functions have global scope, this will always go to the top
   level -- could be optimized by keeping fdecls in a separate global record *)
let rec lookup_func scope fname =
  try
    List.find (fun (s, _, _) -> s = fname) scope.funcs
  with Not_found ->
    match scope.parent with
      Some(p) -> lookup_func p fname
```

```
    | None    -> raise Not_found
;;

(** Look up a type name in scope, and return tdecl if found. *)
let rec lookup_type scope tname =
  try
    List.find (fun (s, _) -> s = tname) scope.types
  with Not_found ->
    match scope.parent with
      Some(p) -> lookup_type p tname
    | None    -> raise Not_found
;;

(** Create a string containing a summary of the scope (for error messages) *)
let rec string_of_scope scope =
  let vnames  = List.map (fun ((name,_),_)  -> name) scope.vars   in
  let tnames  = List.map (fun (name,_)      -> name) scope.types  in
  let fnames  = List.map (fun (name,_,_)    -> name) scope.funcs  in
  let parent_info =
    match scope.parent with
      Some(p) -> "\nPARENT " ^ (string_of_scope p)
    | None    -> "(EARLIEST ANCESTOR)\n"
  in
    "SCOPE:"
    ^ "\n  vars: " ^ String.concat ", " vnames
    ^ "\n  types: " ^ String.concat ", " tnames
    ^ "\n  funcs: " ^ String.concat ", " fnames
    ^ "\n"
    ^ parent_info
;;

(** Convert a LOOM type int a string (for error reporting) *)
let rec string_of_type =
  let str_field (s,t) = s ^ ": " ^ string_of_type t in
    function
      Bool -> "Bool" | Void -> "Void" | Int32 -> "Int32" | Int64 -> "Int64"
    | Uint32 -> "Uint32" | Uint64 -> "Uint64"
    | Float32 -> "Float32"| Float64 -> "Float64"
    | Array(t,_)      ->  string_of_type t ^ "[...]"
    | Record(fields)  ->  "{" ^ String.concat ", " (List.map str_field fields)
                          ^ "}"
    | Function(args_t,ret_t) ->
        "Function ("
        ^ (String.concat ", " (List.map string_of_type args_t))
        ^ "): " ^ string_of_type ret_t
;;

(* -- semantic analysis -- *)

(* FIXME - probably better to just share Ast.binop and Ast.unop types *)
let check_binop = function
    Ast.Mult    -> Mult     | Ast.Div   -> Div    | Ast.Plus -> Plus
  | Ast.Minus   -> Minus    | Ast.Mod   -> Mod    | Ast.Lshift -> Lshift
  | Ast.Rshift  -> Rshift   | Ast.BitOr -> BitOr  | Ast.BitAnd -> BitAnd
  | Ast.BitXor  -> BitXor   | Ast.Equal -> Equal  | Ast.NotEq -> NotEq
  | Ast.LessEq  -> LessEq   | Ast.Less  -> Less   | Ast.GreaterEq -> GreaterEq
  | Ast.Greater -> Greater  | Ast.Map   -> Map    | Ast.Reduce -> Reduce
```

```
  | Ast.Scanl   -> Scanl    | Ast.Scanr -> Scanr
  | Ast.Or      -> Or       | Ast.And   -> And
;;

let check_unop = function
  Ast.Not -> Not | Ast.Negative -> Negative | Ast.BitComplement -> BitComplement
;;

let check_memspace = function
  Ast.Global -> Global | Ast.Local -> Local | Ast.Shared -> Shared
;;

let rec check_loom_type env = function
    Ast.Bool      -> Bool       | Ast.Void      -> Void
  | Ast.Int32     -> Int32      | Ast.Int64     -> Int64
  | Ast.Uint32    -> Uint32     | Ast.Uint64    -> Uint64
  | Ast.Float32   -> Float32    | Ast.Float64   -> Float64

  | Ast.Array(t,e)      ->
      let t'      = check_loom_type env t in
      let (e',_)  = check_expr env e in
        begin
          match e' with
            IntLiteral (_) | Place(Id(Const(_))) -> Array(t',e')
          | _   -> failwith ("\n***␣Array␣type␣declaration␣parameter␣must␣be"
                             ^ "␣either␣an␣integer␣constant␣or␣a␣thread␣constant.")
        end

  | Ast.Record(ts)      ->
      let r = List.map (fun (s,t) -> (s, check_loom_type env t)) ts in
        (* let fieldnames, _ = List.split r in *)
        (* FIXME - check no duplicate field names *)
        Record(r)

  | Ast.Type(s)         ->
      let tdecl = try
        lookup_type env.scope s
      with Not_found    ->
        failwith ("\n***␣Unknown␣type␣" ^ s)
      in
        let (_, t) = tdecl in t

