LéPiX

ThePhD (jm3689)
jm3689@columbia.edu

https://github.com/ThePhD/lepix

December 20, 2016

Contents

Chapter 1

Introduction

LéPiX is a small, general-purpose programming language whose goal is to make working with parallel computation and multidimensional arrays simple. Featuring a preprocessor, namespacing, sliced multidimensional array syntax, parallel blocks based on an invocation count and ID, and bottom-up type derivation (automatic type deduction), the goal of the language is to produce an environment that is heavily statically checked and ensures a degree of correctness the user of the language can rely on.

However, this implementation of LéPiX specifically departed from some of the original design goals due to time constraints and team issues. Therefore, while parallelism and arrays were on the table, neither made it into the implementation in the given time frame I had to put the rest of the language (entire semantic analyzer plus all of the codegen) (about 2 weeks, plus post-mortem time). The implementation here instead set out to demonstrate how 4 techniques can be achieved with the language:

1. Namespaces - namespacing as defined in the original language, but lacking using statements inside definition blocks
2. Overloading - having multiple functions assigned to the same name, separated internally by a name-mangling scheme based on arity and arguments.
3. Bottom-up Type Derivation - deduction of return types for functions based on the expression of returns (or none therein).

4. Call targets as expressions - DICE[1] and other languages – including our own, at first – implemented function calls as an identifier plus necessary function all syntax. This implementation of LéPiX treats it as an expression, which presents unique challenges for the previous 2 goals.

1.1 Language Proposal

Below is our initial language proposal, in its entirety. It was very ambitious, and Professor Edwards told us to scale our goals back considerably. From it, we threw out GPU code generation in exchange for Parallelism and code generation for LLVM IR alone, and also wanted to focus on syntax for multidimensional arrays.

[1]http://www.cs.columbia.edu/ sedwards/classes/2015/4115-fall/reports/Dice.pdf

LéPiX - Ceci n'est pas un Photoshop

Fatima Koly (fak2116)
Manager
`fak2116@barnard.edu`

Gabrielle Taylor (gat2118)
Tester
`gat2118@columbia.edu`

Jackie Lin (jl4162)
Tester
`jl4162@columbia.edu`

Akshaan Kakar (ak3808)
Codegen & Language Guru
`ak3808@columbia.edu`

ThePhD (jm3689)
Codegen & Language Guru
`jm3689@columbia.edu`

`https://github.com/ThePhD/lepix`

September 28, 2016

Contents

Chapter 1

An Overview

1.1 Introduction

Heterogeneous computing and graphics processing is an area of intense research. Many existing solutions – such as C++ AMP [5] and OpenCL [3] – leverage the power of an existing language and add preprocessors and software libraries to connect a user to allow code to be run on the GPU. Due to its massively parallelizable nature, code executed on the GPU can be orders of magnitude faster, but comes at the cost of having to master a specific programming library and often learn new framework-specific or platform-specific language subsets in order to compute on the GPU.

1.2 Enter LéPiX

We envision LéPiXto be a graphics processing language based loosely on a subset of the C language. Using an imperative style with strong static typing, we plan to support primitives that enable quick and concise programs for image creation and manipulation. The most novel feature we have planned for the language is the ability to compile to both the CPU as well as the GPU. The reason is to enable high performance and ease the pain most notably found with trying to write applications which leverage the power of the GPU at the cost of a steep learning curve for explicit non-CPU device computation APIs, e.g. OpenGL Compute Shaders, DirectCompute, OpenCL, CUDA, and others. The final goal is to enable the writing of computer vision and computer graphics algorithms in LéPiX with relative ease compared to other languages.

Chapter 2

The LéPiX Language

2.1 Language

The language itself is meant to follow loosely from imperative C, but it subject to change as we refine our desired set of primitives and base operations. What follows is a loose definition of the primitive types, operations we would like to implement to get a baseline for the language, and how would would like to put those together syntactically and grammatically. All of these definitions will be mostly informal.

The goal of LéPiX is to provide a strongly and statically typed language by which to perform image manipulation easily. The hope is that powerful algorithms can be expressed in the language by providing a useful set of basic types, including the concept of a pixel and a natively slice-able matrix type that will serve as the basis for an image.

- Primitive Data Type: the core types defined by the language itself and given a set of supportable operations
- Built-ins: some of the built-in functions to help users
- Function Definitions: how to define a function in the language and use it
- Operators: which operators put into the language and operate on the primitive types

2.1.1 User Defined Types

Support for user-defined types is planned, but will be a stretch goal (5.1). We want to support having user-defined types that can be used everywhere, and for it to be able to overload the conceivable set of all operations.

2.1.2 Syntax

This is a basic guide to the syntax of the language. We are striving to develop a C-Like imperative language. It will follow many of the C conventions, but with some differences that we think will better fit the domain we are striving to work within. As we do not have a formal grammer just yet, we will present potential programs that we wish to allow to generate appropriate code. You can find these example programs in 3. Below are some quick points about the LéPiX language.

Namespacing As a stretch goal, LéPiX will attempt to support namespacing, to avoid record collision problems as present in OCaml (without the use of modules) and to formalize the good practice of of prepending the short name of the module / library to all functions in C code. Other languages have explicit support. LéPiX will attempt to encourage code sharing and reuse by including the use of namespacing.

Keywords The following words will be reserved for use with the language: namespace, struct, class, typename, typedef, for, while, break, if, elseif, else, void, unit, int, float, uint, pixel, image, vec, vector, mat, matrix. In addition, all identifiers containing __ (two underscores) are reserved for the use of the compiler and the standard library.

The standard library reserves the usage of the namespace lib. The language will place intrinsics and built-ins within the lpx namespace.

Function Definitions Typical function definitions will follow a usual C-style syntax. An example in pseudo-lexer code:

```
function <return type> name ( <parameter list> ) {
    <statement / expression>
    ...
}
```

code/func–def.lex

Ideally, we would like the order of function definition not to matter, so long as it appears *somewhere* in the whole source code listing. Early versions of the LéPiX compiler might require definition before use, for sanity purposes.

As a stretch goal, there are plans for lambda functions (anonymous function values) to be generated by a much more terse syntax.

Control Flow Control flow will follow a C-style syntax as well. An example in pseudo-lexer code:

```
if expr {
     <statement / expression>
     ...
}
elseif expr {
     <statement / expression>
     ...
}
else {
     <statement / expression>
     ...
}
```

code/flow–control.lex

Operations The LéPiX language will support most of the basic mathematical operators. Other operators will be provided VIA functions. These include:

- Mathematical - `plus (+), minus (-), multiply (*), divide (/), modulus (%), power of (**)`
- Logical - `and (&&), or (||), less than (<), greater than (>), equal to (==), not equal to (!=) negate (!)`

Some of these will use both the symbol and the name, such as "not" for negation. Support for bitwise operations, such as left shift and right shift as well as bitwise and / bitwise or, will come as a stretch goal, depending on whether we can handle these basics. Ternary conditionals may also prove useful, but will not be immediately supported.

Comments Single-line comments will begin with //. Multi-line and *nestable* comments will begin with /**/ will be supported as well.

2.1.3 Built-ins

Some useful built-in functions that will be provided with the language:

- Trigonometric functions: lib.sin, lib.cos, lib.tan, lib.asin, lib.acos, lib.atan, lib.atan2

- Power Functions: lib.sqrt, lib.cbrt, lib.pow
- Exponential Functions: lib.expe, lib.exp2, lib.loge, lib.log2, lib.log10

All degree arguments will come in radians. Other functions that operate on built-in types will be provided as library functions, to allow for replacement if necessary.

2.1.4 Primitive Types

Primitives are integral types and multi-dimensional array types. Vectors, Matrices, and Pixels are all subsets of N-dimensional array types. String will be presented as a built-in type, but may be implemented either as a built-in or just an always-included library type. The purpose of string will be specifically to handle reading in and writing out from the file system, as that is the only way to handle such a case. See 2.1 for details.

Table 2.1: Primitive Types in LéPiX

Type	variants	Purpose
void	nothing	built in single-value / empty type
int#	# can be 8/16/32/64 (defaults to 32 without name)	signed integral type
uint#	# can be 8/16/32/64 (defaults to 32 without name)	unsigned integral type
float#	# can be 16/32/64 (defaults to 32 without name)	floating type
<type name>[<optional #>]	array type; can be multidimensional by adding [<optional # >]	basic primitive that will help us represent an image in its decoded form; can be sliced to remove 1 dimension
vec#<type name>	aliases for 1-dimensional fixed-size arrays	base type for pixels
mat#<type name>	aliases to 2-dimensional fixed-size arrays	can be sliced into vectors
string	utf8-encoded string; potentially more later	primarily, to address the file system; not really interested in complex text handling; basically treated as an array of specially-typed uint8
pixel	rgba (red-green-blue-alpha); hsv (hue saturation value)	in the future, this will be something that will need to be customizable to support varying image types of different bit depths

Chapter 3

Examples

The following are some examples of programs that we would like to be written in LéPiX. This does not reflect final syntax and is mostly based on equivalent or near-equivalent C-style code:

```
function image red(int width, int height) {
    // Create an image of a specific width / height
    image ret = image(width, height);
    for(int i = 0; i < width;  i++) {
        for(int j = 0; j < height; j++) {
            // r, g, b creation
            ret[i][j] = pixel(1,0,0);
        }
    }

    return ret;
}

function int main () {
    image img = red(img);
    lib.save(img);
    return 0;
}
```

Listing 3.1: red.lpx

The red example shows up some simple conditionals in a for loop to write all-red to an image. It is not the most exciting code, but it has a lot of moving parts and will help us test several parts of the library, from how to call functions to basic iteration techniques.

```
function void flip(image img) {
    for (int ri = 0; ri < img.height / 2; ri++) {
        // matrix slicing: get back array from index
```

```
        pixel[] toprow = img[ri];
        // 0-based indexing
        pixel[] bottomrow = img[img.height - ri - 1];
        for (int ci = 0; ci < img.width; ++ci) {
            // generic swap call in library
            lib.swap(toprow[ci], bottomrow[ci]);
        }
    }
}

function int main () {
    image img = lib.read("meow.png");
    flip(img);
    lib.save(img);

    // implicit return 0:
    // we want this to be able to work with
    // command-line environments as well
}
```

Listing 3.2: flip.lpx

This above code is a bit more complicated. It shows that we can save a slice of a matrix's (image's) row, operate on it, and even call the library function `lib.swap` on its `pixel` elements. It also demonstrates a string literal, and passing it to the `lib.read` function to pull out a regular image from a PNG, and then saving that same image. It also shows off an implicit return 0 (we expect our programs to be run in the context of a shell environment, and to play nice with the existing C tools in that manner).

Chapter 4

Codegen

4.1 LLVM IR

The current goal for generating the code for this language is to use LLVM and serialize to LLVM IR. This will allow our language to work on a multiple of platforms that LLVM supports, provided we can successfully connect our AST / DAG with the our code generator.

4.2 SPIR-V

For the GPU, we still want to compile to LLVM IR. But, with the caveat that we make it work to push out to SPIR-V code using the Khronos LLVM ¡-¿ SPIR-V Bidirectional Translator[1][1]. The good news is that everything from our source code processing steps to our DAG / AST generation can be done in OCaml. However, SPIR-V is new and OCaml bindings for this relatively new project are not something that is quite established: it is conceivable that our code generator will be written in C++ as opposed to any other language, simply because of the library power behind what is already present is written in C and C++, and interfacing that with OCaml might be exceptionally difficult. Granted, we could also write an OCaml Code Generator for SPIR-V from scratch, but this does not seem like a prudent use of our time.

It is also very important to note that the LLVM IR ¡-¿ SPIR-V Translator produces SPIR-V, which by itself cannot be run on anything. We would need a C or C++ compiled Vulkan Driver to take that bit of our program and run it in SPIR-V land. Furthermore, many operations – such as data reading –

[1]Available here: https://github.com/KhronosGroup/SPIRV-LLVM

are not instructions we can exactly slot into SPIR-V code, and would require a bootstrapper of some sort nontheless.

Chapter 5

Stretch Goals

5.1 User Defined Types

User-defined structures are a stretch goal of this project. The core idea is that if we can manage to create pixel, vec#∗, mat#∗, etc. types using the language, we would be able to simply make this kind of functionality available to users. Currently, we plan to hard code these types in at the moment, however.

5.2 Namespacing

Similar to user defined types, namespacing allows an element of organization to be brought to written code. Currently, we are going to hard code built ins to the lib and lpx namespaces.

5.3 Named Parameters

This is an entirely fluff goal to make it easier to call certain functions. The idea is that arguments not yet initialized by the ordered list of arguments to a function call can be specified out-of-order – as long as others inbetween are defaulted – by passing a name=(expression) pairing, separated by commas like regular function arguments.

5.4 Lambda Functions

As mentioned in 2.1.2, we would like to support lambdas as a way of definition functions. Currently, we do not know what the most succinct and terse syntax for our language would be.

5.5 Standard Library Implementation

It would be nice to fill out a standard library implementation, to vet the LéPiX compiler. Candidates would include some basic functions in the lib namespace for manipulation of the image type.

5.6 Movies: Encoding

If we can have built in types for image and the like, then LéPiX could theoretically handle movies by presenting to the user frames of data in sequential order. Doing this is orders of magnitude difficult.

5.7 Windowing: Realtime Visuals

Part of the magic of the graphics card is its ability to perform specific kinds of computation very quickly. It would be very beneficial to have some sort of way to display those visuals without having to serialize them to disk (e.g., a display function or a window of some sort which can be backed by a write-only image).

5.8 Error Noises

The compiler should make a snobby "Ouhh Hooo!" noise in french when the user puts in ill-formed code.

Bibliography

[1] Khronos Group, *Khronos Registry: SPIR-V 1.1*, April 18, 2016. https://www.khronos.org/registry/spir-v/

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Pat Hanrahan, Mike Houston, Kayvon Fatahalian, *Brook for GPUs*, Stanford University, 2016. https://graphics.stanford.edu/projects/brookgpu/arch.html

[3] Khronos Group, *OpenCL 2.2*, April 12, 2016. https://www.khronos.org/opencl/

[4] Chuck Walbourn, Microsoft, *Compute Shaders*, July 14, 2010. https://blogs.msdn.microsoft.com/chuckw/2010/07/14/directcompute/

[5] Dillion Sharlet, Aaron Kunze, Stephen Junkins, Deepti Joshi, *Shevlin Park: Implementing C++ AMP with Clang/LLVM and OpenCL*, November 2012. http://llvm.org/devmtg/2012-11/Sharlet-ShevlinPark.pdf

Chapter 2

Tutorial

2.1 Invoking the Compiler

Obtaining the LéPiX compiler – lepixc – in order to build LéPiX programs with it is simple, once the dependencies are set up. It requires a working OCaml compiler of version >= 4.0.3. Building requires ocamlbuild, ocamlfind, menhir and OPAM for an easy time, but if you are brave and willing to figure out the nightmare it takes to get these dependencies working on Windows than it can work on Windows machines as well.

From the root of the repository, run make −C source to create the compiler, or cd into the *source* directory and invoke make from there.

When the compiler is made, it will be within the ./source folder of the repository. An invocation without any arguments or filenames will explain to the user how to use it, like so:

```
source/lepixc

Help:
    lepix [options] filename [filenames...]
        filename | filenames can have one option -i or --input
    options:
        -h   --help             print the help message
        -p   --preprocess       Preprocess and display source
        -i   --input            Take input from standard in (default: stdin)
        -o   --output <value> Set the output file (default:
```

```
    stdout)
         -t    --tokens          Print the stream of tokens
         -a    --ast             Print the parsed Program
         -s    --semantic        Print the Semantic Program
         -l    --llvm            Print the generated LLVM code
         -c    --compile         Compile the desired input and
    output the final LLVM
         -v    --verbose         Be as explicit as possible with
    all steps
```

history/lepixc.txt

Users can stack one-word options together using syntax like `source/lepixc -ls`, which will pretty-print the Semantic Analysis tree and also output the LLVM IR code.

2.2 Writing some Code

Function definitions are fairly C-Like, with the exception that all type-annotations appear on the right-hand-side. Users can define variables and functions by using the `var` and `fun` keywords, respectively.

To receive access to the standard library, put `import lib` in the program as well. It will end up looking somewhat like this:

```
import lib

fun main () : int {
    lib.print_n("hello world");
    return 0;
}
```

One function must always be present in your code, and that is the `main` function. It must return an `int`. If the user does not return an integer value from main, then a `return 0` will be automatically done for you. Also of importance is that functions can have their return type figured out from their return statements. For example, you can remove the `: int` above and the code will still compile and run:

```
import lib

fun main () {
    lib.print_n("hello world");
    return 0;
```

```
}
```

This goes for more than just the main function, but any function you define!

Chapter 3

Language Reference Manual

On the next page is the language reference manual for the original LéPiX language. The final implementation did not meet all of the requirements due to having to work alone for the last two weeks and insufficient teammate contribution during the project, but thankfully we still capture the majority of the language's constructs in every thing but the final code generation and semantic analysis stages, leaving room to improve the implementation in the future.