(** Given an Ast expression, recursively build up a typechecked structure. *)
(* Nomenclature: "primed" variables are in Sast (e.g., e', t') *)
and check_expr env = function
    Ast.IntLiteral(n)   ->  IntLiteral(n), Int32
  | Ast.FloatLiteral(x) ->  FloatLiteral(x), Float32
  | Ast.BoolLiteral(p)  ->  BoolLiteral(p), Bool

  | Ast.Convert(e, dest_t)  ->            (* in Ast, Convert takes dest type *)
      let e', source_t' = check_expr env e in
      let dest_t' = check_loom_type env dest_t in
        Convert((e', dest_t'), source_t'), dest_t'

  | Ast.Place(p)            ->      (* have to thread the node type t' through *)
      let (p', t') = check_place env p in
        Place(p'), t'
```

```
| Ast.Binop(e1,op,e2)     ->
  begin
    let (e1', t1') as et1' = check_expr env e1 in
    let (e2', t2') as et2' = check_expr env e2 in
    let op' = check_binop op in
      match op' with

        Map -> begin
          match (t1', t2') with
            Function([arg_t], ret_t), Array(arr_t, n) ->
              if arg_t <> arr_t then
                failwith "\n***␣map:␣function␣arg␣and␣array␣types␣must␣match."
              else
                Binop(et1', op', et2'), Array(ret_t, n)
          | _ -> failwith ("\n***␣map␣operator␣must␣have␣function␣of␣one␣argument"
                           ^ "␣on␣left,␣and␣array␣on␣right")
        end

      | Reduce -> begin
          match (t1', t2') with
            Function([arg1_t; arg2_t], ret_t), Array(arr_t, n) ->
              if arg1_t <> arg2_t || arg1_t <> arr_t then
                failwith "\n***␣reduce:␣function␣arg␣and␣array␣types␣must␣match."
              else
                Binop(et1', op', et2'), ret_t
          | _ -> failwith ("\n***␣reduce␣operator␣must␣have␣function␣of␣two␣"
                           ^ "arguments␣on␣left,␣and␣array␣on␣right")
        end

      | Scanr | Scanl -> begin
          match (t1', t2') with
            Function([arg1_t; arg2_t], ret_t), Array(arr_t, n) ->
              if arg1_t <> arg2_t || arg1_t <> arr_t then
                failwith "\n***␣scan[lr]:␣function␣arg␣and␣array␣types␣must␣match."
              else
                Binop(et1', op', et2'), Array(ret_t, n)
          | _ -> failwith ("\n***␣reduce␣operator␣must␣have␣function␣of␣two␣"
                           ^ "arguments␣on␣left,␣and␣array␣on␣right")
        end

      | Mult | Div | Plus | Minus | Mod | BitOr | BitAnd | BitXor -> begin
          if t1' <> t2' then
            failwith ("\n***␣types␣for␣operands␣of␣'" ^ Ast.string_of_binop op
                      ^ "'␣must␣match␣exactly.")
          else if t1' = Bool || t1' = Void then
            failwith ("\n***␣operands␣to␣" ^ Ast.string_of_binop op
                      ^ "must␣have␣numeric␣type.")
          else
            Binop(et1', op', et2'), t1'
        end

      | Lshift | Rshift -> begin
          if t2' <> Int32 then
            failwith ("\n***␣shift␣operator␣" ^ Ast.string_of_binop op
                      ^ "␣expects␣Int32␣on␣right␣(try␣a␣type␣conversion?).")
          else if t1' = Bool || t1' = Void then
```

```
              failwith ("\n***␣operands␣to␣" ^ Ast.string_of_binop op
                        ^ "must␣have␣numeric␣type.")
            else
              Binop(et1', op', et2'), t1'
          end

        | And | Or -> begin
            if t1' <> Bool || t2' <> Bool then
              failwith ("\n***␣and␣/␣or␣operands␣must␣have␣type␣Bool.")
            else
              Binop(et1', op', et2'), Bool
          end

        | _ -> begin  (* comparison operators *)
            if t1' <> t2' then
              failwith ("\n***␣types␣for␣operands␣of␣'" ^ Ast.string_of_binop op
                        ^ "'␣must␣match␣exactly.")
            else
              Binop(et1', op', et2'), Bool
          end
    end

  | Ast.Unop(op,e)          ->
      let (e', t') as et' = check_expr env e in
      let op' = check_unop op in
        if op' = Not then
          if t' <> Bool then
            failwith "\n***␣not␣operand␣must␣have␣type␣Bool"
          else
            Unop(op', et'), Bool
        else if t' = Bool || t' = Void then
          failwith ("\n***␣operand␣to␣" ^ Ast.string_of_unop op
                    ^ "must␣have␣numeric␣type.")
        else
          Unop(op', et'), t'

  | Ast.Call(f,args)        ->
      let args' = List.map (check_expr env) args in
      let _, types' = List.split args' in
      let (fname, formals_t, ret_t) as fd =
        try
          lookup_func env.scope f
        with Not_found ->
          failwith ("\n***␣Undefined␣function␣name␣" ^ f)
      in
        (* check the types of the actual arguments against function signature *)
        if List.for_all (fun (x, y) -> x = y)
                        (List.combine types' formals_t) then
          Call(fd, args'), ret_t
        else
          failwith ("\n***␣arguments␣to␣" ^ f ^ "(...)␣do␣not␣match␣formals.")

  | Ast.NullExpr            ->  NullExpr, Void

and check_place env =
  let vlookup name =
    try
```

```
    lookup_var env.scope name
  with Not_found ->
    failwith ("\n***␣No␣variable␣or␣function␣declaration␣for␣identifier␣"
              ^ name ^ "␣in␣array/record;\nscope␣trace:␣\n"
              ^ string_of_scope env.scope)
in
function

  Ast.Aref(a,e)           ->
    let (e', t') as et' = check_expr env e in
    let (_, a_t) as vd = vlookup a in
    begin
      match a_t with
        Array(elem_t, _) -> Aref(vd, et'), t'
      | _          -> failwith ("\n***␣" ^ a ^ "␣has␣non-array␣type.")
    end

| Ast.Saref(a,e)          ->
    let (e', t') as et' = check_expr env e in
    let (_, a_t) as vd = vlookup a in
    begin
      match a_t with
        Array(elem_t, _) -> Saref(vd, et'), t'
      | _          -> failwith ("\n***␣" ^ a ^ "␣has␣non-array␣type.")
    end

| Ast.Rref(r,x)           ->
    let (_, r_t) as vd = vlookup r in
    begin
      match r_t with
        Record(fields) ->
          let t' =  try  List.assoc x fields
                    with Not_found -> failwith ("\n***␣Unknown␣field␣" ^ x)
          in
            Rref(vd, x), t'
      | _           ->  failwith ("\n***␣" ^ r ^ "␣has␣non-record␣type.")
    end

| Ast.Id(s)               ->
    let id =                    (* id has type decl *)
      if List.mem s thread_constants then
        Const(s, Int32)
      else
        try Var(lookup_var env.scope s)
        with Not_found ->
          try Func(lookup_func env.scope s)
          with Not_found ->
            failwith ("\n***␣No␣variable␣or␣function␣declaration␣for␣identifier␣"
                      ^ s ^ ";\nscope␣trace:␣\n" ^ string_of_scope env.scope)
    in
    begin
      match id with
        Var(name, t)              ->  Id(Var(name,t)), t
      | Func(name, args_t, ret_t) ->  Id(Func(name, args_t, ret_t)),
                                        Function(args_t, ret_t)
      | Const(name, t)            ->  Id(Const(name,t)), t
    end
```