LéPiX Language Specification

Ceci n'est pas un Photoshop

Fatima Koly (fak2116)
Manager
fak2116@barnard.edu

Gabrielle Taylor (gat2118)
Language Guru
gat2118@columbia.edu

Jackie Lin (jl4162)
Tester
jl4162@columbia.edu

Akshaan Kakar (ak3808)
Codegen
ak3808@columbia.edu

ThePhD (jm3689)
System Architect
jm3689@columbia.edu

https://github.com/ThePhD/lepix

October 26, 2016

Contents

Part I

Introduction

Chapter 1

Tutorial

1.1 Hello, World!

This is an example of a Hello World program in LéPiX. It creates an array from an initializer, and then proceeds to save it to the directory of the running program under the name "hello.bmp":

```
fun main () : int {
    // 2 dimensional array
    // of integers, initialized as a string
    // from a "bitmap"
    var arr : int[[]] = "\
|   | ||| |   |   |||   |   | ||| ||| |   ||
|   | |   |   |   | |   |   | | | | |   | |
||||| ||  |   |   | |   | | | | ||  |   | |
|   | |   |   |   | |   || || | | | |   | |
|   | ||| ||| ||| |||   |   | ||| | | ||| || ";
    lib.save("hello.bmp", arr);
}
```

Listing 1.1: hello world

1.2 Variables and Declarations

1.2.1 Variables

Variables are made with the var declaration. You can declare and assign variables by giving them a name and then referencing that name in other places.

```
fun main () : int {
      var a : int = 24 * 2 + 1;
      // a == 49
      var b : int = a % 8;
      // b == 1
      var c : int[[5, 2]] = [
      0, 2, 4, 6, 8, 10;
          1, 3, 5, 7, 9, 11;
     ];
      var value : int = a + b + c[0, 4];
      // value == 58
      return value;
}
```

Listing 1.2: variable declaration and manipulation

1.2.2 Mutability

Variables can also be declared immutable or unchanging by declaring them with let. That is, let is the same as a var const, and var is the same as let mutable.

```
fun main () : int {
      let a : float = 31.5;
      var const b : float = 0.5;
      var c : int = 0;
      c = lib.trunc(a + b);
      // compiler error: 'var const' variable is immutable
      b = 2.5;
      // compiler error: 'let' variable is immutable
      a = 1.1;
      return c;
}
```

Listing 1.3: mutability

1.3 Control Flow

Control flow is important for programs to exhibit more complex behaviors. LéPiX has for and while constructs for looping, as well as if, else if, else statements. They can be used as in the following sample:

```
fun main () : int {
    for (var x : int = 0 to 10) {
        var x : int = lib.random_int(0, 40);
        if (x < 20) {
            lib.print("It's less than 20!");
        }
        else {
            lib.print("It's equal to or greater than 20");
        }
    }
}
```

"intro/tutorial/code/flow.hak"

1.4 Functions

1.4.1 Defining and Declaring Functions

Functions can be called with a simple syntax. The goal is to make it easy to pass arguments and specify types on those arguments, as well as the return type. All functions are defined by starting with the fun keyword, followed by an identifier including the name, before an optional list of parameters.

```
fun sum (arr : int[]) : int {
    int a = 2;
    int b = 3;
    return a + b;
}

fun numbers () : int[] {
    return [ 1, 2, 3 ];
}

fun main () : int {
    return sum(numbers());
}
```

Listing 1.4: functions

1.4.2 Parameters and Arguments

All arguments given to a function for a function call are passed by value, unless the reference symbol & is written just before the argument, as shown in the below example. This allows a person to manipulate a value that was passed in directly, rather than receiving a copy of it the argument.

```
fun fibonacci_to (n : int, &storage : int[]) : int {
    int index = 0;
    var result : int = 0;
    var n_2 : int = 0;
    var n_1 : int = 1;
    while (n > 0) {
        result = n_1 + n_2;
        storage[index] = result;
        n_2 = n_1;
        n_1 = result;
        --n;
        ++index;
    }
    return result;
}

fun main () : int {
    var storage : int [3] = [];
    var x : int = fibonacci_to(3, storage);
    return x;
}
```

Listing 1.5: arguments

Part II

Reference Manual

Chapter 2

Expressions, Operations and Types

2.1 Variable Names and Identifiers

2.1.1 Identifiers

1. All names for all identifiers in a LéPiX program must be composed of a single start alpha codepoint followed by either zero or more of a digit or an alpha codepoint. Any identifier that does not follow this scheme and does not form a valid keyword, literal or definition is considered ill-formed.

2. All identifiers that containing two underscores __ in any part of the name are reserved for usage by the compiler implementation details and may not be used by programs. If an identifier has two underscores the program is considered ill-formed.

3. All identifiers prefixed by '`lib.`' (i.e., belong in the `lib` namespace) are reserved by the standard to the standard library and nothing may be defined in that namespace by the program, aside from implementations of the standard library.

2.2 Literals

2.2.1 Kinds of Literals

There are many kinds of literals. They are:

literal:

boolean-literal
integer-literal
floating-literal
string-literal

2.2.2 Boolean Literals

1. A boolean literal are the keywords `true` or `false`.

2.2.3 Integer Literals

1. An integer literal is a valid sequence of digits with some optional alpha characters that change the interpretation of the supplied literal.
2. A decimal integer literal uses digits '0' through '9' to define a base-10 number.
3. A hexidecimal integer literal uses digits '0' through '9', 'A' through 'F' (case insensitive) to define a base-16 number. It must be prefixed by `0x` or `0x`.
4. An octal integer literal uses digits '0' and '7' to define a base-8 number. It must be prefixed by `0c` or `0C`.
5. A binary integer literal uses digits 0 and 1 to define a base-2 number. It must be prefixed by `0b` (case sensitive).
6. An n-digit integer literal uses the characters below to define a base-n number. It must be prefixed by `0n` or `0N`. It must be suffixed by `#n`, where n is the desired base. The character set defined for these bases

goes up to 63 characters, giving a maximum arbitrary base of 63. The characters which are:

$0 - 9$, $A - Z$, $a - z$, _

7. Arbitrary bases for n-digit must be base-10 numbers.

8. Groups of digits may be separated by a `'` and do not change the integer literal at all.

2.2.4 Floating Literals

1. A floating literal has two primary forms, utilizing digits as defined in 2.2.3.

2. The first form must have a dot '.' preceded by an integer literal and/or suffixed by an integer literal. It must have one or the other, and may not omit both the prefixing or suffixing integer literal.

3. The second form follows 2, but includes the exponent symbol `e` and another integer literal describing that exponent. Both the exponent and integer literal must be present in this form, but if the exponent is included then the dot is not necessary and may be prefixed with only an integer literal or just an integer literal and a dot.

2.2.5 String Literals

1. A string literal is started with a single '`'`' or double '`"`' quotation mark and does not end until the next matching single '`'`' or double '`"`' quotation mark character, with respect to what the string was started with. This includes any and all spacing characters, including newline characters.

2. Newline characters in a multi-line string will be included in the string as an ASCII Line Feed `\n` character.

3. A string literal must remove the leading space on each line that are equivalent to all other lines in the text, and any empty leading space at the start of the string.

4. A string literal may retain the any leading space and common indentation by prefixing the opening single or double quotation mark with an 'R'.

2.3 Variable Declarations

2.3.1 let and var declarations

variable-initialization:

> *let | var (mutable | const)$_{optional}$ <identifier> : <type>;*

1. A variable can be declared using the let and var keywords, an identifier as defined in 2.1.1 and optionally followed by a colon ':' and type name. This is called a variable declaration.

2. A variable declared with let is determined to be immutable. Immutable variables cannot have their values re-assigned after declaration and initialization.

3. A variable declared with var is immutable. Mutable variables can have their values re-assigned after declaration and initialization.

4. let mutable is equivalent to var const.

5. It is valid to initialize or assign to a mutable variable from an immutable variable.

6. A declaration can appear at any scope in the program.

2.4 Initialization

2.4.1 Variable Initialization

variable-declaration:

> *let | var (mutable | const)$_{optional}$ <identifier> : <type> = (expression);*

1. Initialization is the assignment of an expression on the right side to a variable declaration.
2. If the expression cannot directly initialize or be coerced to initialize the type on the left, then the program is ill-formed.

2.4.2 Assignment

assignment-expression:

expression = *expression*

2.5 Access

2.5.1 Member Access

member-access-expression:

(*expression*) . <*identifier*>

1. Member access is performed with the dot '.' operator.
2. If the expression does not evaluate to a type that can be accessed with the dot operator, the program is ill-formed.
3. If the identifier is not available per lookup rules in 2.5.2 on the evaluated type, the program is ill-formed.

2.5.2 Member Lookup

1. When a member is accessed through the dot operator as in 2.5.1, a name must be found that matches the supplied identifier . If there is none,

2.6 Parenthesis

parenthesis-expression:

> ‘(’ *expression* ‘)’

1. Parentheses define expression groupings and supersede precedence rules in 2.1.

2.7 Arithmetic Expressions

2.7.1 Binary Arithmetic Operations

addition-expression:

> *expression* + *expression*

subtraction-expression:

> *expression* − *expression*

division-expression:

> *expression* / *expression*

multiplication-expression:

> *expression* * *expression*

modulus-expression:

> *expression* % *expression*

1. Symbolic expression to perform the commonly understood mathematical operations on two operands.

2. All operations are left-associative.

2.7.2 Unary Arithmetic Operations

unary-minus-expression:

$-expression$

1. Unary minus is typically interpreted as negation of the single operand.
2. All operations are left-associative.

2.8 Incremental Expressions

2.8.1 Incremental operations

post-increment-expression:

$(\ expression\ )++$

pre-increment-expression:

$++(\ expression\ )$

post-decrement-expression:

$(\ expression\ )--$

pre-decrement-expression:

$--(\ expression\ )$

1. Symbolic expression that should semantically evaluate to $(\ \text{expression}\ ) = (\ \text{expression}\ ) + 1$.
2. $(\ \text{expression}\ )$ is only evaluated once.

2.9 Logical Expressions

2.9.1 Binary Compound Boolean Operators

and-expression:

expression *and* *expression*

expression *&&* *expression*

or-expression:

expression *or* *expression*

expression || *expression*

1. Symbolic expressions to check for logical conjunction and disjunction.
2. For the and−expression, short-circuiting logic is applied if the expression on the left evaluates to false. The right hand expression will not be evaluated.
3. For the or−expression, short-circuiting logic is applied if the expression on the left evaluates to true. The right hand expression will not be evaluated.
4. All operations are left associative.

2.9.2 Binary Relational Operators

equal-to-expression:

expression == *expression*

not-equal-to-expression:

expression != *expression*

less-than-expression:

expression < *expression*

greater-than-expression:

expression > *expression*

less-than-equal-to-expression:

expression <= *expression*

greater-than-equal-to-expression:

expression $>=$ *expression*

1. Symbolic expression to perform relational operations meant to do comparisons.
2. All operations are left-associative.

2.9.3 Unary Logical Operators

inversion-expression:

! expression

complement-expression:

~expression

1. Symbolic expression to perform unary logic operations, such as logical complement and logical inversion.

2.10 Bitwise Operations

2.10.1 Binary Boolean Operators

bitwise-and-expression:

expression & expression

bitwise-or-expression:

expression | *expression*

bitwise-xor-expression:

expression ^ *expression*

1. Symbolic expressions to perform logical / bitwise and, or, and exclusive-or operations.
2. All operations are left associative.

2.11 Operator and Expression Precedence

Precedence is defined as follows:

2.12 Expression and Operand Conversions

2.12.1 Boolean Conversions

1. Expressions that are expected to evaluate to booleans for the purposes of Flow Control as defined in 5.3 and for common relational and logical operations as in 2.9 will have their rules checked against the following:
   (a) If the evaluated value is already a boolean, use the value directly.
   (b) If the evaluated value is of an integral type, then any such type which compares equivalent to the integral literal 0 will be false: otherwise, it is true.
   (c) If the evaluated value is of a floating point type, then any such type which compares equivalent to the floating point literal 0.0 will be false: otherwise, it is true.
   (d) Otherwise, if there is no defined conversion, then the program is ill-formed.

2.12.2 Mathematical Conversions

int to float	float variable has the same value as integer.
float to int	integer has largest integral value less than the float.
bool to int	integer has value 1 if true, otherwise it will be 0.
int to bool	bool is true if int is not equal to 0 and false otherwise.

1. Implicit type conversions are carried out only for compatible types. The implicit casting occurs during assignment or when a value is passed as a function argument.

2. The four types of conversions that are supported are summarized in the table below.

Table 2.1: Precedence Table

<table>
<tr><th>Precedence</th><th>Operator</th><th>Variants</th><th>Associativity</th></tr>
<tr><td>1</td><td>++ −−
( )
[ ]
.</td><td>Postfix</td><td>Left to Right</td></tr>
<tr><td>2</td><td>++ −−
+ -
! ~</td><td>Prefix, Unary Operations</td><td>Right to Left</td></tr>
<tr><td>3</td><td>*
/
%</td><td rowspan="9">Binary Operations</td><td rowspan="9">Left to Right</td></tr>
<tr><td>4</td><td>+
-</td></tr>
<tr><td>5</td><td><< >></td></tr>
<tr><td>6</td><td><
<=
>
>=</td></tr>
<tr><td>7</td><td>== !=</td></tr>
<tr><td>8</td><td>&</td></tr>
<tr><td>9</td><td>^</td></tr>
<tr><td>10</td><td>|</td></tr>
<tr><td>11</td><td>||</td></tr>
<tr><td>12</td><td>= += -=
*= /= %=
<<= >>=
&=
^=
|=</td><td>Assignments</td><td>Right-To-Left</td></tr>
</table>

3. If there is no conversion operator defined for those two types exactly, then the program is ill-formed.

Chapter 3

Functions

3.1 Functions and Function Declarations

Functions are independent code that perform a particular task and can be reused across programs. They can appear in any order and in one or many source files, but cannot be split among source files.

Function declarations tell the compiler how a function should be called, while function definitions define what the function does.

3.1.1 Function Definitions

```
fun <identifier> ([<parameter_declarations>]) : <
    return_type> {
     <function_body>
      [return <expression>;]
}
```

1. All function definitions in LéPiX are of the above form where they begin with the keyword fun, followed by the identifier, a list of optional parameter declarations enclosed in parentheses, optionally the return type, and the function body with an optional return statement.

2. return types can be variable types or void.

3. Functions that return void can either omit the return statement or leave it in or return the value unit:

```
fun zero ( &arr:int[] ) : void {
      for (var i : int = 0 to arr.length) {
            arr[i] = 0;
      }
}

fun zero ( &arr:int[] ) : void {
      for (var i : int = 0 to arr.length) {
            arr[i] = 0;
      }
      return;
}
```

4. Functions that return any other variable type must include a return statement and the expression in the return statement must evaluate to the same type as the return type or be convertible to the return type:

```
fun add ( arg1:float , arg2:float ) : float {
      return arg1 + arg2;
}
```

5. In the function add, arg1 and arg2 are passed by value. In the function zero, arr is passed by reference.

6. Function input parameters can be passed by value, for all variable types, or by reference, only for arrays and array derived variable types. See 3.1.3 for more about passing by value and reference.

3.1.2 Function Declarations

1. All function declarations in LéPiX are of the form

```
fun <identifier> ([<parameter_declarations>]) : <
    return_type>;
```

2. The function declaration for the add function from 3.1.1 would be

```
fun add ( arg1:float , arg2:float ) : float;
```

3. Function declarations are identical to function definitions except for the absence or presence of the code body.

4. Function declarations are optional, but useful to include when functions are used across multiple translation units to ensure that functions are called appropriately.

3.1.3 Function Scope and Parameters

1. Variables are declared as usual within the body of a function. The variables declared within the body of a function exist only in the scope of the function and are discarded when they go out of scope.

2. External variables are passed into functions as parameters. All variable types except arrays and array derived variable types are passed by value. Arrays and array derived variable types can be passed by both value and reference.

3. Passing value copies the object, meaning changes are made to the copy within the function and not the original. Passing by reference gives a pointer to the original object to the function, meaning changes are to the original within the function.

4. To pass by value to a function, use the variable name: `add ( x, y );`

5. To pass by reference to a function, use the symbol & and the variable name, as in `zero ( &arr );`.

Chapter 4

Data Types

4.1 Data Types

The types of the language are divided into two categories: primitive types and data types derived from those primitive types. The primitive types are the boolean type, the integral type int, and the floating-point type float. The derived types are struct, Array, and image and pixel, which are both special instances of arrays.

4.1.1 Primitive Data Types

1. `int`

   By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of 2^{32}.

2. `float`

   The float data type is a single precision 32-bit IEEE 754 floating point.

3. `boolean`

   The boolean data type has possible values true and false.

4.1.2 Derived Data Types

Besides the primitive data types, the derived types include arrays, structs, images, and pixels.

1. `array`

   An array is a container object that holds a fixed number of values of a single type. Multi-dimensional arrays are also supported. They need to have arrays of the same length at each level.

2. `pixel`

   A pixel data type is a wrapper for an array that will contain the representation for each pixel of an image. It will contain the rgb values, each as a separate int, and the gray value of a pixel.

3. `image`

   The image data type is just an alias for a 2-dimensional array. The 2-d array will define the size of an image and contains a pixel as each of its data elements.

4. `struct`

   A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help to organize data because they permit a group of related variables to be treated as a unit instead of as separate entities.

Chapter 5

Program Structure and Control Flow

5.1 Statements

1. Any expression followed by a semicolon becomes a statement. For example, the expressions x = 2, lib.save(...), return x become statements:

   ```
   x = 2;
   lib.save(...);
   return x;
   ```

2. The semicolon is used in this way as a statement terminator.

5.2 Blocks and Scope

Braces { and } are used to group statements in to blocks. Braces that surround the contents of a function are an example of grouping statements like this. Statements in the body of a for, while, if or switch statement are also surrounded in braces, and therefore also contained in a block. Variables declared within a block exist only in that block. A semicolon is not required after the right brace.

5.2.1 Blocks

1. At any point in a program, braces can be used to create a block. For example,

```
var x: int = 2;
var y: int = 4;
var result: int;
{
      var z: int = 6
      result = x + y + z;
}
```

2. In this trivial example, the statements on lines 5 and 6 live within their own block.

3. Blocks do have access to named definitions of their surrounding scope, however variables define within a block exist only within that block.

5.2.2 Scope

1. Scopes are defined as the collection of identifiers and available within the current lexicographic block[1].

2. Every program is implicitly surrounded by braces, which define the **global block**.

5.2.3 Variable Scope

1. Variables are in scope only within their own block[2].

2. In the example in Section 5.2.1, z is declared within the braces.

3. Variables declared within blocks last only within lifetime of that block.

4. If we attempted to access z outside of this block, this would cause an error to occur.

5. If a variable with a particular identifier has been declared and the identifier is re-used within a nested block.

[1] This is usually between two curly braces {}
[2] E.g., between the brackets {}

6. The original definition of the identifier is **shadowed** and the new one is used until the end of the block.

7. Variables are constructed, that is, stored in memory when they are first encountered in their scope, and destructed at the scope's end in the reverse order they were encountered in.

5.2.4 Function Scope

1. Function definitions define a new block, which each have their own scope.

2. Function definitions have access to any variables within their surrounding scope, however anything defined in the function definition's block is not accessible in the surrounding blocks.

3. Variables defined in a parameter list belong to the definition-scope of the function.

5.2.5 Control Flow Scope

1. Control flow also introduces a new block with its own scope.

2. Variables initialized in any control flow statement, that is within the parenthesis before the block, belong to the control flow block and are not accessible in the surrounding block.

3. In the statement `for (var x = 0 to 5) { ... }`, `x` only exists within that for loop and destructed after the loop ends.

5.3 Namespaces

1. Namespaces are essentially blocks that allow identifiers to be prefix with an arbitrary nesting of names. They are declared with the `namespace` keyword, followed by several identifiers delimited by a dot '.' symbol.

2. Accessing variables and functions inside of a namespace must have the name of the namespace prefixed before the name of the desired identifier.

3. The namespaces lib and compiler is reserved for use by standard library implementations and the compiler.

4. Namespaces are the only bracket-delimited lexical scope that do not dictate the lifetime of the variables associated with them. These variables are part of the **global scope**.

5.4 if

```
if (expression; expression; ...)
    statements
else
    alternative-statements
```

1. if statements are used to make decisions in control flow.