```
;;

let rec check_block env stmts =
  (* create a new child scope *)
  let scope' =  { parent = Some(env.scope);
                  vars  = [];
                  funcs = [];
                  types = [] } in
  let env' = { env with scope = scope' } in
  let stmts' = List.map (fun s -> check_stmt env' s) stmts in
    stmts'

and check_stmt env = function
    Ast.Block(stmts) ->
      let stmts' = check_block env stmts in
        Block(stmts')

  | Ast.Vdecl(space, name, t, init) ->
    begin
      if List.exists (fun ((s,_),_) -> s = name) env.scope.vars then
        failwith ("\n***␣Attempt␣to␣re-declare␣variable␣" ^ name)
      else
        let t' = check_loom_type env t in
        let space' = check_memspace space in
          env.scope.vars <-
            ((name, t'), space') :: env.scope.vars;
          match init with
            Some(e) ->  let (_,init_t) as et' = check_expr env e in
                          if init_t <> t' then
                            failwith ("\n***␣Variable␣" ^ name
                                      ^ "␣initialization␣has␣incorrect␣type.")
                          else
                            Vdecl(space', (name, t'), Some(et'))
          | None    ->  Vdecl(space', (name, t'), None)
    end

  | Ast.Tdecl(name, t) ->
      if List.exists (fun (s,_) -> s = name) env.scope.types then
        failwith ("\n***␣Attempt␣to␣re-declare␣type␣" ^ name)
      else
        let t' = check_loom_type env t in
          env.scope.types <- (name, t') :: env.scope.types;
          Tdecl(name,t')  (* FIXME - could probably just be NullStmt *)

  | Ast.Expr(e)   -> Expr(check_expr env e)

  | Ast.Return(e) ->
      let (_, t') as et' = check_expr env e in
        if t' <> env.return_type then
          failwith ("\n***␣Return␣expression␣(of␣type␣"
                    ^ string_of_type t'
                    ^ ")␣must␣match␣function␣return␣type␣"
                    ^ string_of_type env.return_type ^ "␣in␣"
                    ^ env.func_name ^ ".")
        else
          Return(et')
```

```
  | Ast.Assign(p,e) ->
      let p' = check_place env p in
      let e' = check_expr env e in
        Assign(p', e')

  | Ast.If(e,stmts1,stmts2) ->
      let (_,t') as et' = check_expr env e in
        if t' <> Bool then
          failwith "\n***␣Expression␣in␣if␣statement␣must␣have␣type␣Bool."
        else
          If(et', check_block env stmts1, check_block env stmts2)

  | Ast.For(name,t,e1,e2,stmts) ->
      let t' = check_loom_type env t in
      let (_,t1') as et1' = check_expr env e1 in
      let (_,t2') as et2' = check_expr env e2 in
        if t' <> t1' || t' <> t2' || t' = Bool || t' = Void then
          failwith "\n***␣Mismatched␣or␣non-numeric␣types␣in␣for␣loop."
        else
          (* can't use check_block because we have to squeeze in index var *)
          let scope' =  { parent = Some(env.scope);
                          vars  = [((name,t'),Local)];
                          funcs = [];
                          types = [] } in
          let env' = {  func_name = env.func_name
                      ; return_type = env.return_type
                      ; loop_level = env.loop_level + 1
                      ; scope = scope' } in
          let stmts' = List.map (fun s -> check_stmt env' s) stmts in
            For((name, t'), et1', et2', stmts')

  | Ast.While(e,stmts) ->
      let (_,t') as et' = check_expr env e in
        if t' <> Bool then
          failwith "\n***␣Expression␣in␣while␣statement␣must␣have␣type␣Bool."
        else
          let env' = { env with loop_level = env.loop_level + 1 } in
            While(et', check_block env' stmts)

  | Ast.Break     ->
      if env.loop_level > 0 then
        Break
      else
        failwith "\n***␣Attempt␣to␣BREAK␣outside␣of␣loop␣environment."

  | Ast.Continue  ->
      if env.loop_level > 0 then
        Break
      else
        failwith "\n***␣Attempt␣to␣CONTINUE␣outside␣of␣loop␣environment."

  | Ast.Sync      -> Sync

  | Ast.NullStmt  -> NullStmt
;;

(* Side effect: updates env scope (as in variable declaration). *)
```

```
let check_formal env (name, t) =
  let t' = check_loom_type env t in
  let new_var = ((name, t'), Param) in
    env.scope.vars <- new_var :: env.scope.vars;
    (name, t')
;;

let empty_environment name =
  let empty_scope = { parent = None
                    ; vars = []
                    ; types = []
                    ; funcs = [] } in
    { func_name   = name
    ; return_type = Void
    ; loop_level  = 0
    ; scope       = empty_scope }
;;

let extract_fdecl (f : Ast.func) : fdecl =
  let empty_env = empty_environment ""                    in
  let (_, types) = List.split f.Ast.formals               in
  let types' = List.map (check_loom_type empty_env) types in
  let ret_t' = check_loom_type empty_env f.Ast.ret_type   in
    (f.Ast.fname, types', ret_t')
;;

let get_fdecls funcs = List.map extract_fdecl funcs
;;

let check_kernel fdecls k =
  let name      = k.Ast.kname in
  let env       = empty_environment name in
  let kformals' = List.map (check_formal env) k.Ast.kformals in
    env.scope.funcs <- fdecls;
    let kbody'  = check_block env k.Ast.kbody in
      { kname     = name
      ; kformals  = kformals'
      ; kbody     = kbody' }
;;

let check_func (fdecls : fdecl list) (f : Ast.func) =
  let name      = f.Ast.fname in
  let env       = empty_environment name in
  let ret_type' = check_loom_type env f.Ast.ret_type in
  let env'      = { env with return_type = ret_type' } in
  let formals'  = List.map (check_formal env') f.Ast.formals in
    env'.scope.funcs <- fdecls;
    let body'   = check_block env' f.Ast.body in
      { fname     = f.Ast.fname
      ; formals   = formals'
      ; ret_type  = ret_type'
      ; body      = body' }
;;

let analyze : Ast.program -> program = function
  Some(k), funcs ->
    let fdecls = get_fdecls funcs in
```

```
    let k' = check_kernel fdecls k in
    let funcs' = List.map (check_func fdecls) funcs in
      (k', funcs')
| _              -> failwith "\n***␣No␣kernel␣function␣found."
;;

(* Sast pretty printer not yet implemented *)
let to_string _ = "Typechecking␣finished␣with␣no␣errors.\n"
```

compiler.mli

```
type ptx

val translate : Sast.program -> ptx
val to_string : ptx -> string
```

compiler.ml

```
(** Module Compiler *)

(** PTX virtual machine types
    Pred is used for conditionals, U16 is used for thread constants *)
type ptx_t = Pred | Void | S32 | S64 | U32 | U64 | F32 | F64 | B32 | B64 | U16

(** Memory address spaces. Param is the space of actual arguments to the kernel *)
type memspace = Local | Shared | Global | Param | Reg

type ptx_line =
    Direct      of  string
  | Entry       of  string * (ptx_t * string) list
  | Func        of  ptx_t * string * (ptx_t * string) list
  | Decl        of  memspace * ptx_t * string
  | Label       of  string
  | Branch      of  string option * string  (* option for conditioned branch *)
  | Op2         of  string * ptx_t * string * string
  | Op3         of  string * ptx_t * string * string * string
  | Convert     of  ptx_t * ptx_t * string * string
  | Load        of  memspace * ptx_t * string * string * int
  | Store       of  memspace * ptx_t * string * int * string

type ptx = ptx_line list

(* FIXME - incomplete LOOM type implementation; only covers basic types and
  arrays of basic types *)
type full_type =
    Basic   of ptx_t
  | ArrayN  of ptx_t * int      (* Size of array is integer literal *)
  | ArrayC  of ptx_t * string   (* Size of array specified by thread const *)

(* Type for translation environment *)
type symbols = {
  parent            : symbols option
; mutable vars      : (memspace * full_type * string) list
; return_label      : string
```

```
; continue_label    : string option
; break_label       : string option
}

type count_t = {
  mutable lab : int (* Label index count *)
; mutable pred: int (* Counts for various virtual register types *)
; mutable s32 : int
; mutable s64 : int
; mutable u32 : int
; mutable u64 : int
; mutable f32 : int
; mutable f64 : int
; mutable u16 : int
}

let counts =
  {lab=0; pred=0; s32=0; s64=0; u32=0; u64=0; f32=0; f64=0; u16=0}
;;

(** Return consecutively-numbered labels *)
let new_label () =
  let current = counts.lab in
  counts.lab <- current + 1;
  "L" ^ string_of_int current
;;