2. Variations on this syntax are permitted, e.g. The else block of the if statement is optional.

3. If the expression is evaluated and returns true, then the first portion of the if statement is executed. Otherwise, if there is an else the portion after it is executed, and if there is none then the function continues at the next statement.

4. Parenthesis are optional after the if block if there is a single statement. If there are multiple statements, parenthesis are needed.

5. Variables can be initialized inside the expression portion of the if statement as long as the final expression in a semi-colon delimited list evaluates to a Boolean value.

   ```
   if (var x = 20; var y = 50; x < y) {
       statements
   }
   ```

6. The scope for variables x and y is within that particular if statement.

7. If statements can also be nested so that multiple conditions can be tested:

```
if (x < 0)
    y = -1
else if (x > 0)
    y = 1
else
    y = 0
```

5.5 switch

```
switch (variable) {
    case (constant expression):
        statements
    end;
    case (constant expression):
        statements
    end;
    case (constant expression):
        statements
    end;
    default: statements
}
```

1. switch statements can be used as an alternative to a nested if statements.
2. The variable is compared against the constant expression for each case, and if it is equal to this expression then the statements in that case are executed.
3. If the variable does not match any of the cases then the default case is executed.
4. The statements in each case must be followed by an end statement.

As with if statements, if a variable is declared within the switch like switch (var x = other_variable; x) { ... }, the scope for variable x is within that particular switch block.

5.6 while

```
while (expression; expression; ... ; condition) {
      statements
}
```

1. while loops are used to repeat a block of code until some condition is met.
2. Every time a loop condition evaluates to true, the while loop's block and statements are executed.
3. When the condition evaluates to false, the while loop's execution is stopped.
4. Expressions before condition are evaluated only once. For example: `while (var x = 20; x < 30) { ... }` is a valid the while loop, and only the final `x < 30` is evaluated on each loop execution.
5. Loops are dangerous because they can potentially run forever. Make sure your conditions are done properly, or use Flow Control keywords and primitives discussed in 5.8:

   ```
   while (var x = 1; x <= 10) {
         arr[x] = 1;
         x = x + 1;
   }
   ```

5.7 for

```
for (variable = lower_bound to upper_bound by size) {
      statements
}
```

1. For loops are another way to repeat a group of statements multiple times.
2. The by keyword and argument size are optional and used to specify how much the variable should change by each iteration of the loop: `for (x = 1 to 10 by 2) { ... }` will increment x by two each iteration rather than the default value of 1.
3. Variables can be declared in the loop declaration, as in `for (var x = 1 to 10) { ... }`.

4. For loops can also be used to decrement by swapping the positions of the lower_bound and upper_bound arguments, and using a negative value for the size (if using the by keyword) The while loop in Section 5.6 could be expressed as a for loop as follows.

```
for (var x = 1 to 10) {
      arr[x] = 1;
}
```

5. C-style for loops are also supported:

```
for (var x = 1; x <= 10; x++) {
      arr[x] = 1;
}
```

5.8 break and continue

Break and continue statements are used to exit a loop immediately, before the specified condition has been reached.

5.8.1 break

1. Break statements exit the block of a loop immediately.

```
while (...) {
      statements_above
      break;
      statements_below
}
```

2. In the example above, statements_above would be executed only once. The statements_below would never be executed.[3]

5.8.2 break N

1. Break statements can be used to exit nested loops by jumping out of multiple scopes by adding an integral constant after the break keyword.

[3]Break statements are usually included inside of an if statement within the loop to immediately exit on a particular condition.

2. The example below will allow the user to break out of both for loops with only one break statement.[4]

```
for (...) {
    for (...) {
        statements_above
        if (condition)
            break 2;
        statements_below
    }
}
```

5.8.3 continue

1. Continue statements jump to the end of the loop body and begin the next iteration.

```
for (...) {
    statements_above
    if (expressions...) {
        continue;
    }
    statements_below
}
```

2. When a continue statement is executed, statements below the continue keyword are not executed, and the loop post-action and condition are immediately re-evaluated.

3. In the example above, statements_above would always be executed. The statements_below would be executed on iterations where the if condition was false, since when the if condition were true execution would jump back to the for loop's top.

[4] This could be considered a structured version of goto for loops and should be used with the programmer's utmost discretion.

Chapter 6

Parallel Execution

Since a large number of elementary operations in the realm of image processing are embarrassingly parallel matrix operations, the LéPiX language supports a simple parallelization scheme.

6.1 Parallel Execution Model

1. Parallel Execution is when two computations defined by the language are run at the exact same time by the abstract virtual machine, capable of accessing the same memory space.
2. The primary parallel primitive is a parallel-marked block.
3. Use of parallel primitives does not guarantee parallel execution: computation specified to run in parallel may run sequentially.[1]

6.2 Syntax

1. The syntax for parallel code is code simply marked with the keyword `parallel`.

[1] This could be due to hardware limitations, operating system limitations, and other factors of the machine.

2. In the situation where there are variables that must be shared by all the threads, a comma separated list of variable identifiers can be specified in parentheses using the keyword shared as in parallel { <block> }.

3. In the case of nested for loops, only the outermost loop carrying the parallel keyword is parallel.

6.3 Threads

1. The code inside of a parallel block can be dispatched to multiple executing threads.

2. Each thread that is spawned in this way will have its own scope, which is created when the thread is spawned and destroyed when the thread is killed.

3. Each thread has its own copy of each variable that is declares within the scope of the loop statements.

4. All variables are shared by default, except the ones declared in the parallel scope.

Part III

Grammar Specification

Chapter 7

Grammar

7.1 Lexical Definitions and Conventions

A program consists of one or more translation units, which are translated in two phases, namely the preprocessing step and the lexing step. The preprocessing step entails carrying out directives which begin with # in a C-like style. The lexing step reduces the program to a sequence of tokens.

7.1.1 Tokens

1. Tokens belong to _categories. These are whitespace, keywords, operators, integer literals, floating point literals, string literals, identifiers, and brackets.

2. Whitespace tokens are used to separate other tokens and are ignored in any case where they do not occur between other non-whitespace tokens.

7.1.2 Comments

1. Comments come in two flavors: single-line and multi-line.

2. Single line comment begin with // and continue until the next newline character is found. Multi-line comments begin with /* and end with

*/. They are nested.

3. Comments are treated as whitespace tokens, but for various purposes may still appear between other whitespace tokens in a program's token stream.

7.1.3 Identifiers

1. Identifiers are composed of letters, numbers and the underscore character (_) but must begin with a letter. Identifiers beginning with underscores and numbers will be reserved for use within the implementation of the language.

7.1.4 Keywords

1. A set of identifiers has been reserved for use as keywords and cannot be used in other cases. The list of keywords is in the table below.

int	float	void	bool
unit	char	codepoint	string
vector	matrix	vector	vec
var	let	if	else
for	while	by	to
return	true	false	mutable
const	fun	struct	maybe
protected	public	private	shared
as	of	parallel	atomic

7.1.5 Literals

1. Literals are of three types: integer literals, floating literals, and string literals, as detailed in 2.2.1. All of them use the following definitions for their digits:

   ⟨*decimal-digit*⟩ ::= one of
   0 1 2 3 4 5 6 7 8 9

$\langle$*hexidecimal-digit*$\rangle$ ::= one of
0 1 2 3 4 5 6 7 8 9
A B C D E F
a b c d e f

$\langle$*binary-digit*$\rangle$ ::= one of
0 1

$\langle$*octal-digit*$\rangle$ ::= one of
0 1 2 3 4 5 6 7

$\langle$*n-digit*$\rangle$ ::= one of
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
'_'

$\langle$*decimal-digit-sequence*$\rangle$ ::= $\langle\rangle$
| $\langle$*decimal-digit*$\rangle$ $\langle$*decimal-digit-sequence*$\rangle$

$\langle$*binary-digit-sequence*$\rangle$ ::= $\langle\rangle$
| $\langle$*binary-digit*$\rangle$ $\langle$*binary-digit-sequence*$\rangle$

$\langle$*octal-digit-sequence*$\rangle$ ::= $\langle\rangle$
| $\langle$*octal-digit*$\rangle$ $\langle$*octal-digit-sequence*$\rangle$

$\langle$*hexidecimal-digit-sequence*$\rangle$ ::= $\langle\rangle$
| $\langle$*hexidecimal-digit*$\rangle$ $\langle$*hexidecimal-digit-sequence*$\rangle$

$\langle$*n-digit-sequence*$\rangle$ ::= $\langle\rangle$
| $\langle$*n-digit*$\rangle$ $\langle$*n-digit-sequence*$\rangle$

2. Integer literals consist of sequences of digits are always interpreted as decimal numbers. They can be represented by the following lexical compositions:

$\langle$*integer-literal*$\rangle$ ::= $\langle$*decimal-literal*$\rangle$
| $\langle$*binary-literal*$\rangle$
| $\langle$*octal-literal*$\rangle$
| $\langle$*hexidecimal-literal*$\rangle$
| $\langle$*n-digit-literal*$\rangle$

$\langle$*decimal-literal*$\rangle$ ::= $\langle$*decimal-digit-sequence*$\rangle$

⟨*binary-literal*⟩ ::= 0b ⟨*binary-digit-sequence*⟩
 | 0B ⟨*binary-digit-sequence*⟩

⟨*octal-literal*⟩ ::= 0c ⟨*octal-digit-sequence*⟩
 | 0C ⟨*octal-digit-sequence*⟩

⟨*hexidecimal-literal*⟩ ::= 0x ⟨*hexidecimal-digit-sequence*⟩
 | 0X ⟨*hexidecimal-digit-sequence*⟩

⟨*n-digit-literal*⟩ ::= 0n ⟨*n-digit-sequence*⟩ '#'
 | 0N ⟨*n-digit-sequence*⟩

3. Floating point literals can be specified using digits and a decimal points or in scientific notation. The following regular expression represents the set of acceptable floating-point constants.

   ⟨*e-part*⟩ ::= e ⟨+ |−″ -⟩ ⟨*integral-literal*⟩

   ⟨*floating-literal*⟩ ::= ⟨*integral-literal*⟩$_{opt}$. ⟨*integral-literal*⟩$_{opt}$ ⟨*e-part*⟩$_{opt}$
    | ⟨*integral-literal*⟩ .$_{opt}$ ⟨*integral-literal*⟩$_{opt}$ ⟨*e-part*⟩

4. String literals are sections of quote-delimited items. They are defined as follows:

   ⟨*single-quote*⟩ ::= '

   ⟨*double-quote*⟩ ::= "

   ⟨*raw-specifier*⟩ ::= R$_{opt}$

   ⟨*character*⟩ ::= ⟨*escape-character*⟩ ⟨*source-character*⟩

   ⟨*character-sequence*⟩ ::= ⟨⟩
    | ⟨*character*⟩ ⟨*character-sequence*⟩

   ⟨*string-literal*⟩ ::= ⟨*raw-specifier*⟩ ⟨*double-quote*⟩ ⟨*character-sequence*⟩ ⟨*double-quote*⟩
    | ⟨*raw-specifier*⟩ ⟨*single-quote*⟩ ⟨*character-sequence*⟩ ⟨*single-quote*⟩

7.2 Expressions

The following sections formalize the types of expressions that can be used in a LéPiX program and also specify completely, the precedence of operators and left or right associativity.

7.2.1 Primary Expression

⟨*primary_expression*⟩ ::= ⟨*identifier*⟩
| ⟨*integer-constant*⟩
| ⟨*float-constant*⟩
| (expression)

1. A primary expression are composed of either a constant, an identifier, or an expression in enclosing parentheses.

7.2.2 Postfix Expressions

⟨*postfix_expression*⟩ ::= ⟨*primary_expression*⟩
| ⟨*postfix_expresion*⟩ (argument_list)
| ⟨*postfix_expression*⟩ [expression]
| ⟨*postfix_expression*⟩ . identifier ⟨*argument_list*⟩ ::= ⟨⟩
| ⟨*argument_list*⟩ , ⟨*postfix_expression*⟩

1. A postfix expression consist of primary expression followed by postfix operators. The operators in postfix expressions are left-associative.

Indexing

1. Array indexing consists of a postfix expression, followed by an expression enclosed in square brackets. The expression in the brackets must evaluate to an integer which will represent the index to be accessed.
2. The value returned by indexing is the value in the array at the specified index.

Function Calls

1. A function call is a postfix expression (representing the name of a defined function) followed by a (possibly empty) list of arguments enclosed in parentheses.
2. The argument list is represented as a comma separated list of postfix expressions.

Structure access

1. The name of a structure followed by a dot and an identifier name is a postfix expression. The expression's value is the named member's of the structure that is being accessed.

7.2.3 Unary Expression

⟨*unary_operator*⟩ ::= ˜
  | !
  | -
  | *

⟨*unary_expression*⟩ ::= ⟨*unary_operator*⟩ ⟨*postfix_expression*⟩

1. A unary expression consists of ¡postfix_expression¿ preceded by a unary operator (- ,˜ , !, * ,&).
2. Unary expressions are left-associative.
3. The unary operation is carried out after the postfix expression has been evaluated.

The function of each unary operator has been summarized in the table below:

-	Unary minus
˜	Bitwise negation operator
ˆ	Logical negation operator
*	Indirection operator

7.2.4 Casting

The LéPiX language supports the casting of an integer to a floating point value and vice versa. It also supports casting of an integer value to a boolean value and vice versa. Integer to float casting creates a floating point constant with the same value as the integer. Casting a floating point value to an integer rounds down to the nearest integral value. Casting a boolean value to an integer gives 1 if the value is true and 0 if it is false. Casting an integer to a boolean value yields false if the value is 0 and true otherwise.

⟨*cast_expression*⟩ ::= ⟨*unary_expression*⟩
  | ⟨*unary_expression*⟩ as ⟨*type_name*⟩

7.2.5 Multiplicative Expressions

The multiplication (*), division (/) and modulo (%) operators are left associative.

⟨*multiplicative_expression*⟩ ::= ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ * ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ / ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ % ⟨*cast_expression*⟩

7.2.6 Additive Expressions

The addition (+) and subtraction (-) operators are left associative.

⟨*additive_expression*⟩ ::= ⟨*multiplicative_expression*⟩
  | ⟨*additive_expression*⟩ + ⟨*cast_expression*⟩
  | ⟨*additive_expression*⟩ - ⟨*cast_expression*⟩

7.2.7 Relational Expressions

The relational operators less than (¡), greater than (¿), less than or equal to (¡=) and greater than or equal to (¿=) are left associative.

⟨*relational_expression*⟩ ::= ⟨*additive_expression*⟩
  | ⟨*relational_expression*⟩ ⟨ ¡*additive_expression*⟩
  | ⟨*relational_expression*⟩ ⟨= ¡*additive_expression*⟩
  | ⟨*relational_expression*⟩ ¿ ⟨*additive_expression*⟩
  | ⟨*relational_expression*⟩ ¿= ⟨*additive_expression*⟩

7.2.8 Equality Expression

⟨equality_expression⟩ ::= ⟨relational_expression⟩
| ⟨equality_expression⟩ != ⟨relational_expression⟩
| ⟨equality_expression⟩ == ⟨relational_expression⟩

7.2.9 Logical AND Expression

⟨logical_and_expression⟩ ::= ⟨equality_expression⟩
| ⟨logical_and_expression⟩ && ⟨equality_expression⟩

1. The logical and operator (&&) is left associative and returns true if both its operands are not equal to false.

7.2.10 Logical OR Expression

⟨logical_or_expression⟩ ::= ⟨logical_and_expression⟩
| ⟨logical_or_expression⟩ || ⟨logical_and_expression⟩

1. The logical OR operator (——) is left associative and returns true if either of its operands are not equal to false.

7.2.11 Assignment Expressions

⟨assignment_expression⟩ ::= ⟨logical_or_expression⟩
| ⟨unary_expression⟩ = ⟨assignment_expression⟩

1. The assignment operator (=) is left associative.

7.2.12 Assignment Lists

⟨assignment_list⟩ ::= ⟨assignment_expression⟩
| ⟨assignment_list⟩ ::= ⟨assignment_list⟩ , ⟨assignment_expression⟩

1. Assignment lists consist of multiple assignment statements separated by commas.

7.2.13 Declarations

⟨declaration⟩ ::= let ⟨storage_class⟩ ⟨identifier⟩ : ⟨type_name⟩ = ⟨postfix_expression⟩
  | var ⟨storage_class⟩ ⟨identifier⟩ : ⟨type_name⟩ = ⟨postfix_expression⟩
  | ⟨declaration⟩ ⟨array⟩

⟨storage_class⟩ ::= mutable
  | const

⟨type_name⟩ ::= void
  | unit
  | bool
  | int
  | float
  | ⟨type_name⟩ ⟨array⟩

⟨array⟩ ::= [⟨int_list⟩]
  | [⟨array⟩]

⟨int_list⟩ ::= ⟨integer⟩
  | ⟨int_list⟩ , ⟨integer⟩

1. Declarations of a variable specify a type for each identifier and a value to be assigned to the identifier.
2. Declarations do not always allocate memory to be associated with the identifier.

7.2.14 Function Declaration

⟨function_declaration⟩ ::= fun ⟨identifier⟩ (⟨params_list⟩) : ⟨type_name⟩

⟨params_list⟩ ::= ⟨ ⟩
  | ⟨identifier⟩ : ⟨type_name⟩
  | ⟨params_list⟩ , ⟨identifier⟩ : ⟨type_name⟩

1. Function declarations consist of the keyword fun followed by an identifier and a list of parameters enclosed in parentheses.
2. The list of arguments is followed by a colon and a type name which represents the return type for the function.
3. The arguments list is specified as a comma-separated list of identifier, type pairs.

7.3 Statements

Statements are executed sequentially in all cases except when explicit constructs for parallelization are used. Statements do not return values.

⟨*statement*⟩ ::= ⟨*expression_statement*⟩
  | ⟨*branch_statement*⟩
  | ⟨*compound_statement*⟩
  | ⟨*iteration_statement*⟩
  | ⟨*return_statement*⟩

7.3.1 Expression Statements

⟨*expression_statement*⟩ ::= ⟨ ⟩
  | ⟨*expression*⟩ ;

1. Expression statements are either empty or consist of an expression.
2. These effects of one statement are always completed before the next is executed.
3. This guarantee is not valid in cases where explicit parallelization is used.
4. Empty expression statements are used for loops and if statements where not action is to be taken.

7.3.2 Statement Block

$\langle block\rangle$ ::= { $\langle compound\_statement\rangle$ }
| { $\langle block\rangle$ $\langle compound\_statement\rangle$}

$\langle compound\_statement\rangle$ ::= $\langle declaration\rangle$
| $\langle statement\rangle$
| $\langle compound\_statement\rangle$; $\langle declaration\rangle$
| $\langle compound\_statement\rangle$; $\langle statement\rangle$

1. A statement block is a collection of statements declarations and statements.
2. If the declarations redefine any variables that were already defined outside the block, the new definition of the variable is considered for the execution of the statements in the block.
3. Outside the block, the old definition of the variable is restored.