(** Return consecutively-numbered virtual register names of a PTX type *)
let new_register = function
    Pred -> let current = counts.pred in
              counts.pred <- current + 1;
              "%rp" ^ string_of_int current
  | S32 -> let current = counts.s32 in
              counts.s32 <- current + 1;
              "%rs" ^ string_of_int current
  | S64 -> let current = counts.s64 in
              counts.s64 <- current + 1;
              "%rsl" ^ string_of_int current
  | U32 -> let current = counts.u32 in
              counts.u32 <- current + 1;
              "%ru" ^ string_of_int current
  | U64 -> let current = counts.u64 in
              counts.u64 <- current + 1;
              "%rul" ^ string_of_int current
  | F32 -> let current = counts.f32 in
              counts.f32 <- current + 1;
              "%rf" ^ string_of_int current
  | F64 -> let current = counts.f64 in
              counts.f64 <- current + 1;
              "%rfl" ^ string_of_int current
  | U16 -> let current = counts.u16 in
              counts.u16 <- current + 1;
              "%rh" ^ string_of_int current
  | _   -> failwith "***␣can't␣make␣register␣of␣this␣type"
;;

let sizeof = function
```

```
    U16                           -> 2
  | Pred  | S32 | U32 | F32 | B32 -> 4
          | S64 | U64 | F64 | B64 -> 8
  | Void -> failwith "***␣should␣not␣try␣to␣determine␣size␣of␣void␣type"
;;

(* -- string output -- *)

let string_of_type = function
    Pred  ->  ".pred"   | Void  ->  ""
  | S32   ->  ".s32"    | S64   ->  ".s64"
  | U32   ->  ".u32"    | U64   ->  ".u64"
  | F32   ->  ".f32"    | F64   ->  ".f64"
  | B32   ->  ".b32"    | B64   ->  ".b64"
  | U16   ->  ".u16"
;;

let string_of_space = function
    Local   ->  ".local"
  | Shared  ->  ".shared"
  | Global  ->  ".global"
  | Param   ->  ".param"
  | Reg     ->  ".reg"
;;

let string_of_params (params) =
  let one (t, name) = ".param␣" ^ string_of_type t ^ "␣" ^ name in
    String.concat ",␣" (List.map one params)
;;

let string_of_ptx_line = function
    Direct(s)           ->  "\t"  ^ s
  | Entry(name,args)    ->  "\t.entry␣" ^ name ^ "("
                                        ^ string_of_params args ^ ")"
  | Func(Void,name,args)->  "\t.func␣"  ^ name ^ "("
                                        ^ string_of_params args ^ ")"
  | Func(t,name,args)   ->  "\t.func␣(" ^ string_of_params [(t,name)] ^ ")␣"
                                        ^ name ^ "("
                                        ^ string_of_params args ^ ")"
  | Decl(space,t,name)  ->  "\t" ^ string_of_space space ^ "␣"
                                ^ string_of_type t ^ "␣"
                                ^ name ^ ";"
  | Label(lab)          ->  lab ^ ":"
  | Branch(None,lbl)    ->  "\tbra\t" ^ lbl ^ ";"
  | Branch(Some(p),lbl) ->  "@" ^ p ^ "\tbra\t" ^ lbl ^ ";"
  | Op2(op,t,d,a)       ->  "\t"  ^ op ^ string_of_type t ^ "\t"
                              ^ d ^ ",␣" ^ a ^ ";"
  | Op3(op,t,d,a,b)     ->  "\t"  ^ op ^ string_of_type t ^ "\t"
                              ^ d ^ ",␣" ^ a ^ ",␣" ^ b ^ ";"
  | Convert(t1,t2,d,a)  ->  "\tcvt" ^ string_of_type t1 ^ string_of_type t2 ^ "\t"
                              ^ d ^ ",␣" ^ a ^ ";"
  | Load(s,t,d,a,i)     ->  "\tld"  ^ string_of_space s ^ string_of_type t ^ "\t"
                              ^ d ^ ",␣[" ^ a ^ "+" ^ string_of_int i ^ "];"
  | Store(s,t,d,i,a)    ->  "\tst"  ^ string_of_space s ^ string_of_type t ^ "\t"
                              ^ "[" ^ d ^ "+" ^ string_of_int i ^ "],␣" ^ a ^ ";"
;;
```

```
let to_string p =
  let strs = List.map string_of_ptx_line p in
    (String.concat "\n" strs) ^ "\n"
;;

(* -- translation functions -- *)

let trans_type = function
    Sast.Bool       ->  Pred
  | Sast.Void       ->  failwith "***␣cannot␣translate␣Sast.Void␣type."
  | Sast.Int32      ->  S32
  | Sast.Int64      ->  S64
  | Sast.Uint32     ->  U32
  | Sast.Uint64     ->  U64
  | Sast.Float32    ->  F32
  | Sast.Float64    ->  F64
  | _               ->  U32  (* pointer for other types *)
;;

let trans_type_full t =
  match t with
    Sast.Bool   | Sast.Void   | Sast.Int32    | Sast.Int64
  | Sast.Uint32 | Sast.Uint64 | Sast.Float32  | Sast.Float64 ->
      Basic(trans_type t)
  | Sast.Array(elem_t, Sast.IntLiteral(n)) ->
      ArrayN(trans_type elem_t, n)
  | Sast.Array(elem_t, Sast.Place(Sast.Id(Sast.Const(s, _)))) ->
      ArrayC(trans_type elem_t, s)
  | _ -> failwith "***␣Not␣all␣compound␣types␣implemented."
;;

let trans_space = function
  Sast.Local -> Local   | Sast.Global -> Global
| Sast.Shared -> Shared | Sast.Param -> Param
;;

let trans_special = function
  "T"         -> "%tid"
| "THREADS"   -> "%threads" (* FIXME -- need to think more about how these work *)
| "B"         -> "%ctaid"
| "BLOCKS"    -> "%blocks"
| "BLOCKSIZE" -> "%blocksize"
| _           -> failwith "***␣unknown␣special␣constant."
;;

(** Translate expression, returning a register, instruction-list pair *)
(* Need to pass in env for array, identifier lookups *)
let rec trans_expr env (e, t) =
  let t' = trans_type t in
  let rname = new_register t' in  (* create a virtual register for each node *)
    match e with
      Sast.IntLiteral(n)    ->
        rname, [ Op2("mov", t', rname, string_of_int n) ]

    | Sast.FloatLiteral(f)  ->
        rname, [ Op2("mov", t', rname, string_of_float f) ]
```

```
  | Sast.BoolLiteral(b)   ->
      rname, [ Op2("mov", t', rname, if b then "1" else "0") ]

  | Sast.Place(p)         -> trans_place env p t

  | Sast.Binop(et1, op, et2)  ->
      let reg1, ptx1 = trans_expr env et1 in
      let reg2, ptx2 = trans_expr env et2 in
      let ptx_op =
        ptx1
      @ ptx2
      @ begin
          match op with
            Sast.Mult   ->  [ Op3("mul", t' , rname, reg1, reg2) ]
          | Sast.Div    ->  [ Op3("div", t' , rname, reg1, reg2) ]
          | Sast.Plus   ->  [ Op3("add", t' , rname, reg1, reg2) ]
          | Sast.Minus  ->  [ Op3("sub", t' , rname, reg1, reg2) ]
          | _ -> [ Direct("//␣BinOp␣placeholder") ]
        end
      in
        rname, ptx_op

  | Sast.Unop(op,et)       ->
      let reg1, ptx1 = trans_expr env et in
      let ptx_op =
        ptx1
      @ begin
          match op with
          | Sast.Negative       ->
              let byte_t = if (sizeof t') = 32 then B32 else B64 in
              let reg1' = new_register t' in
                [ Op2("neg", byte_t , rname, reg1') ]
          | Sast.Not
          | Sast.BitComplement  ->
              let byte_t = if (sizeof t') = 32 then B32 else B64 in
              let reg1' = new_register t' in
                [ Op2("not", byte_t , rname, reg1') ]
        end
      in
        rname, ptx_op

  | Sast.Convert((e, dest_t) as et, source_t)    ->
      (* FIXME: only works for basic types *)
      let reg1, ptx1 = trans_expr env et in
      let dest_t' = trans_type dest_t in
      let source_t' = trans_type source_t in
      let ptx_op =
        ptx1
      @ [ Convert(source_t', dest_t', rname, reg1) ]
      in
        rname, ptx_op

  | Sast.Call(f,args)     ->
      rname, [ Direct("//␣Call␣placeholder") ]

  | Sast.NullExpr  -> rname, []
```

```
(** Translate a 'place' (lvalue), returning a register holding the address and
    a list of instructions used to calculate the address *)
and trans_place env p t =
    "r_debug", [ Direct("//␣Place␣placeholder") ]
;;

let rec trans_stmt env stmt =
  match stmt with
    Sast.Block(stmts)   ->
      (* PTX has scope blocks built-in *)
        [ Direct("{") ]
      @ List.concat (List.map (trans_stmt env) stmts)
      @ [ Direct("}") ]