Branch Statements

$\langle branch\_statement\rangle$ ::= if ($\langle expression\rangle$) $\langle statement\rangle$ fi
| if ($\langle expression\rangle$) $\langle statement\rangle$ else $\langle statement\rangle$ fi

1. Branch statement are used to select one of several statement blocks based on the value of an expression.

7.3.3 Loop Statements

$\langle loop\_statement\rangle$ ::= while ($\langle expression\rangle$) $\langle statement\rangle$
| for ($\langle identifier\rangle$ = $\langle expression\rangle$ to $\langle expression\rangle$) $\langle statement\rangle$
| for ($\langle assignment\_expression\rangle$ = $\langle expression\rangle$ to $\langle expression\rangle$ by $\langle expression\rangle$) $\langle statement\rangle$
| for ($\langle expression\rangle$; $\langle expression\rangle$; expression¿) $\langle statement\rangle$

1. Loop statements specify the constructs used for iteration and repetition.

7.3.4 Jump Statements

$\langle jump\_statement \rangle$ ::= break $\langle integer\text{-}literal \rangle_{opt}$
| continue

1. Jump statements are used to break out of a loop or to skip the current iteration of a loop.

7.3.5 Return Statements

$\langle return\_statement \rangle$ ::= return
| return $\langle expression \rangle$

1. Return statements are used to denote the end of function logic and the also to specify the value to be returned by a call to the function in question.

7.4 Function Definitions

$\langle function\_definition \rangle$ ::= $\langle function\_declaration \rangle$ $\langle block \rangle$

1. Function definitions consist of a function declaration followed by a statement block.

7.5 Preprocessor

$\langle preprocessor\_directive \rangle$ ::= #define $\langle identifier \rangle$ $\langle expression \rangle$
| #ifdef $\langle identifier \rangle$
| #ifndef $\langle identifier \rangle$
| #endif
| #import $\langle identifier \rangle$
| #import "$\langle file\_name \rangle$"
| #import string $\langle file\_name \rangle$

1. Before the source for a LePix program is compiled, the program is consumed by a preprocessor, which expands macro definitions and links libraries and other user-defines to the current file, as specified by appropriate preprocessor directives.

2. define macros create an alias for a value or expression.

3. ifdef and ifndef macros are used to check if a particular alias has already been assigned. Import directives are used to link files/libraries with the current program.

7.6 Grammar Listing

⟨*primary_expression*⟩ ::= ⟨*identifier*⟩
| ⟨*integer-constant*⟩
| ⟨*float-constant*⟩
| (expression)

⟨*postfix_expression*⟩ ::= ⟨*primary_expression*⟩
| ⟨*postfix_expresion*⟩ (argument_list)
| ⟨*postfix_expression*⟩ [expression]
| ⟨*postfix_expression*⟩ . identifier

⟨*argument_list*⟩ ::= ⟨⟩
| ⟨*argument_list*⟩ , ⟨*postfix_expression*⟩

⟨*unary_operator*⟩ ::= ~
| !
| -
| *

⟨*unary_expression*⟩ ::= ⟨*unary_operator*⟩ ⟨*postfix_expression*⟩

⟨*cast_expression*⟩ ::= ⟨*unary_expression*⟩
| ⟨*unary_expression*⟩ as ⟨*type_name*⟩

⟨*multiplicative_expression*⟩ ::= ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ * ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ / ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ % ⟨*cast_expression*⟩

⟨*additive_expression*⟩ ::= ⟨*multiplicative_expression*⟩
| ⟨*additive_expression*⟩ + ⟨*cast_expression*⟩
| ⟨*additive_expression*⟩ - ⟨*cast_expression*⟩

⟨*relational_expression*⟩ ::= ⟨*additive_expression*⟩
| ⟨*relational_expression*⟩ < ⟨*additive_expression*⟩
| ⟨*relational_expression*⟩ <= ⟨*additive_expression*⟩
| ⟨*relational_expression*⟩ > ⟨*additive_expression*⟩
| ⟨*relational_expression*⟩ >= ⟨*additive_expression*⟩

⟨*equality_expression*⟩ ::= ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ != ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ == ⟨*relational_expression*⟩

⟨*logical_and_expression*⟩ ::= ⟨*equality_expression*⟩
| ⟨*logical_and_expression*⟩ && ⟨*equality_expression*⟩

⟨*logical_or_expression*⟩ ::= ⟨*logical_and_expression*⟩
| ⟨*logical_or_expression*⟩ || ⟨*logical_and_expression*⟩

⟨*assignment_expression*⟩ ::= ⟨*logical_or_expression*⟩
| ⟨*unary_expression*⟩ = ⟨*assignment_expression*⟩

⟨*assignment_list*⟩ ::= ⟨*assignment_expression*⟩
| ⟨*assignment_list*⟩ ::= ⟨*assignment_list*⟩ , ⟨*assignment_expression*⟩

⟨*declaration*⟩ ::= var ⟨*storage_class*⟩ ⟨*identifier*⟩ : ⟨*type_name*⟩ = ⟨*postfix_expression*⟩

⟨*type_name*⟩ ::= bool
| int
| float
| ⟨*type_name*⟩ ⟨*array*⟩

⟨*array*⟩ ::= [⟨*int_list*⟩]
| [⟨*array*⟩]

⟨*int_list*⟩ ::= ⟨*integer*⟩
| ⟨*int_list*⟩ , ⟨*integer*⟩

⟨*function_declaration*⟩ ::= fun ⟨*identifier*⟩ (⟨*params_list*⟩) : ⟨*type_name*⟩

⟨*params_list*⟩ ::= ⟨ ⟩
  | ⟨*identifier*⟩ : ⟨*type_name*⟩
  | ⟨*params_list*⟩ , ⟨*identifier*⟩ : ⟨*type_name*⟩

⟨*statement*⟩ ::= ⟨*expression_statement*⟩
  | ⟨*branch_statement*⟩
  | ⟨*compound_statement*⟩
  | ⟨*iteration_statement*⟩
  | ⟨*return_statement*⟩

⟨*expression_statement*⟩ ::= ⟨ ⟩
  | ⟨*expression*⟩ ;

⟨*block*⟩ ::= { ⟨*compound_statement*⟩ }
  | { ⟨*block*⟩ ⟨*compound_statement*⟩}

⟨*compound_statement*⟩ ::= ⟨*declaration*⟩
  | ⟨*statement*⟩
  | ⟨*compound_statement*⟩ ; ⟨*declaration*⟩
  | ⟨*compound_statement*⟩ ; ⟨*statement*⟩

⟨*parallel_block*⟩ ::= parallel (⟨*parallel_control_variables*⟩) ⟨*block*⟩

⟨*branch_statement*⟩ ::= if (⟨*expression*⟩) ⟨*statement*⟩ fi
  | if (⟨*expression*⟩) ⟨*statement*⟩ else ⟨*statement*⟩ fi

⟨*branch_statement*⟩ ::= if (⟨*expression*⟩) ⟨*statement*⟩ fi
  | if (⟨*expression*⟩) ⟨*statement*⟩ else ⟨*statement*⟩ fi

⟨*loop_statement*⟩ ::= while (⟨*expression*⟩) ⟨*statement*⟩
  | for (⟨*identifier*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩) ⟨*statement*⟩
  | for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩ by ⟨*expression*⟩) ⟨*statement*⟩
  | for (⟨*expression*⟩; ⟨*expression*⟩; expression¿) ⟨*statement*⟩

⟨*identifier_list*⟩ ::= ⟨*identifier*⟩
  | ⟨*identifier_list*⟩ , ⟨*identifier*⟩

⟨*jump_statement*⟩ ::= break ⟨*integer-literal*⟩$_{opt}$
  | continue

⟨*return_statement*⟩ ::= return
  | return ⟨*expression*⟩

Chapter 4

Plan

Our plan was developed slowly and mostly solidified around the making of our Language Reference Manual and a bit afterwards. We met once a week, sometimes a second time if our Advisor had the time for it, and occasionally held extra meetings to help get things done.

4.1 Process

Most of our planning was done in-person via weekly meetings. We also used Github Issues to track things and also bikeshed some of our progress and implementation. We closed issues as they passed and had issues tied to Milestones in the project:

4.2 Timeline

Our timeline was given by the milestones we had for the project. We opened them early, meaning that one of the milestones (GPU Codegen) was scrapped when our team decided that we would not pursue such an avenue.

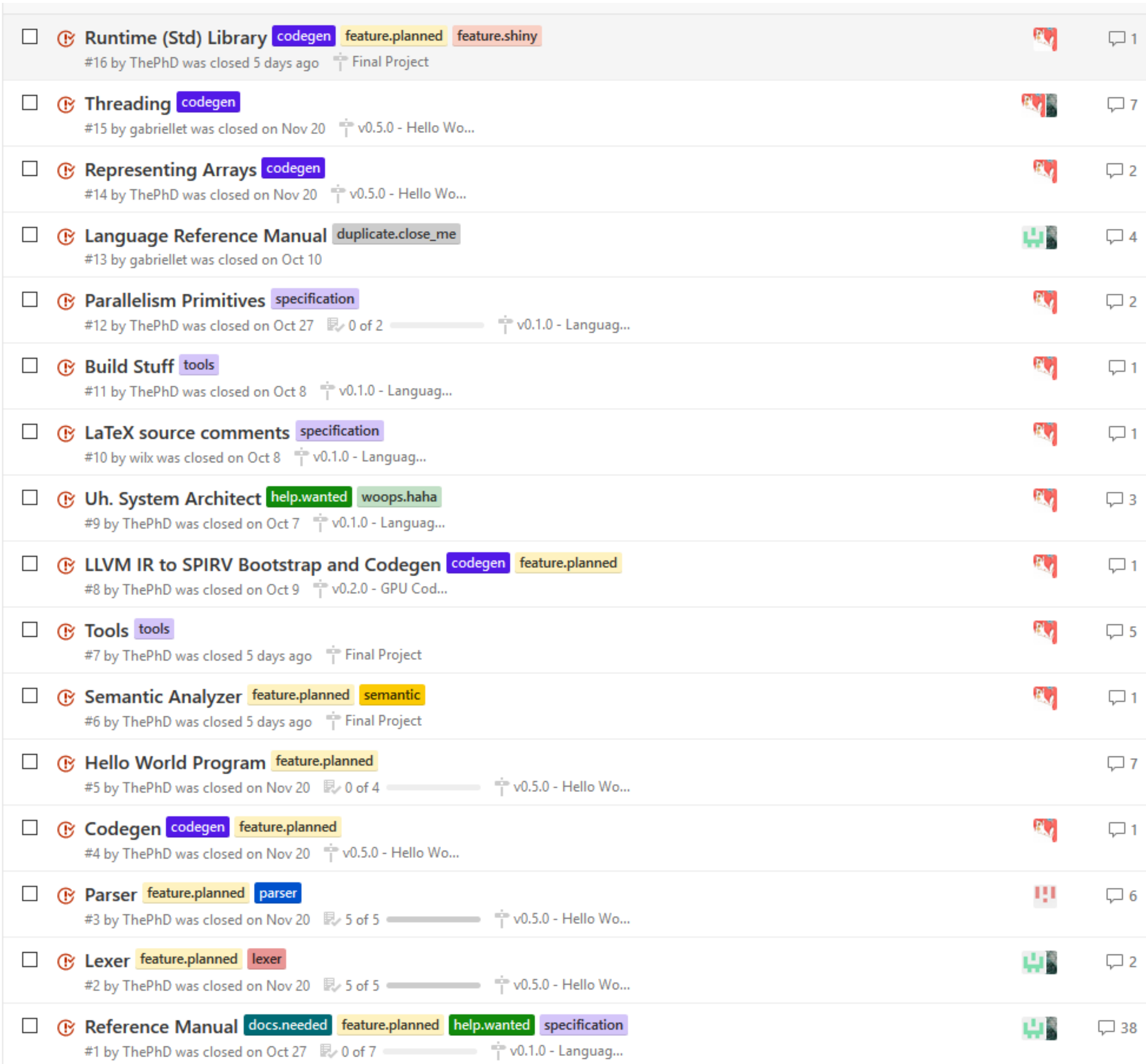

Figure 4.1: Closed issues throughout the project `https://github.com/ThePhD/lepix/issues?q=is%3Aissue+is%3Aclosed`.

4.3 Tools

Everyone was free to develop in whatever IDE or editor they wished, just so long as they could invoke the makefile. As I was originally the System Architect, I put together a list of all the tools someone would need to invoke the build process in Figure 4.3. The command line dependencies here helped me figure out what was needed when we started to do testing 6.

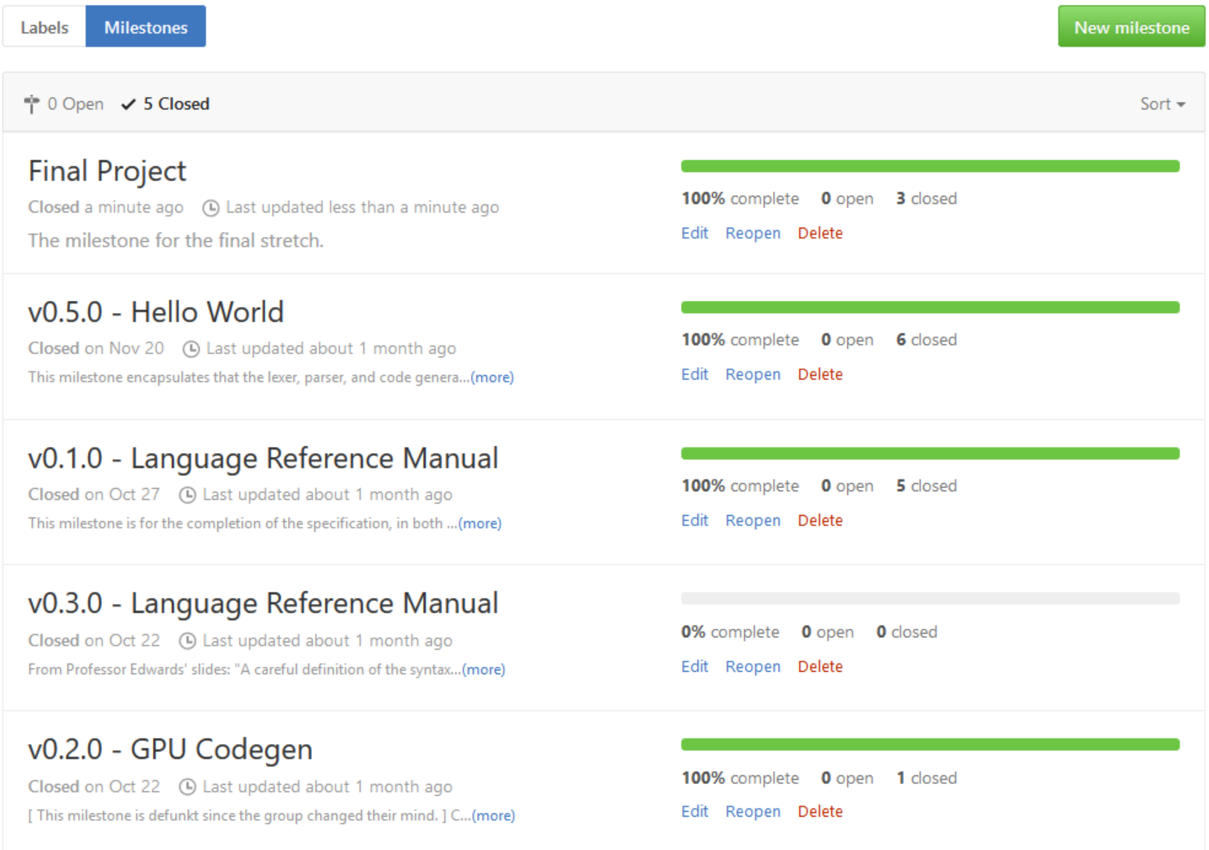

Figure 4.2: Milestones for the project `https://github.com/ThePhD/lepix/milestones?state=closed`.

4.4 Project Log

Asides from issues being closed and comments being made, the best project log that shows how I did is the git commit log for all the branches, included below. It was generated from git using the command git −−no−pager log −−graph −−abbrev−commit −−decorate −−date=relative −−all.

```
* commit 5d983f7 (HEAD -> master, origin/master, origin/HEAD)
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   27 minutes ago
|
|     Final clean implementation of bottom-type type derivation
  for function returns, good literals, and overloading
|
* commit e323adc
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   22 hours ago
```

```
|
|     overloading tests among other things
|
* commit 714d07f
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 days ago
|
|     last pdfs and reports and readme update
|
* commit aa558db
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 days ago
|
|     remove temporaries and debug print statements
|
* commit a7e8fa2 (origin/feature/semantic, feature/semantic)
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 days ago
|
|     Buh.
|
* commit 816bad8
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   4 days ago
|
|     All I have left....
|
* commit ecbdf74
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   4 days ago
|
|     We're so close to the end. Don't give up. Trust in yourself, and fight for what was right...
|
* commit fb3b94d
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   8 days ago
|
|     It finally builds... the basics of the Semantic AST, finally more or less in place...!
|
* commit c00e2ab
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   2 weeks ago
|
|     Beef up the semantic AST and fully complete the pretty printer for it.
|
* commit 817f6b8
```

```
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 weeks ago
|
|     Ensure lowercase acceptance as well.
|
* commit 79fc476
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 weeks ago
|
|     Fixing up the parser for integer literals and other more useful things.
|
* commit 2bfb992
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 weeks ago
|
|     scanner and parser are up to snuff
|
* commit 77847d7
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 weeks ago
|
|     handle extra cases in the parser for increment, decrement, and assignment-ops
|
* commit 92ee4d6
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 weeks ago
|
|     Beat up the parser lots.
|
*   commit 4eb1aeb
|\  Merge: 5efaebe 989e1d2
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Merge branch 'master' into feature/semantic
| |
| *   commit 989e1d2
| |\  Merge: 9ccd7e0 bebe5ad
| | | Author: Jackie Lin <jackielin13@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     Merge branch 'master' of https://github.com/ThePhD/lepix
| | |
| * | commit 9ccd7e0
| | | Author: Jackie Lin <jackielin13@gmail.com>
| | | Date:   3 weeks ago
```

```
| | |
| | |     tests
| | |
* | | commit 5efaebe
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     [ci skip] heavily modify the parser to handle type qualifications, improve the AST, and begin to consider scoping rules
| | |
* | | commit 6c4629a
| |/  Author: ThePhD <phdofthehouse@gmail.com>
|/|   Date:   3 weeks ago
| |
| |       [ci skip] commit so I can jump back to helping on master
| |
* | commit bebe5ad (origin/testing/travis, testing/travis)
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     REALLY fuck you, python3
| |
* | commit d5cb6ef
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Fuck you too, python3
| |
* | commit fc38fe8
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     specifically invoke python3 because environments are stupid?
| |
* | commit a10dfaa
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     51 builds later, it should work...
| |
* | commit 38f438b
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     ensure only python 3 is available on the system
| |
```

```
* | commit 7eb9368
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Only need one of either --rm or -d
| |
* | commit d2f844f
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Proper travis ci with safety net for made directories for tests
| |
* | commit aebffd0
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     make sure the shell is configured with eval opam...
| |
* | commit cc6dca8
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     This is getting a tad tiresome, but it's my fault for not having a good handle on travis-ci
| |
* | commit cbbbe02
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Proper escaping?
| |
* | commit df0a2f1
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     escaped operators
| |
* | commit 48edeca
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Party with the cd comms
| |
* | commit 14788e0
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     "Docker never dies!" (Sleep infinity)
```

```
| |
* | commit 35af11a
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Attemping without heredoc and just docker exec...
| |
* | commit d9d03ee
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     No -c on bash when using heredoc
| |
* | commit 2f6003c
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Attempting to make docker behave better?
| |
* | commit 9d0cf03
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     explicit printing
| |
* | commit c6e8299
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     Fix the tests because I'm bad at writing python code, weee
| |
* | commit f27c410
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   3 weeks ago
| |
| |     blot out the Semantic Stuff until its time
| |
* |   commit 71c87ad
|\ \   Merge: b032732 b55ae1c
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     Merge branch 'feature/semantic' into testing/travis
| | |
| | |     # Conflicts:
| | |     #    .travis.yml
| | |
| * | commit b55ae1c
```

```
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     default llvm fails because Opam is a heaping pile of shit
| | |
| * | commit 7a13678
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     Attempt to properly propogate bash errors and fix travis files
| | |
| * | commit 2ae921c
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     update test harness
| | |
| * | commit 81fb441
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     semantic analyizer start
| | |
| * | commit a9011e3 (origin/feature/preprocessor, feature/preprocessor)
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     fix gitignore
| | |
| * | commit 11b86f8
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     Full preprocessor implementation
| | |
| * | commit 63a06f6
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   3 weeks ago
| | |
| | |     update gitignore
| | |
* | | commit b032732
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   4 weeks ago
| | |
| | |     Go fuck yourself, travis, and your rules against tabs
```

```
| | |
* | | commit 75b8679
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   4 weeks ago
| | |
| | |     Let's try this again...
| | |
* | | commit 1e827e2
|/ /  Author: ThePhD <phdofthehouse@gmail.com>
| |   Date:   4 weeks ago
| |
| |       messing around with docker
| |
* | commit 01adc38
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     One more missing package from depext
| |
* | commit 8fa39e9
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     package names were wrong
| |
* | commit 337c74b
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     source script to propogate errors better
| |     autoshit and mcrap tools need to be there
| |
* | commit 871197c
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     travis gooo
| |
* | commit 868dc82
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     assume yes for ALL cases...
| |
* | commit 81d341a
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     say yes, all the time
```

```
| |
* | commit b4dba58
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     update ignore
| |
* | commit 5176344
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     update properly
| |
* | commit e9fb2af
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Keeep trying with docker....
| |
* | commit 7b41eef
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Poking at things 'till it works...
| |
* | commit f4c7c3b
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Properly bind the mount volume with the -v command, then swap into it
| |
* | commit d6b090f
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     poke at env to understand what's going on
| |
* | commit 667d0d1
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     travis_run file and friends
| |
* | commit 2e8aab2
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     try it from a file now...
```

```
| |
* | commit 83afd3b
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     super duper docker
| |
* | commit 78f655d
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Goddamn tabs
| |
* | commit cca9cd2
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     update travis work
| |
* | commit c0e85a3
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Use a different language to attempt to stay out of the python shell
| |
* | commit 9e00c5f
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     update tests file and travis CI yaml file
| |
* | commit 8b71e6d
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Add .travis file to start CI
| |
* | commit 0f25039 (origin/feature/codegen, feature/codegen)
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Testing fixture
| |
* | commit e200757
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
```

```
| |     Example code for linking an external library. Various small changes to the driver of lepix and the codegen in preparation for the Semantic Analyzer and the AST.
| |     We still need something to preprocess source code... another regular parser, perhaps?
| |
* | commit f6a208d
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     Re-raise any bad errors we don't know how to catch
| |
* | commit 6b73260
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   4 weeks ago
| |
| |     CARAT DIAGNOSTICS YEEEAAH
| |
* | commit 07c1a08
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     better polyfill code
| |
* | commit cf4f1a5
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Better options parser, again, mostly for the sake of writing clearer, better code
| |
* | commit c50176d
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     properly guard additions to sub
| |
* | commit 9c6eca8
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     More functional string_split
| |
* | commit 251c6ea
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     "cleaner" polyfill...?
| |
```

```
* | commit 058cd89
|/  Author: ThePhD <phdofthehouse@gmail.com>
|   Date:   5 weeks ago
|
|       goofing off with trying to write better functional code
|
* commit f5cc25f
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     Full on driver and options implementation
|     Polyfill layer to replace any missing batteries / core stuff
|     IO layer for opening and writing to a file
|
* commit e97a9c4
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     More comments, restructuring, and lexer-error handling.
|
* commit 5933da5
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     We now have a driver that handles the code
|     There is now an option to print out the token stream
|     The lepix top level performs a basic amount of error handling now
|     The parser and lexer now do a very thorough job of tracking line information; may want to propogate into the AST somehow
|
* commit b60947a
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     It might be beneficial to mess with how the lexing and parsing are run through, so we can generate the proper line numbers and token lists.
|     We should also look into reading from and writing to files, even if we don't have the Batteries library and other bits set up for this.
|     One day...
|
* commit 5fb47e1
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
```

```
|     counted arrays, namespace declarations and proper parallel binding declarations
|
* commit ce06c8f
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     It works.
|     Now I'm going to redo the whole goddamn AST and Parser so we can really get going...
|
* commit 82b848f
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     Segmentation fault.
|     S E G M E N T A T I O N F A U L T L A D I E S .
|
*   commit c35d50d
|\  Merge: 677d5bb c8e22b9
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Merge branch 'feature/codegen'
| |
| * commit c8e22b9
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     skeleton of semantic analyzer
| |
* | commit 677d5bb
|/  Author: fennilin <jackielin13@gmail.com>
|   Date:   5 weeks ago
|
|       Create 11-17-16
|
* commit f736ab5
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   5 weeks ago
|
|     properly append qualified id to list
|
*   commit 748f5c0
|\  Merge: a347044 e120364
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Merge branch 'feature/codegen'
```

```
| |
| |     # Conflicts:
| |     # .gitignore
| |
| * commit e120364
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Allow for interwoven function and data declarations
| |     Properly concatenate qualified IDs
| |     Start on code generation (nothing actually appears)
| |     Make sure top-level does not trigger semantic analyzer (its empty right now)
| |
| * commit 018952e
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Array type now takes a number plus a type, rather than having a separate type for each one
| |     scratch source example that can be modified and committed to any current contention for a person working on the compiler
| |
| * commit e8b60ed
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     More ignore files and an empty main test.
| |
| * commit f5dc624
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Remove built files (please don't commit these again...)
| |
| * commit 6f0db16
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   5 weeks ago
| |
| |     Clean up these commits...
| |
| *   commit 9b1384b
| |\   Merge: ae4b9f6 907c6b0
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   5 weeks ago
| | |
| | |     Merge remote-tracking branch 'origin/master' into feature/codegen
```

```
| | |
| | |      # Conflicts:
| | |      #     source/parser.mly
| | |      #     source/scanner.mll
| | |
| * | commit ae4b9f6
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   5 weeks ago
| | |
| | |     Add qualified ID handling (we will improve it later to handle arbitrarily long strings)
| | |
* | | commit a347044
| |/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
|/|   Date:   5 weeks ago
| |
| |       Edited .gitignore to omit built files in the source directory
| |
* | commit 907c6b0
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   5 weeks ago
| |
| |     Added atomic statement blocks
| |
* | commit 5956c1b
|/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
|   Date:   5 weeks ago
|
|       Simple parallel blocks (without atomic sections) and array literals now work in Parser+AST
|
* commit 7cc276a
| Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| Date:   5 weeks ago
|
|     AST complete with mildly-pretty printing
|
* commit d1dcd73
| Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| Date:   5 weeks ago
|
|     JK function declarations work too LOL
|
* commit f810dd3
| Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| Date:   5 weeks ago
|
```

```
|     Almost done with AST and pretty printing for all language constructs. Only function and variable decls to go
|
* commit 71a6fcb
| Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| Date:   5 weeks ago
|
|     Added simple top level. Edited ast, parser and lexer but some errors remain. Pretty printing needs to be set up.
|
* commit d29fa01
| Author: Fatima <fatimakoli14@gmail.com>
| Date:   6 weeks ago
|
|     Parallelblock and jump statements added
|
*   commit 772f3b9
|\  Merge: e039350 2a939a4
| | Author: Fatima <fatimakoli14@gmail.com>
| | Date:   6 weeks ago
| |
| |     Merge branch 'master' of https://github.com/ThePhD/lepix
| |
| * commit 2a939a4
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   6 weeks ago
| |
| |     Deleted intermediate files and yacc output
| |
| * commit 80e8614
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   6 weeks ago
| |
| |     Parser simplified. Multiple expression grammar rules collapsed into single rule
| |
| *   commit 7fd0c99
| |\  Merge: bb3f3bb 1dcf3f5
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   6 weeks ago
| | |
| | |     Merge remote-tracking branch 'origin/master'
| | |
| * | commit bb3f3bb
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   6 weeks ago
| | |
| | |     Makin' a bootstrapper....
| | |
```

```
| * | commit ec7f5dc
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   6 weeks ago
| | |
| | |     example LLVM code for the Linux architecture.
| | |
* | | commit e039350
| |/  Author: Fatima <fatimakoli14@gmail.com>
|/|   Date:   6 weeks ago
| |
| |       Completed-needs to be tested
| |
* | commit 1dcf3f5
| | Author: Fatima <fatimakoli14@gmail.com>
| | Date:   6 weeks ago
| |
| |     Pretty printer started and array nodes added
| |
* |   commit aad0d34
|\ \  Merge: f093c1a 4fe3fa3
| | | Author: Fatima <fatimakoli14@gmail.com>
| | | Date:   6 weeks ago
| | |
| | |     Merge branch 'master' of https://github.com/ThePhD/
    lepix
| | |
| * | commit 4fe3fa3
| |/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| |   Date:   6 weeks ago
| |
| |       Added empty files for semantic checker and codegen
| |
| * commit 4660520
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   6 weeks ago
| |
| |     floating point hello world and other examples as well
| |
| * commit e87ca95
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   6 weeks ago
| |
| |     example code in C for many of the hello worlds and basic
     examples
| |
| * commit 32bb043
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   6 weeks ago
| |
```

```
| |     That's some thick LLVM IR...
| |
| * commit 9b966c6
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   6 weeks ago
| |
| |     Perfect parallel_2d example
| |
| * commit 6211aa9
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   6 weeks ago
| |
| |     It works, uguu.
| |
| * commit a5dfc40
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   6 weeks ago
| |
| |     Additional hello world and the beginnings of a fleshed out parallel looping structure
| |
* | commit f093c1a
|/  Author: Fatima <fatimakoli14@gmail.com>
|   Date:   7 weeks ago
|
|       edited ast
|
* commit ad02f05
| Author: Gabrielle A Taylor <gat2118@columbia.edu>
| Date:   7 weeks ago
|
|     Simple C threading program, sums 2d array vertically
|
* commit cd6f557
| Author: Gabrielle A Taylor <gat2118@columbia.edu>
| Date:   7 weeks ago
|
|     Simple C threading program that sums 2d array
|
* commit f9e1a85
| Author: Gabrielle A Taylor <gat2118@columbia.edu>
| Date:   7 weeks ago
|
|     Simple C threading program
|
* commit 2e71b8e
| Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| Date:   7 weeks ago
|
```

```
|     Added top level file (lepix.ml) and deleted intermediate
    files from lex and yacc
|
*   commit 6d409f9
|\  Merge: 542ee64 dbf4809
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   7 weeks ago
| |
| |     Merge branch 'master' of https://github.com/ThePhD/lepix
| |
| * commit dbf4809
| | Author: fennilin <jackielin13@gmail.com>
| | Date:   7 weeks ago
| |
| |     Create 11-03-16
| |
* | commit 542ee64
|/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
|   Date:   7 weeks ago
|
|       Added missing tokens to parser
|
* commit b47e043
| Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| Date:   7 weeks ago
|
|     All rules added to parser. No S/R or R/R conflicts. Need
    to defined entry point for compiler (i.e. 'main')
|
*   commit 980e2a3
|\  Merge: 0ecdb5f f6283ac
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   8 weeks ago
| |
| |     Merge branch 'master' of https://github.com/ThePhD/lepix
| |
| |     Edited parser.mly
| |
| * commit f6283ac
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     preprocessor is still eluding me with a parse error.
    Need to get more info about this.
| |
| * commit 847c13c
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
```

```
| |     smaller array size, return value at end of main function
| |
| * commit 4592035
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     fix my dumb math
| |
* | commit 0ecdb5f
|/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
|   Date:   8 weeks ago
|
|       Implemented parser for all expression types. No shift
    reduce errors
|
* commit b38b568
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   8 weeks ago
|
|     ignore intermediate files
|
* commit 458c07e
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   8 weeks ago
|
|     parallel example and preprocessor code
|
*   commit a1129c4
|\  Merge: b46b06f 2b3d9d7
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     Merge branch 'master' into feature/preprocessor
| |
| |     # Conflicts:
| |     # .gitignore
| |
| * commit 2b3d9d7
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     update toplevel display file
| |
| * commit aba7a94
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     updated specification source files
| |
```

```
| * commit 35483aa
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     Toplevel PDFs we can link to.
| |
| * commit 42168dd
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     final specificaiton before submission
| |
| * commit e29b45e
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     update specification commit and ignore files
| |
| * commit 8b811a3
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     update specification
| |
| * commit cb1e9f1
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   9 weeks ago
| |
| |     Added appropriate rules for integer and float literals
    in expr grammar in parser.mly
| |
| *   commit f1c2181
| |\  Merge: a53a16f 8bee3d7
| | | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | | Date:   9 weeks ago
| | |
| | |     Merge branch 'master' of https://github.com/ThePhD/
    lepix
| | |
| | |     Merging
| | |
| * | commit a53a16f
| | | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | | Date:   9 weeks ago
| | |
| | |     Added type, loops, conditionals and array access
    grammars to parser
| | |
* | | commit b46b06f
```

```
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   8 weeks ago
| | |
| | |     update specification
| | |
* | | commit 5740bf0
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   9 weeks ago
| | |
| | |     Not quite there yet. Need to ask about it.
| | |
* | | commit 8e666f1
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   9 weeks ago
| | |
| | |     the skeleton of the preprocessor for all of this stuff
| | |
* | | commit 3cb13fa
| |/  Author: ThePhD <phdofthehouse@gmail.com>
|/|   Date:   9 weeks ago
| |
| |       begin preparing the bootstrap.py
| |
* | commit 8bee3d7 (origin/specification, specification)
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   9 weeks ago
| |
| |     specification updates
| |
* | commit c3d3fb9
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   9 weeks ago
| |
| |     re-add specification to align git submodules without
    breaking anything
| |
* | commit 25a2e7d
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   9 weeks ago
| |
| |     remove specification source since it was bugged
| |
* | commit 96a20b7
|/  Author: ThePhD <phdofthehouse@gmail.com>
|   Date:   9 weeks ago
|
|       update git modules
|
*   commit daec995
```

```
|\   Merge: 3d945dd d7811ca
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   9 weeks ago
| |
| |     Merge branch 'master' of https://github.com/ThePhD/lepix
| |
| * commit d7811ca
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   9 weeks ago
| |
| |     remove old specification files
| |
| * commit 604101b
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   9 weeks ago
| |
| |     make correct overleaf bridge in right place
| |
| *   commit 5418373
| |\   Merge: d539d38 bcea4f1
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   9 weeks ago
| | |
| | |     Merge branch 'master' of github.com:ThePhD/lepix
| | |
| | |     Fix deletion of everything
| | |
| * | commit d539d38
| | | Author: ThePhD <phdofthehouse@gmail.com>
| | | Date:   9 weeks ago
| | |
| | |     make overleaf bridge
| | |
* | | commit 3d945dd
| |/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
|/|   Date:   9 weeks ago
| |
| |       Added rules for single line comments as well as for nesting multi-line comments
| |
* | commit bcea4f1
| | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
| | Date:   9 weeks ago
| |
| |     Added regex for floating pointer literals to scanner.mll
| |
* | commit 36e9ae5
|/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
|   Date:   9 weeks ago
```

```
|
|       Added augmented version of the MicroC scanner
|
* commit f9d771a
| Author: Fatima <fatimakoli14@gmail.com>
| Date:   2 months ago
|
|     Basic tokens added to Parser
|
* commit 8174ed7
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     Remove SPIRV-LLVM setup.
|
* commit bed07b3
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     As it stands... we will not be doing SPIRV stuff. Since
    the focus will JUST be on multicore, which can be done fine
    on the CPU itself.
|
* commit 805e0c5
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     This commit allows for grammar basics.
|
|     Need to figure out how to wrap threads in LLVM IR code.
|
* commit de5101d
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     SPIRV-LLVM node
|
* commit 83d693e
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     Submodule LLVM <-> SPIRV
|
* commit 2a032f1
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     remove old lepix file name
|
```

```
* commit 31a94b4
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     Spoopy language specification
|
* commit 547718b
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
|     Skeleton files, to get ready to work.
|
* commit 2497bbd
| Author: ThePhD <phdofthehouse@gmail.com>
| Date:   3 months ago
|
| Purged.
```

ThePhD commented on Oct 3 Owner

VirtualBox, for VM needs:
Get VirtualBox here
Download and install, then install Ubuntu LTS 16 from here

You can create a Virtual Machine using that 16.04 LTS image. Just walk through the Virtual Box creation wizard: you should be able to get going immediately. If you're missing the ability to make a 64-bit Ubuntu Guest/VM, then on Windows you may need to specifically disable Hyper-V.