  | Sast.Vdecl(space, (name, t), init_opt)  ->  begin
      let space' = trans_space space in
      let full_t = trans_type_full t in
      let name', t' =
        match full_t with
          Basic(typ)      ->  (name, typ)
        | ArrayN(typ, n)  ->  (name ^ "[" ^ string_of_int n ^ "]", typ)
        (* FIXME - need to implement ArrayC types *)
        | _               ->  failwith "***␣cannot␣declare␣compound␣type"
      in
        [ Decl(space', t', name') ]
    end

  | Sast.Tdecl(name, t) ->
      (* Defined types are interpreted by the typechecker, so nothing to do here *)
        []

  | Sast.Expr(et)     ->    (* throw away register holding value *)
       let rname, lst = trans_expr env et in
         lst

  | Sast.Return(e, t)   ->
       [ Branch(None, env.return_label) ]

  | Sast.Assign(p, e)   ->
      (* let place_r, place_t = trans_place p *)
        [ Direct("//␣Assign␣placeholder") ]

  | Sast.If((e, t), stmts1, stmts2)     ->
       [ Direct("//␣If␣placeholder") ]

  | Sast.For((name, t), e1, e2, stmts)  ->
       [ Direct("//␣For␣placeholder") ]

  | Sast.While(et, stmts)        ->
    (*  General idea:
       label1
         if (test_code) branch label2
         branch label3
       label2:
         recursively call trans_stmt on the body block with a translation
             environment having break_label = label3 and continue_label = label1
         branch label1
```

```
        label3:
      *)
        [ Direct("//␣For␣placeholder") ]

  | Sast.Break          ->
        [ Direct("//␣Break␣placeholder") ]

  | Sast.Continue       ->
        [ Direct("//␣Continue␣placeholder") ]

  | Sast.Sync           ->
        [ Direct("bar.sync;") ]

  | Sast.NullStmt       -> []
;;

let trans_params params =
  let trans_one (name, t) = (trans_type t, name) in
    (List.map trans_one params) @ [(U32, "grid")]
;;

let trans_params_full params =
  let trans_one (name, t) = (Param, trans_type_full t, name) in
    (List.map trans_one params) @ [(Param, ArrayN(U32,10), "grid")]
;;

let new_environment vs =
  { parent          = None
  ; vars            = vs
  ; return_label    = new_label()
  ; continue_label  = None
  ; break_label     = None }
;;

let trans_kernel k =
  let kname = k.Sast.kname in
  let basic_params = trans_params k.Sast.kformals in
  let full_params = trans_params_full k.Sast.kformals in
  let env = new_environment full_params in
    [ Entry(kname, basic_params)
    ; Direct("{")
    ; Decl(Reg, Pred, "%rp<99>" )   (* Declare types of virtual registers *)
    ; Decl(Reg, S32,  "%rs<99>" )
    ; Decl(Reg, S64,  "%rsl<99>")
    ; Decl(Reg, U32,  "%ru<99>" )
    ; Decl(Reg, U64,  "%rul<99>")
    ; Decl(Reg, F32,  "%rf<99>" )
    ; Decl(Reg, F64,  "%rfl<99>")
    ; Decl(Reg, B32,  "%rb<99>" )
    ; Decl(Reg, U16,  "%rh<99>" )
    ; Decl(Reg, U32,  "%threads")   (* Set up thread constants *)
    ; Decl(Reg, U32,  "%blocks" )
    ; Decl(Reg, U32,  "%blocksize")
    ; Load(Param, U32, "%threads",    "grid", 0)
    ; Load(Param, U32, "%blocks",     "grid", 4)
    ; Load(Param, U32, "%blocksize",  "grid", 8) ]
  @ trans_stmt env (Sast.Block(k.Sast.kbody))
```

```
  @ [ Label(env.return_label)
    ; Direct("exit;")
    ; Direct("}")
    ]
;;

let trans_func f = [ Direct("//␣FUNC␣" ^ f.Sast.fname ^ "␣placeholder") ]
;;

let translate (k, fs) =
  [ Direct(".version␣1.4")
  ; Direct(".target␣sm_10,␣map_f64_to_f32") ]
  @ List.concat (List.map trans_func fs)
  @ (trans_kernel k)
;;
```