ThePhD self-assigned this on Oct 6

ThePhD added this to the **v0.1.0 - Specification** milestone on Oct 8

ThePhD added the tools label on Oct 8

ThePhD commented on Oct 8 • edited Owner

Commands to execute on a fresh Virtual Box Ubuntu LTS 16 VM to have everything you need:

```
sudo apt install python3
sudo apt install git build-essential m4 autotools-dev autoconf pkg-config
sudo apt install ocaml menhir opam llvm llvm-dev llvm.3.8
sudo npm install -g ocamlBetterErrors
opam install llvm
opam install llvm.3.8
opam llvm depext llvm
opam llvm depext llvm.3.8
eval $(opam config env)
```

Some of the above OPAM commands will fail

This comment will be updated as new tools and things are deemed necessary

ThePhD commented on Oct 10 Owner

LaTeX Distrubition, for building the LaTeX sources: MikTeX
Cross-platform LaTeX editor, for working with LaTeX sources: TeX Studio

TeX Studio offers the ability to build and view changes in real-time and should work

Figure 4.3: Tools and development environment setup (`https://github.com/ThePhD/lepix/issues/7`).

Chapter 5

Design

5.1 Interface

The overall interface works by simply inferring more and more information from the previous step, in a manner like so:

> Input (String) ⇒ Preprocessor [Separate Lexer, Parser] (String) ⇒ Lexer (Token Stream) ⇒ Parser (Abstract Syntax Tree) ⇒ Semantic Analyzer (Program Attributes, Semantic Syntax Tree) ⇒ Code Generation (LLVM IR Module)

Each step feeds a slimmed-down step to the next parser. The diagram for the workflow can be seen in Figure 5.1.

5.1.1 Top Level Work-flow

The way it works is simple at the highest level. Each stage produces one piece of work and hands it off to the next. To support error-reporting, a context argument is also provided to certain stages, geared to hold tracking information for that stage.

Each component flows from the next, with Preprocessing being an optional step that took in an input file and produced a source string. Because of the

```
    +------------------------------+     +---------------------------+     +---------------------------+
    |Lepix Compiler                |  +--->Abstract Syntax Tree (Raw) |  +--->  LLVM IR Code Generation  |
    +------------------------------+  |  +---------------------------+  |  +---------------------------+
+--->lepix.ml                      |  |  |scanner.mll+--->parser.mly |  |  |         codegen.ml        |
|   | *error handlers              |  |  +-----------+ | +-----------+  |  +-----------+ + +-----------+
|   | *option handling             |  |  |             v             |  |  |             v             |
|   | *glue between stages         |  |  |driver.ml: lex, parse calls|  |  |driver.ml: generate call   |
|   |                              |  |  |                           |  |  |                           |
|   +------------------------------+  |  +---------------------------+  |  +---------------------------+
+--->predriver.ml                  |  |  +---------------------------+  |
|   | *text preprocessor           +--^  |   Semantic Syntax Tree    |  |
|   |                              |     +---------------------------+  |
|   +------------------------------+     | semast.ml +---> semant.ml |  |
+--->driver.ml                     |     +-----------+ | +-----------+  |
|   | *lex, parse, check, generate |     |             v             +--^
|   | *source+line insertion       |     |driver.ml: check call      |
|   |                              |     |                           |
|   +------------------------------+     +---------------------------+
|
|   +--------------------------------------------------------------------------------------------------+
^---+Utility Bits                                                                                      |
    +--------------------------------------------------------------------------------------------------+
    |  * Error Messages: errors.ml (compiler exceptions and exit codes), messages.ml (formatting)       |
    |  * Option Helpers: options.ml                                                                     |
    |  * Missing Routines / OCaml Fill: polyfill.ml, base.ml                                            |
    |  * Basic IO routines: io.ml                                                                       |
    |                                                                                                   |
    +--------------------------------------------------------------------------------------------------+
```

Figure 5.1: Compiler and implementation organization.

way we handled input to the lexer and parser, defining an input channel for either all the text or using an input stream such as stdin was simple to handle.

5.1.2 Error Handling

True error handling with notices and carat diagnostics were only implemented for the first 3 stages of the compiler: preprocessing, lexing and parsing. Every thing else only has basic exception handlers and no context object to propagate source information or provide carat diagnostics. Thankfully, the test programs were small enough that it was easy to know what was producing errors. The downside is that this means the compiler is not very friendly to users beyond the initial parsing stages, and errors can be even more cryptic than OCaml's.

My primary motivation for good error handling came from OCaml's lacking error messages. Dozens upon dozens of "syntax error" messages that did not even seem to point to the right line, where let statements would chain well with inner expressions and only error at the end of the program, even though the error that threw off the parser in the first place was much further up in the program. Using and definitions helped in that regard, but there was still a lot of lost implementation time.

Unfortunately, our error handling again does not do a good job for the semantic errors, which – once you get used to OCaml's error messages – are actually quite good. This would take a lot more time to do appropriately, so it is unfortunate that I did not get to do more of it. I really liked implementing carat diagnostics and good error messages with line and character information, and I think it helped me fix the parser and lexer much faster and iterate over it better.

5.2 Division of Labor

I wrote essentially the entire implementation, with little kept from older commits. At one point, Fatima Koly and Akshaan Kakar's for the parser and lexer remained.

Chapter 6

Testing and Continuous Integration

6.1 Test Code

Some of the more interesting test cases include one to include a preprocessing directive (a temporary replacement for a decent module system), bottom-up type derivation for return values from functions, and a demonstration of overloading. The test cases are very involved and often nest elements to reveal bugs or other inconsistencies in the code generator (for example, properly implementing Llvm.build_load only in conditions where the type being asked for is a form of pointer). Most of these tests also had a failure case on the other side of it as well, especially in the case of overloading and bad literals. There are still bugs with expressions not quite being checked when assigned back to the original for proper convertibility, but I managed to cover a small but good area of code for working on this by myself.

```
import lib

#import "imported.lepix"

fun main () : int {
    var f : int = x.d();
    lib.print_n(f);
    return 0;
}
```

Listing 6.1: preprocess.lepix

```
fun two () {
    return 2;
}

fun main () {
    return two();
}
```

Listing 6.2: auto.lepix

```
import lib

namespace n.s {

    var global : int = 8;

}

namespace n {
    namespace s {
        var stuff : float = 3.5;
    }
}

fun s () : int {
    return 2;
}

fun s (x : int) : int {
    return x + 2;
}

fun main () : int {
    var local : int = n.s.global;
    var svalue : int = s();
    lib.print_n(local);
    lib.print_n(svalue);
    lib.print_n(s(2));
    lib.print_n(n.s.global);
    lib.print_n(n.s.stuff);
```

```
    return svalue;
}
```

Listing 6.3: overloads.lepix

6.2 Test Automation

6.2.1 Test Suite

Our test suite is a Python 3 Unit Test[1] suite, using the subprocess module to write code that called the lepix compiler, lepixc. In the case of return code 0 (success), it would then call LLVM's IR interpeter lli with the −c flag to run the program.

6.2.2 Online Automation

This was a decent bit of automation, but to further enhance our capability to know what was broken and what was fixed, I implemented Travis Continuous-Integration (travis-ci) [2] support through a travis.yml file in the top level our repository. Travis-ci is free for any publicly available, open-source github repository (the code is MIT Licensed).

6.2.3 Online Automation Tools

Docker came in handy when travis-ci had not updated their own pool of images for a very long time. We configured travis-ci to run all our commands in a small docker container using the latest ubuntu, ensuring that we had the proper OPAM, OCaml, and other development tools we needed. This was extremely helpful, and if anyone has problems in the future docker is a good way to get around old and un-updated environments. It took quite a few commits to get it working (see the testing/travis-ci branch and the plenty of frustrated commits trying to work with docker, bash and everything else to behave properly), but when it worked it was quite helpful for catching

[1] unittest is built into the python standard library, ensuring less installation steps to get going: `https://docs.python.org/3/library/unittest.html`.

[2] Our builds are here: `https://travis-ci.org/ThePhD/lepix/builds`.

any bad changes and keeping a log of things that went wrong so it could be looked at later to fix problems.

```
dist: trusty
sudo: required

language: cpp

services:
- docker

before_install:
- docker pull ubuntu:latest
- docker run -v${PWD}:/ci_repo -d --name lepix_ci ubuntu:latest sleep infinity
- docker exec lepix_ci bash -e -v -c "apt-get update"
- docker exec lepix_ci bash -e -v -c "apt install -y git python3 build-essential m4 autotools-dev autoconf pkg-config ocaml menhir opam llvm llvm-dev llvm.3.8"
- docker exec lepix_ci bash -e -v -c "opam init -y"
- docker exec lepix_ci bash -e -v -c "opam install -y core depext llvm.3.8"

script:
- docker exec lepix_ci bash -e -v -c "source ci_repo/ci/travis.sh"

after_script:
- docker stop lepix_ci
- docker rm lepix_ci

notifications:
email:
on_success: change
on_failure: change
```

Listing 6.4: .travis.yml

6.3 Division of Labor

I wrote a large number of examples and also wrote tests, implemented travis-ci integration, and wrote the python bootstrapper and test suite code.

Chapter 7

Post-Mortem and Lessons Learned

This is going to be the most in-depth section because it is here where I can explain primarily why I think the group did not meet its target and why I felt like splitting off would be more worth it than staying with the team. While I individually put in a lot of effort and achieved some very good technical goals, the divide with my group near the end was still a problem and resulted in a lot of codegen for constructs successfully put into the parser and AST to not be implemented.

7.1 Talk to your Teammates, Early

When I experienced problems with my teammates not hitting deadlines, I at first was confused. I did not know why they were not delivering the portions of code they said they would deliver on the deadlines they imposed on themselves, and at certain points when they did deliver I had to constantly revise what they had done. Here are some examples of how I did not optimally handle bad situations:

For one, the Parser and Lexer for this LéPiX implementation look nothing like the one committed and declared to our advisor as ”complete”. It did not parse our language and there were obvious holes in its syntax: for loops variable initialization did not work, initializer lists for control flow did not work, parallel block initializers were not considered, the parallel for syntax we changed for a parallel block were not changed, namespaces were not recognized and qualified identifiers did not exist.

Rather than tell my teammates what was wrong and what needed to be fix and divide the work, I instead implemented all of the things mentioned above, committed them, and then moved on. I felt that if my teammates would not run the code against the example LéPiX code we had to see if it works properly, that they were not doing the bare minimum to even know if what they wrote was correct or good. I had to learn everything, put it all together under pressure, and then fix it in time for the next Milestone.

7.2 Manage Expectations, Know What You Want

One of the next major issues is that team members had differing expectations about the quality of work. In particular, I was expecting a very thorough, consistent applied effort from my team and not things done a few weeks after the Professor, TA, and others had urged us needed to be done long before we had begun to look at it.

On the good side, the Language Reference Manual was done on-time with participation from everyone. It was the one part of the project where – even if we were working up to the deadline – everyone participated, took a section, made their work clear and actually did their work during the times they said they would.

Unfortunately, this flopped for actual implementation. One of our group members held onto the Semantic AST for nearly five weeks of time, refusing to commit code when asked and spinning down the time of myself and other group mates eager to get started on Code Generation. The Lexer and Parser were not up to parsing our language. Many disconnects appeared in how the implementation was done, which was entirely strange because we had specifically said we would wait for the Language Reference Manual to be done to begin working so everyone would have a very clear goal and standard.

Talking to your teammates about what exactly is expected, even with a document like the Language Reference Manual, would be helpful in the future. You and your teammates should be able to look at previous projects, and see

1. To achieve X feature it took Y lines of code.
2. Is that feasible if you give yourself Z amount of time to write Y lines with W people?
3. What quality of implementation do you want? Proof of concept? Fully vetted with compiler errors?

As an example, I wanted full source code information and carat diagnostics throughout the program. I only managed to add that to the first half of the project, and in my lack of help and time for the second half did not implement it for Semantic AST and Codegen errors.

Other groups would consider this silly and not bother with it at all. Your team should agree on just how much effort and polish your implementation deserves, and have a frank discussion about whether people will do that work.

If people impose deadlines on themselves and do not mean them, talk to them immediately about it rather than just implementing it yourself in frustration. Only when they do not respond to your inquiries do you turn to outside sources and begin to re-evaluate what can and cannot be done with your time.

7.3 Start Confrontations

When people in my group slipped deadlines, I vented my frustrations elsewhere while implementing the code just in time for deadlines or pulling together LaTeX documents and editing them furiously. I confronted my team only once very early in September and CCed the professor and a TA with an e-mail, where I demanded they never put me in a situation similar to the one where I wrote the entire LéPiX proposal by myself and then have them – only an hour or so before the deadline – tell me grammatical edits that I needed to fix.

After that, I did not expect to have to send them anymore particularly strongly-worded e-mails. They had agreed not to do something like that again and indeed everyone participated in the Language Reference Manual. We had communication over GroupMe about why the AST and Semantic AST were not being done on time, but I had not made it clear that their lack of implementation was unacceptable: I only patched it over in the days before the deadline after I had grown tired of waiting and needed to have implementation work done to do my part.

You must have confrontations. You must butt heads. Do this early, and do it often when a group member does not hand in their work. Growing frustrated in silence while implementing things you would have expected your teammates to do will only wear you out and ultimately lead you to a place where you will want to discard anything your team does, good or bad, and not take their suggestions in because you feel like they will just let you down.

Chapter 8

Appendix

8.1 Source Code Listing

```
.PHONY: default
default: all;

# Clean intermediate files
clean :
ocamlbuild -use-menhir -build-dir obj -clean
rm -rf lepixc lepix
rm -rf scanner.ml parser.ml parser.mli
rm -rf prescanner.ml preparser.ml preparser.mli
rm -rf *.cmx *.cmi *.cmo *.cmx *.o
rm -rf parser.automaton preparser.automaton
rm -rf parser.output preparser.output parser.conflicts preparser
    .conflicts

# Build top level lepix executable
lepix :
ocamlbuild -use-ocamlfind -use-menhir -tag thread -pkgs core,
    llvm,llvm.analysis -build-dir obj lepix.native
cp -f obj/lepix.native lepixc

install :
cp lepixc /usr/local/bin/lepixc

uninstall :
rm -f /usr/local/bin/lepixc

.PHONY: all
```

```
all : lepix
```

Listing 8.1: source/Makefile

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* In Javascript, there's a concept called 'Polyfill'. It's the concept that
stuff that's missing can be filled over by libraries implemented by regular people
because the committee that oversees Javascript can't just decide to
make certain implementations and other things standard.

This is that thing, for OCrapml. *)

(* Algorithm *)
let foldi f value start_index len =
```

```
  let end_index = start_index + len - 1 in
  if start_index >= end_index then value else
  let accumulated = ref value
  in
  for i = start_index to end_index do
    accumulated := ( f !accumulated i )
  done;
  !accumulated

let foldi_to f value start_index end_index =
  foldi f value start_index (end_index - start_index)

(* Integer *)
let rec powi n = function
  | 0 -> 1
  | 1 -> n
  | x -> n * ( powi n  x - 1 )

let int_of_bool b = if b then 1 else 0

let int_of_string_base b s =
  let len = (String.length s) in
  let acc num i = let c = s.[i] in
    let v = if c >= '0' || c <= '9' then
        int_of_char c - int_of_char '0'
      else
        if c >= 'A' || c <= 'Z' then
          int_of_char c - int_of_char 'A' + 10
        else
          if c >= 'a' || c <= 'z' then
            int_of_char c - int_of_char '0' + 10
          else 0
    and place = len - 1 - i
    in
    num + ( v * ( powi b place ) )
  in
  foldi acc 0 0 len

(* Num *)

exception BadBase of string
exception DigitGreaterThanBase of string

let num_of_string_base_part b s =
  if b > 36 || b < 1 then raise(BadBase "num_of_string_base
```

```
    : base cannot be greater than 36 or less than 1") else
  let n0 = Num.num_of_int 0 in
  let len = (String.length s) in
  if len < 1 then n0 else
  let (mid, starter) = try ( ( String.index s '.' ), 1 )
    with _ -> (len - 1, 0)
  and nb = Num.num_of_int b
  in
  let acc (n, skipval) i = let c = s.[i] in
    if c = '.' then (n, skipval - 1) else
    let v = if c >= '0' && c <= '9' then
        ( int_of_char c ) - ( int_of_char '0' )
      else
        if c >= 'A' && c <= 'Z' then
          ( int_of_char c ) - ( int_of_char 'A' ) + 10
        else
          if c >= 'a' && c <= 'z' then
            int_of_char c - ( int_of_char 'a' ) + 10
          else 0
    and place = mid - i - skipval
    in
    if v > b then raise(DigitGreaterThanBase ("
    num_of_string_base: digit '" ^ (String.make 1 c) ^ "' ("
     ^ (string_of_int v) ^ ") is higher than what base '" ^
    (string_of_int b) ^ "' can handle")) else
    let nv = Num.num_of_int v
    and nplace = Num.num_of_int place
    in
    (Num.add_num n ( Num.mult_num nv ( Num.power_num nb
    nplace ) ), skipval)
  in
  let (n, _) = foldi acc (n0, starter) 0 len in
  n

let num_of_string_base b s =
  num_of_string_base_part b s

let num_of_string s =
  let slen = String.length s in
  try
    let eidx = String.index s 'e' in
    if eidx < 1 then raise(Not_found);
    let eidxp1 = ( eidx + 1 ) in
    let nval = num_of_string_base_part 10 (String.sub s 0
    eidx)
```

```
    and eval = if eidxp1 < slen then
   num_of_string_base_part 10 (String.sub s eidxp1 (slen -
   eidxp1)) else ( Num.num_of_int 0 )
    in
    Num.mult_num nval ( Num.power_num (Num.num_of_int 10)
   eval )
  with
    | Not_found -> num_of_string_base_part 10 s

(* Char *)
let is_whitespace = function
  | ' ' -> true
  | '\t' -> true
  | '\n' -> true
  | '\r' -> true
  | _ -> false

(* String *)
let string_to_list s =
  let l = ref [] in
  let acc c =
    l := c :: !l; ()
  in
  String.iter acc s;
  List.rev !l

let iteri f start_index len =
  let end_index = start_index + len - 1 in
  if start_index < end_index then
    for i = start_index to end_index do
      ( f i )
    done

type split_option =
  | RemoveDelimeter
  | KeepDelimeter

let string_split_with v s opt =
  let e = String.length s
  and vlen = String.length v
  in
  if vlen >= e then [s] else
  let forward_search start =
    let acc found idx =
      found && ( s.[start + idx] = v.[idx] )
```

```
    in
    foldi acc true 1 (vlen - 1)
  in
  let add_sub len slist start =
    if len < 1 then ( start , slist ) else
    let fresh = ( String.sub s start len )
    and last = start + len + vlen in
    begin match opt with
      | RemoveDelimeter -> ( last , fresh :: slist )
      | KeepDelimeter ->
        let slist = v :: slist in
        ( last , fresh :: slist )
    end
  in
  let acc (last , slist) start =
    if (start < last) then (last , slist) else
    if ( s.[start] = v.[0] ) then
      if (forward_search start) then
        let len = start - last in
        (add_sub len slist last )
      else
        (last , slist)
    else
      if ( start = ( e - 1 ) ) then
        let len = e - last in
        (add_sub len slist last )
      else
        (last , slist)
  in
  let (_, slist) = ( foldi acc (0, []) 0 e ) in
  (* Return complete split list *)
  List.rev slist

let string_starts_with str pre =
  let prelen = (String.length pre) in
  prelen <= (String.length str ) && pre = (String.sub str 0 prelen)

let string_split v s =
  string_split_with v s RemoveDelimeter
```

../source/polyfill.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person
```

```
    obtaining a copy of this
software and associated documentation files (the "Software"
    ), to deal in the Software
without restriction, including without limitation the
    rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
permit persons to whom the Software is furnished to do so,
    subject to the following
conditions:

The above copyright notice and this permission notice shall
    be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Base types and routines. *)

type token_source = {
      token_source_name : string;
  token_number : int;
      token_line_number : int;
      token_line_start : int;
  token_column_range : int * int;
  token_character_range : int * int;
}

type target =
  | Pipe
  | File of string

let target_to_string = function
  | Pipe -> "pipe"
  | File(s) -> "file: " ^ s
```

```
let target_to_pipe_string i b = match i with
  | Pipe -> if b then "stdin" else "stdout"
  | File(s) -> "file: " ^ s

type action =
  | Help
  | Preprocess
  | Tokens
  | Ast
  | Semantic
  | Llvm
  | Compile

let action_to_int = function
  | Help -> -1
  | Preprocess -> 0
  | Tokens -> 1
  | Ast -> 10
  | Semantic -> 100
  | Llvm -> 1000
  | Compile -> 10000

let entry_point_name = "main"

(* Core options *)

let default_integral_bit_width = 32
let default_floating_bit_width = 64

(* Error message helpers *)

let line_of_source src token_info =
  let ( absb, abse ) = token_info.token_character_range
  and linestart = token_info.token_line_start
  in
  let ( lineend, _ ) =
    let f (endindex, should_skip) idx =
      let c = src.[idx] in
      let skip_this = c = '\n' in
      if should_skip || skip_this then
        (endindex, true)
      else
        (endindex + 1, false)
    in
```

```
    Polyfill.foldi f ( linestart, false ) linestart ( ( String.length src ) - linestart )
  in
  let srcline = String.sub src linestart (max 0 (lineend - linestart - 1)) in
  let srclinelen = String.length srcline in
  let ( srcindent, _ ) =
    let f (s, should_skip) idx =
      let c = srcline.[idx] in
      let nws = not ( Polyfill.is_whitespace c ) in
      if should_skip || nws then
        (s, false)
      else
        (s ^ ( String.make 1 c ), true)
    in
    Polyfill.foldi f ( "", false ) 0 srclinelen
  in
  let indentlen = String.length srcindent
  and tokenlen = lineend - absb
  in
  ( srcline, srcindent, (max ( srclinelen - indentlen - tokenlen ) 0 ) )


let brace_tabulate str tabs =
  let len = ( String.length str ) in
  let lines = Polyfill.string_split_with "\n" str Polyfill.KeepDelimeter in
  let lineslen = ( List.length lines ) in
  let buf = Buffer.create ( len + ( lineslen * 4 ) ) in
  let acc ( buf, t ) line =
    let tmod = 0 - ( Polyfill.int_of_bool ( String.contains line '}' ) ) in
    let t = t + tmod in
    Buffer.add_string buf (String.make t '\t'); Buffer.add_string buf line;
    let t = t + ( Polyfill.int_of_bool ( String.contains line '{' ) ) in
    (buf, t)
  in
  let (buf, _) = List.fold_left acc ( buf, tabs ) lines in
  Buffer.contents buf
```

../source/base.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Options / argument parser *)

type option =
  | Dash of string
  | DoubleDash of string
  | Argument of int * string

type options_context = {
     mutable options_help : string -> string;
}

let read_options ocontext sys_argv =
  let argc = Array.length sys_argv - 1 in
  (* Skip first argument one (argv 0 is the path
  of the exec on pretty much all systems) *)
  let argv = ( Array.sub sys_argv 1 argc )
```

```
and action = ref Base.Help
and verbose = ref false
and input = ref Base.Pipe
and output = ref Base.Pipe
and specified = ref []
and seen_stdin = ref false
in
(* Our various options *)
let update_action a =
  specified := a :: !specified;
  if ( Base.action_to_int !action ) < ( Base.action_to_int a ) then
    action := a;
in
let options = [
  ( 1, "h", "help", "print the help message",
    fun _ _ -> ( update_action(Base.Help) )
  );
  ( 1, "p", "preprocess", "Preprocess and display source",
    fun _ _ -> ( update_action(Base.Preprocess) )
  );
  ( 1, "i", "input", "Take input from standard in (default: stdin)",
    fun _ _ -> ( input := Base.Pipe; seen_stdin := true )
  );
  ( 2, "o", "output", "Set the output file (default: stdout)",
    fun _ o -> ( output := Base.File(o) )
  );
  ( 1, "t", "tokens", "Print the stream of tokens",
    fun _ _ -> ( update_action(Base.Tokens) )
  );
  ( 1, "a", "ast", "Print the parsed Program",
    fun _ _ -> ( update_action(Base.Ast) )
  );
  ( 1, "s", "semantic", "Print the Semantic Program",
    fun _ _ -> ( update_action(Base.Semantic) )
  );
  ( 1, "l", "llvm", "Print the generated LLVM code",
    fun _ _ -> ( update_action(Base.Llvm) )
  );
  ( 1, "c", "compile", "Compile the desired input and output the final LLVM",
    fun _ _ -> ( update_action(Base.Compile) )
```

```
    );
    ( 1, "v", "verbose", "Be as explicit as possible with all steps",
      fun _ _ -> ( verbose := true )
    );
  ]
  and position_option arg_index positional_index arg =
    if Sys.file_exists arg then
      input := Base.File( arg )
    else
      raise(Errors.OptionFileNotFound(arg))
  in
  let help tabulation =
    let value_text = "<value>" in
    let value_text_len = String.length value_text in
    let longest_option =
      let acc len o = match o with
        | (sz, _, long, _, _ ) ->
          let newlen = (String.length long)
            + if sz = 2 then 1 + value_text_len else 0
          in
          if newlen > len then newlen else len
      in
      let l = 1 + ( List.fold_left acc 1 options ) in
      if l < value_text_len then value_text_len else l
    in
    let concat_options t =
      let builder s o = match o with
        | ( sz, short, long, desc, _ ) ->
          let long_len = String.length long in
          let spacing_size = longest_option - long_len - ( if sz = 2 then 1 + value_text_len else 0 ) in
          let spacing_string = (String.make spacing_size ' ') in
          s
          ^ "\n" ^ t ^ "-" ^ short
          ^ "\t--" ^ long ^ (if sz = 1 then "" else " " ^ value_text)
          ^ spacing_string
          ^ desc
      in
      (List.fold_left builder "" options)
    in
    let (_, input_short, input_long, _, _) = List.nth options 2 in
```

```
    let msg = "Help:"
      ^ "\n" ^ tabulation ^ "lepix [options] filename [filenames...]"
      ^ "\n" ^ tabulation ^ "\t" ^ "filename | filenames can have one option -" ^ input_short ^ " or --" ^ input_long
      ^ "\n" ^ tabulation ^ "options:"
      ^ ( concat_options (tabulation ^ "\t") )
    in
    msg
  in
  ocontext.options_help <- help;
  (* Exit early if possible *)
  if argc < 1 then
    (!input, !output, !action, !specified, !verbose)
  else

  let to_option idx arg =
    let arglen = String.length arg in
    match arg with
      | _ when Polyfill.string_starts_with arg "--" -> DoubleDash((String.sub arg 2 (arglen - 2)))
      | _ when Polyfill.string_starts_with arg "-" -> Dash((String.sub arg 1 (arglen - 1)))
      | _ -> Argument(idx, arg)
  in
  (* Convert all arguments to the Option type first *)
  let options_argv = Array.mapi to_option argv in

  (* Function for each argument *)
  let f (index, positional_index, skip_next) option_arg =
    if skip_next then (1 + index, positional_index, false) else
    let execute_on_match_sub_option ( opt_failure, should_block ) opt_string pred = match opt_failure with
      (* There is some failure, so just propogate it through *)
      | Some(x) -> ( opt_failure, should_block )
      (* There is no failure, so now work with the list *)
      | None -> begin match List.filter pred options with
        (* We use filter instead of find because find is dumb and throws an
        exception instead of just returning an optional because
        whoever designed the OCaml standard library is an
```

```
absolute
    bell end. *)
    | ( 1, _, _, _, f ) :: tail -> (* Only needs 1
argument *)
      (f opt_string "");
      ( opt_failure, should_block )
    | ( 2, _, _, _, f ) :: tail -> (* Needs 2 arguments
, look ahead by 1 *)
      if ( index + 1 ) >= argc then
        raise(Errors.MissingOption(opt_string));
      let nextarg = ( options_argv.(1 + index) ) in
      let _ = match nextarg with
        | Argument(idx, s) -> (f opt_string s)
        | _ -> raise(Errors.BadOption(opt_string))
      in
      ( opt_failure, true )
    | _ -> (* Unhandled case: return new failure string
 *)
      ( Some opt_string, should_block )
   end
 and on_failure dashes opt arglist arg =
   let msg = dashes ^ opt
     ^ if ( List.length arglist ) > 1 then " ( in " ^
dashes ^ arg ^ " )" else ""
   in
   raise(Errors.BadOption(msg))
 in
 let ( should_skip_next, was_positional ) = match
option_arg with
   | Dash(arg) ->
     (* if it has a dash only *)
     (* each letter can be its own thing *)
     let perletter (opt_failure, should_break) c =
       let opt_string = ( String.make 1 c ) in
       let short_pred (_, short, _, _, _ ) =
         short = opt_string
       in
       execute_on_match_sub_option (opt_failure,
should_break) opt_string short_pred
     in
     (* look at every character. If there's 1 match
among them, go crazy *)
     let arglist = (Polyfill.string_to_list arg) in
     let (opt_failure, causes_skip) = ( List.fold_left
perletter ( None, false ) arglist ) in
```

```
        begin match opt_failure with
          | None ->
            ( causes_skip , 0 )
          | Some(opt) -> let _ = (on_failure "--" opt
    arglist arg) in
            ( causes_skip , 0 )
        end
      | DoubleDash(arg) ->
        (* if it has a double dash... *)
        (* each comma-delimeted word can be its own option
    *)
        let perword (opt_failure, problems) opt_string =
          let long_pred (_, _, long, _, _ ) =
            long = opt_string
          in
          execute_on_match_sub_option (opt_failure,
    problems) opt_string long_pred
        in
        (* look at word character. If there's 1 match among
     them, go crazy *)
        let arglist = Polyfill.string_split "," arg in
        let (opt_failure, causes_skip) = ( List.fold_left
    perword ( None, false ) arglist ) in
        begin match opt_failure with
          | None ->
            ( causes_skip , 0 )
          | Some(opt) -> let _ = (on_failure "--" opt
    arglist arg) in
            ( causes_skip , 0 )
        end
      (* otherwise, it's just a positional argument *)
      | Argument(idx, arg) ->
        (position_option index positional_index arg);
        ( skip_next , 1 )
    in
    (1 + index, positional_index + was_positional,
    should_skip_next)
  in
  (* Iterate over the arguments *)
  let _ = Array.fold_left f (0, 0, false) options_argv in
  (* Return tuple of input, output, action *)
  ( !input, !output, !action, !specified, !verbose )
```

../source/options.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD
```

```

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Message formatters and helpers. *)


let preprocessing_error pcontext =
  let ( t, info ) = pcontext.Predriver.token in
  let ( source_line, source_indentation, columns_after_indent ) =
    ( Base.line_of_source pcontext.Predriver.source_code info )
  in
  let column_range = info.Base.token_column_range in
  let (column_text, is_columns_wide) = Representation.token_range_to_string column_range in
  let msg = "Preprocessing Error in " ^ pcontext.Predriver.source_name ^ ":"
  ^ "\n" ^ "\t" ^ "Unrecognizable parse pattern at token #"
```

```
      ^ ( string_of_int pcontext.Predriver.token_count )
      ^ ": [id " ^ string_of_int info.Base.token_number ^ ":" ^ Representation.preparser_token_to_string t ^ "]"
    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int info.Base.token_line_number
    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: " else "Column: " ) ^ column_text
    ^ "\n"
    ^ "\n" ^ source_line
    ^ "\n" ^ source_indentation ^ ( String.make columns_after_indent ' ' ) ^ "^~~"
  in
  msg

let preprocessing_lexer_error pcontext core_msg c s e =
  let ( t, info ) = pcontext.Predriver.token in
  let ( source_line, source_indentation, columns_after_indent ) =
    ( Base.line_of_source pcontext.Predriver.source_code info )
  in
  let column_range = info.Base.token_column_range in
  let (column_text, is_columns_wide) = Representation.token_range_to_string column_range in
  let msg = "Preprocessing Lexing Error in " ^ pcontext.Predriver.source_name ^ ":"
    ^ "\n" ^ "\t" ^ core_msg ^ " at token#" ^ ( string_of_int pcontext.Predriver.token_count )
      ^ ": [id " ^ string_of_int info.Base.token_number ^ ":" ^ Representation.preparser_token_to_string t ^ "]"
    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int info.Base.token_line_number
    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: " else "Column: " ) ^ column_text
    ^ "\n"
    ^ "\n" ^ source_line
    ^ "\n" ^ source_indentation ^ ( String.make columns_after_indent ' ' ) ^ "^~~"
  in
  msg

let lexer_error context core_msg c s e =
  let abspos = s.Lexing.pos_cnum in
  let endabspos = e.Lexing.pos_cnum in
  let relpos = 1 + abspos - s.Lexing.pos_bol in
```

```
    let endrelpos = 1 + endabspos - e.Lexing.pos_bol in
    let (column_text, is_columns_wide) = Representation.
      token_range_to_string ( relpos, endrelpos ) in
    let msg = "Lexing Error in " ^ context.Driver.source_name
       ^ ":"
    ^ "\n" ^ "\t" ^ core_msg ^ " at character: " ^  c
    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int s.Lexing.
      pos_lnum
    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: "
      else "Column: " ) ^ column_text
    in
    msg

let parser_error context core_msg =
    let ( t, info ) = context.Driver.token in
    let ( source_line, source_indentation,
      columns_after_indent ) =
       ( Base.line_of_source context.Driver.source_code info )
    in
    let column_range = info.Base.token_column_range in
    let (column_text, is_columns_wide) = Representation.
      token_range_to_string column_range in
    let msg = "Parsing Error in " ^ context.Driver.
      source_name ^ ":"
    ^ "\n" ^ "\t" ^ core_msg ^ " at token #" ^ (
      string_of_int context.Driver.token_count )
       ^ ": [id " ^ string_of_int info.Base.token_number ^ ":"
       ^ Representation.parser_token_to_string t ^ "]"
    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int info.Base.
      token_line_number
    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: "
      else "Column: " ) ^ column_text
    ^ "\n"
    ^ "\n" ^ source_line
    ^ "\n" ^ source_indentation ^ ( String.make
      columns_after_indent ' ' ) ^ "^~~"
    in
    msg
```

../source/message.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person
     obtaining a copy of this
software and associated documentation files (the "Software"
```

```
), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Top-level of the LePiX compiler: scan & parse the input,
check the resulting AST, generate LLVM IR, and dump the module *)

let _ =
  let input = ref Base.Pipe in
  let output = ref Base.Pipe in
  let action = ref Base.Compile in
  let verbose = ref false in
  let specified = ref [] in
  let context = {
    Driver.source_name = "";
    Driver.source_code = "";
    Driver.original_source_code = "";
    Driver.token_count = 0;
    Driver.token = ( Parser.EOF,
      { Base.token_source_name = ""; Base.token_number = 0;
      Base.token_line_number = 0; Base.token_line_start = 0;
      Base.token_column_range = (0, 0); Base.
```

```
  token_character_range = (0, 0) }
  );
}
and pcontext = {
      Predriver.source_name = "";
    Predriver.source_code = "";
    Predriver.original_source_code = "";
    Predriver.token_count = 0;
    Predriver.token = ( Preparser.EOF,
      { Base.token_source_name = ""; Base.token_number = 0;
      Base.token_line_number = 0; Base.token_line_start =
  0;
      Base.token_column_range = (0, 0); Base.
  token_character_range = (0, 0) }
    );
     }
in
let ocontext = {
  Options.options_help = fun (s) -> ( "" );
} in
(* Call options Parser for Driver *)
let _ =
try
  let ( i, o, a, s, v ) = ( Options.read_options ocontext
  Sys.argv ) in
    input := i;
    output := o;
    action := a;
    specified := s;
    verbose := v
with
  | err -> let _ = match err with
    | Errors.BadOption(s) ->
      let msg = "Options Error:"
      ^ "\n" ^ "\t" ^ "Unrecognized option: " ^ s
      ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
      prerr_endline msg
    | Errors.NoOption ->
      let msg = "Options Error:"
      ^ "\n" ^ "\t" ^ "No inputs or options specified"
      ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
      prerr_endline msg
    | Errors.MissingOption(o) ->
      let msg = "Options Error:"
      ^ "\n" ^ "\t" ^ "Flag " ^ o ^ " needs an additional
```

```
    argument after it that is not dashed"
        ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
        prerr_endline msg
      | Errors.OptionFileNotFound(f) ->
        let msg = "Options Error:"
        ^ "\n" ^ "\t" ^ "File " ^ f ^ " was not found"
        ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
        prerr_endline msg
      | err ->
        let msg = "Unknown Error during Option parsing:"
        ^ "\n" ^ "\t" ^ "Contact the compiler vendor for
    more details and possibly include source code, or try
    simplifying the program"
        in
        prerr_endline msg;
        raise(err)
    in
    (* Exit if arguments are wrong *)
    ignore( ( exit Errors.option_error_exit_code ) )
  in
  (* Perform actual lexing and parsing using the Driver
    here *)
  try
    let allactions = !specified in
    let source_name = ( Base.target_to_pipe_string !input
    true ) in
    let output_to_target ( s ) = match !output with
      | Base.Pipe -> ( print_endline s )
      | Base.File(f) -> ( Io.write_file_text s f )
    in
    let print_predicate b =
      fun v -> ( v = b )
    in
    let print_help () =
      let msg = ( ocontext.Options.options_help "\t" ) in
      print_endline msg
    in
    if !action = Base.Help then begin
      print_help ()
    end else
    (* Since we do the actions in these functions multiple
    times,
    We refactor them out here to make our lives easier
    while we tweak
    stuff *)
```

```
    let get_source () =
      let pre_source_text = match !input with
        | Base.Pipe -> Io.read_text stdin
        | Base.File(f) -> ( Io.read_file_text f )
      in
      let source_text = Predriver.pre_process pcontext !input pre_source_text in
      context.Driver.source_name <- source_name;
      context.Driver.original_source_code <- pre_source_text;
      context.Driver.source_code <- source_text;
      source_text
    in
    let dump_tokens f tokenstream =
      if ( List.exists (print_predicate Base.Tokens) allactions ) then f( Representation.parser_token_list_to_string tokenstream )
    and dump_ast f program =
      if ( List.exists (print_predicate Base.Ast) allactions ) then f( Representation.string_of_program program )
    and dump_semantic f semanticprogram =
      if ( List.exists (print_predicate Base.Semantic) allactions ) then f( Representation.string_of_s_program semanticprogram )
    and dump_module f m =
      f( Llvm.string_of_llmodule m )
    in
    let _ = match !action with
      | Base.Help -> print_help ()
      | Base.Preprocess ->
        let source_text = get_source () in
        output_to_target( source_text )
      | Base.Tokens ->
        let source_text = get_source () in
        let lexbuf = Lexing.from_string source_text in
        let tokenstream = Driver.lex source_name lexbuf in
        ( dump_tokens output_to_target tokenstream )
      | Base.Ast ->
        let source_text = get_source () in
        let lexbuf = Lexing.from_string source_text in
        let tokenstream = Driver.lex source_name lexbuf in
        ( dump_tokens print_endline tokenstream );
        let program = Driver.parse context tokenstream in
        ( dump_ast output_to_target program)
```

```
        | Base.Semantic ->
          let source_text = get_source () in
          let lexbuf = Lexing.from_string source_text in
          let tokenstream = Driver.lex source_name lexbuf in
          ( dump_tokens print_endline tokenstream );
          let program = Driver.parse context tokenstream in
          ( dump_ast print_endline program );
          let semanticprogram = Driver.analyze program in
          ( dump_semantic output_to_target semanticprogram )
        | Base.Llvm ->
          let source_text = get_source () in
          let lexbuf = Lexing.from_string source_text in
          let tokenstream = Driver.lex source_name lexbuf in
          ( dump_tokens print_endline tokenstream );
          let program = Driver.parse context tokenstream in
          ( dump_ast print_endline program );
          let semanticprogram = Driver.analyze program in
          ( dump_semantic print_endline semanticprogram );
          let m = Codegen.generate semanticprogram in
          if !verbose then ( dump_module print_endline m );
          ( dump_module output_to_target m )
        | Base.Compile ->
          let source_text = get_source () in
          let lexbuf = Lexing.from_string source_text in
          let tokenstream = Driver.lex source_name lexbuf in
          ( dump_tokens print_endline tokenstream );
          let program = Driver.parse context tokenstream in
          ( dump_ast print_endline program );
          let semanticprogram = Driver.analyze program in
          ( dump_semantic print_endline semanticprogram );
          let m = Codegen.generate semanticprogram in
          Llvm_analysis.assert_valid_module m;
          ( dump_module output_to_target m )
    in
    ()
  with
    | err -> let _ = match err with
      (* Preprocessor-Specific Errors *)
      (* Preprocessing Parser Errors *)
      | Preparser.Error ->
        let msg = Message.preprocessing_error pcontext in
        prerr_endline msg
      | Errors.PreUnknownCharacter( c, (s, e) ) ->
        let msg = Message.preprocessing_lexer_error
   pcontext "Unrecognized character in program" c s e in
```

```
        prerr_endline msg

      (* General Compiler Errors *)
      (* Lexer Errors *)
      | Errors.UnknownCharacter( c, (s, e) ) ->
        let msg = Message.lexer_error context "Unrecognized character in program" c s e in
        prerr_endline msg

      | Errors.BadNumericLiteral( c, (s, e) ) ->
        let msg = Message.lexer_error context "Bad character in numeric literal" c s e in
        prerr_endline msg

      (* Parser Errors *)
      | Parser.Error
      | Parsing.Parse_error ->
        let msg = Message.parser_error context "Unrecognizable parse pattern" in
        prerr_endline msg
      | Errors.MissingEoF ->
        let msg = "Parsing Error in" ^ context.Driver.source_name ^ ":"
        ^ "\n" ^ "\t" ^ "Missing EoF at end of token stream (bad lexer input?)"
        in
        prerr_endline msg

      (* Semantic Analyzer and Codegen Errors *)
      (* Semantic Errors *)
      (* TODO: positional information should be tracked through the AST and SemAST,
      all the way to codegen, as well... *)
      | Errors.BadFunctionCall(s) ->
        let msg = "Bad Function Call error: " ^ s
        in
        prerr_endline msg
      | Errors.FunctionAlreadyExists(s) ->
        let msg = "Function Already Exists error: " ^ s
        in
        prerr_endline msg
      | Errors.VariableAlreadyExists(s) ->
        let msg = "Variable Already Exists error: " ^ s
        in
        prerr_endline msg
```

```
    | Errors.TypeMismatch(s) ->
      let msg = "Mismatched types error: " ^ s
      in
      prerr_endline msg
    | Errors.IdentifierNotFound(s) ->
      let msg = "Identifier Not Found error: " ^ s
      in
      prerr_endline msg
    | Errors.InvalidFunctionSignature(s, n) ->
      let msg = "Invalid signature: " ^ s ^ " in " ^ n
      in
      prerr_endline msg
    | Errors.InvalidMainSignature(s) ->
      let msg = "Invalid signature: " ^ s
      in
      prerr_endline msg
    | Errors.InvalidBinaryOperation(s)
    | Errors.InvalidUnaryOperation(s) ->
      let msg = "Invalid operation: " ^ s
      in
      prerr_endline msg

    (* Direct Codegen Errors *)
    | Errors.UnknownVariable(s) ->
      let msg = "Codegen (LLVM IR) error: " ^ s
      in
      prerr_endline msg
    | Errors.UnknownFunction(s) ->
      let msg = "Codegen (LLVM IR) error: " ^ s
      in
      prerr_endline msg
    | Errors.VariableLookupFailure(name, _) ->
      let msg = "Codegen (LLVM IR) error: could not
properly find variable with the name " ^ name
      in
      prerr_endline msg
    | Errors.FunctionLookupFailure(name, mangledname) ->
      let msg = "Codegen (LLVM IR) error: looking for the
 function with the name " ^ name ^ " (mangled name: " ^
mangledname ^ ")"
      in
      prerr_endline msg
    | Errors.BadPrintfArgument ->
      let msg = "Codegen (LLVM IR) error: lib.print and
related functions only take either a string, an integer,
```

```
 or a floating point argument"
        in
        prerr_endline msg

      (* Common Errors *)
      (* Missing File/Bad File Name, Bad System Calls *)
      | Sys_error(s) ->
        let msg = "Sys_error: \n\t" ^ s
        in
        prerr_endline msg

      (* Unsupported features *)
      | Errors.Unsupported(s) ->
        let msg = "Unsupported (ran out of implementation
 time): \n\t" ^ s
        in
        prerr_endline msg

      (* Unknown Errors *)
      | err ->
        let msg = "Unknown Error during Compilation:"
        ^ "\n" ^ "\t" ^ "Contact the compiler vendor for
 more details and possibly include source code, or try
 simplifying the program"
        in
        prerr_endline msg;
        raise(err)
    in
    ignore( exit Errors.compiler_error_exit_code )
```

../source/lepix.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person
    obtaining a copy of this
software and associated documentation files (the "Software"
  ), to deal in the Software
without restriction, including without limitation the
  rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies
  of the Software, and to
permit persons to whom the Software is furnished to do so,
  subject to the following
conditions:

```

```
The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* A listing of exceptions and the methods that power them
to make the parser more expressive *)

(* Driver and Related class of errors *)
let option_error_exit_code = 1

(* Option Errors *)
exception NoOption
exception BadOption of string
exception MissingOption of string
exception OptionFileNotFound of string

(* Compiler class of Errors *)
let compiler_error_exit_code = 2
(* Lexer Errors *)
exception PreUnknownCharacter of string * ( Lexing.position * Lexing.position )
exception UnknownCharacter of string * ( Lexing.position * Lexing.position )
exception BadNumericLiteral of string * ( Lexing.position * Lexing.position )

(* Parser Errors *)
exception MissingEoF
exception BadToken

(* Semantic and Codegen Errors *)
exception Unsupported of string
exception FunctionAlreadyExists of string
```

```
exception VariableAlreadyExists of string
exception IdentifierNotFound of string
exception TypeMismatch of string
exception BadFunctionCall of string
exception InvalidMainSignature of string
exception InvalidFunctionSignature of string * string
exception InvalidBinaryOperation of string
exception InvalidUnaryOperation of string

(* Codegen Errors *)
exception UnknownVariable of string
exception UnknownFunction of string
exception BadPrintfArgument
exception FunctionLookupFailure of string * string
exception VariableLookupFailure of string * string
```

../source/errors.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person
    obtaining a copy of this
software and associated documentation files (the "Software"
    ), to deal in the Software
without restriction, including without limitation the
    rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
permit persons to whom the Software is furnished to do so,
    subject to the following
conditions:

The above copyright notice and this permission notice shall
    be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
```

```
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Drives the typical lexing and parsing algorithm
while adding pertinent source, line and character information. *)

type context = {
    mutable source_name : string;
    mutable source_code : string;
  mutable original_source_code : string;
  mutable token_count : int;
    mutable token : Parser.token * Base.token_source;
}

let lex sourcename lexbuf =
  let rec acc lexbuf tokennumber tokens =
    let next_token = Scanner.token lexbuf
    and startp = Lexing.lexeme_start_p lexbuf
    and endp = Lexing.lexeme_end_p lexbuf
    in
    let line = startp.Lexing.pos_lnum
    and relpos = (1 + startp.Lexing.pos_cnum - startp.Lexing.pos_bol)
    and endrelpos = (1 + endp.Lexing.pos_cnum - endp.Lexing.pos_bol)
    and abspos = startp.Lexing.pos_cnum
    and endabspos = endp.Lexing.pos_cnum
    in
    let create_token token =
      let t = ( token, { Base.token_source_name = sourcename; Base.token_number = tokennumber;
        Base.token_line_number = line; Base.token_line_start = startp.Lexing.pos_bol;
        Base.token_column_range = (relpos, endrelpos); Base.token_character_range = (abspos, endabspos) }
      ) in
      t
    in
    match next_token with
    | Parser.EOF as token -> ( create_token token ) :: tokens
    | token -> ( create_token token ) :: ( acc lexbuf ( 1 + tokennumber ) tokens )
  in
  acc lexbuf 0 []
```

```
let parse context token_list =
  (* Keep a reference to the original token list
  And use that to dereference rather than whatever crap we get from
  the channel *)
  let tokenlist = ref(token_list) in
  let tokenizer _ = match !tokenlist with
  (* Break each token down into pieces, info and all*)
  | (token, info) :: rest ->
    context.source_name <- info.Base.token_source_name;
    context.token_count <- 1 + context.token_count;
    context.token <- ( token, info );
    (* Shift the list down by one by referencing
    the beginning of the rest of the list *)
    tokenlist := rest;
    (* return token we care about *)
    token
  (* The parser stops calling the tokenizer when
  it hits EOF: if it reaches the empty list, WE SCREWED UP *)
  | [] -> raise (Errors.MissingEoF)
  in
  (* Pass in an empty channel built off a cheap string
  and then ignore the fuck out of it in our 'tokenizer'
  internal function *)
  let program = Parser.program tokenizer (Lexing.from_string "") in
  program

let analyze program =
  (* TODO: other important checks and semantic analysis here
  that will create a proper checked program type*)
  let sem = Semant.check program in
  sem
```

../source/driver.ml

```
(* LePiX - LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the
```

```
rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Routines for preprocessing source code. *)

type pre_context = {
      mutable source_name : string;
      mutable source_code : string;
      mutable original_source_code : string;
  mutable token_count : int;
      mutable token : Preparser.token * Base.token_source;
}

let pre_lex sourcename lexbuf =
  let rec acc lexbuf tokens tokennumber =
    let next_token = Prescanner.token lexbuf
    and startp = Lexing.lexeme_start_p lexbuf
    and endp = Lexing.lexeme_end_p lexbuf
    in
    let line = startp.Lexing.pos_lnum
    and relpos = (1 + startp.Lexing.pos_cnum - startp.Lexing.pos_bol)
    and endrelpos = (1 + endp.Lexing.pos_cnum - endp.Lexing.pos_bol)
    and abspos = startp.Lexing.pos_cnum
```

```
      and endabspos = endp.Lexing.pos_cnum
      in
      let create_token token =
        let t = ( token, { Base.token_source_name = sourcename; Base.token_number = tokennumber;
          Base.token_line_number = line; Base.token_line_start = startp.Lexing.pos_bol;
          Base.token_column_range = (relpos, endrelpos); Base.token_character_range = (abspos, endabspos) }
        ) in
        t
      in
      let rec matcher = function
        | [] -> raise (Errors.MissingEoF)
        | Preparser.EOF :: [] -> ( create_token Preparser.EOF ) :: tokens
        | token :: [] -> ( create_token token ) :: ( acc lexbuf tokens ( 1 + tokennumber ) )
        | token :: rest -> ( create_token token ) :: ( matcher rest )
      in matcher next_token
    in acc lexbuf [] 0

let pre_parse context token_list =
    (* Keep a reference to the original token list
    And use that to dereference rather than whatever crap we get from
    the channel *)
    let tokenlist = ref(token_list) in
    let tokenizer _ = match !tokenlist with
    (* Break each token down into pieces, info and all*)
    | (token, info) :: rest ->
      context.source_name <- info.Base.token_source_name;
      context.token_count <- 1 + context.token_count;
      context.token <- ( token, info );
      (* Shift the list down by one by referencing
      the beginning of the rest of the list *)
      tokenlist := rest;
      (* return token we care about *)
      token
    (* The parser stops calling the tokenizer when
    it hits EOF: if it reaches the empty list, WE SCREWED UP *)
    | [] -> raise (Errors.MissingEoF)
    in
```

```
  (* Pass in an empty channel built off a cheap string
  and then ignore the fuck out of it in our 'tokenizer'
  internal function *)
  let past = Preparser.source tokenizer (Lexing.from_string "") in
  past

let rec pre_process context source source_text =
  let source_name = Base.target_to_string source in
  context.source_name <- source_name;
  context.source_code <- source_text;
  let reldir = match source with
    | Base.Pipe -> ( Sys.getcwd () )
    | Base.File(f) -> Filename.dirname f
  in
  let generate v p = match p with
    | Preast.Text(s) -> v ^ s
    | Preast.ImportString(f) -> v ^ "\"" ^ Io.read_file_text (Filename.concat reldir f) ^ "\""
    | Preast.ImportSource(f) -> let realf = (Filename.concat reldir f) in
      let ftext = Io.read_file_text realf in
      let processedtext = ( pre_process context ( Base.File(f) ) ftext ) in
      v ^ processedtext
  in
  let tokenstream = pre_lex source_name ( Lexing.from_string source_text ) in
  (*TODO: debug shit tokens at a later date*)
  (*print_endline ( Representation.preparser_token_list_to_string tokenstream );*)
  let past = pre_parse context tokenstream in
  List.fold_left generate "" past
```

../source/predriver.ml

```
(* LePiX - LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
```

```
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Source types for preprocessing LePiX source code. *)

type pre_blob =
  | Text of string
  | ImportString of string
  | ImportSource of string

type pre_source = pre_blob list
```

../source/preast.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall
```

```
be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Types and routines for the abstract syntax tree and
representation of a LePiX program. *)

type id = string

type qualified_id = id list

type builtin_type =
  | Auto
  | Void
  | Bool
  | Int of int
  | Float of int
  | String
  | Memory

type constness = bool
type referenceness = bool

type type_qualifier = constness * referenceness

type type_name =
  | BuiltinType of builtin_type * type_qualifier
  | Array of type_name * int * type_qualifier
  | SizedArray of type_name * int * int list *
    type_qualifier
  | Function of type_name * type_name list * type_qualifier

let no_qualifiers = (false, false)
```

```
let void_t = BuiltinType(Void, no_qualifiers)
let string_t = BuiltinType(String, no_qualifiers)
let int32_t = BuiltinType(Int(Base.
    default_integral_bit_width), no_qualifiers)
let float64_t = BuiltinType(Float(64), no_qualifiers)

type binding = id * type_name

let add_const (id, t) = match t with
  | BuiltinType(bt, tq) -> let ( _, refness ) = tq in
    (id, BuiltinType(bt, (true, refness)))
  | Array(tn, d, tq) -> let (_, refness) = tq in
    (id, Array(tn, d, (true, refness)))
  | SizedArray(tn, d, il, tq) -> let (_, refness) = tq in
    (id, SizedArray(tn, d, il, (true, refness)))
  | Function(tn, pl, tq) -> let (_, refness) = tq in
    (id, Function(tn, pl, (true, refness)))

type binary_op = Add | Sub | Mult | Div | Modulo
  | AddAssign | SubAssign | MultAssign | DivAssign |
    ModuloAssign
  | Equal | Neq | Less | Leq | Greater | Geq
  | And | Or

type prefix_op =
  | Neg | Not | PreIncrement | PreDecrement

type postfix_op =
  PostIncrement | PostDecrement

type literal =
  | BoolLit of bool
  | IntLit of int64 * int
  | FloatLit of float * int
  | StringLit of string

type expression =
  | Literal of literal
  | ObjectInitializer of expression list
  | ArrayInitializer of expression list
  | QualifiedId of qualified_id
  | Member of expression * qualified_id
  | Call of expression * expression list
  | Index of expression * expression list
  | BinaryOp of expression * binary_op * expression
```

```
  | PrefixUnaryOp of prefix_op * expression
  | Assignment of expression * expression
  | Noop

type parallel_expression =
  | Invocations of expression
  | ThreadCount of expression

type variable_definition =
  | VarBinding of binding * expression

type general_statement =
  | ExpressionStatement of expression
  | VariableStatement of variable_definition

type control_initializer = general_statement list * general_statement

type statement =
  | General of general_statement
  | Return of expression
  | Break of int
  | Continue
  | ParallelBlock of parallel_expression list * statement list
  | AtomicBlock of statement list
  | IfBlock of control_initializer * statement list
  | IfElseBlock of control_initializer * statement list * statement list
  | WhileBlock of control_initializer * statement list
  | ForBlock of general_statement list * expression * expression list * statement list
  | ForByToBlock of expression * expression * expression * statement list

type function_definition =
  qualified_id (* Name *)
  * binding list (* Parameters *)
  * type_name (* Return Type *)
  * statement list (* Body *)

type basic_definition =
  | VariableDefinition of variable_definition
  | FunctionDefinition of function_definition
```

```
type import_definition =
  | LibraryImport of qualified_id

type definition =
  | Import of import_definition
  | Basic of basic_definition
  | Namespace of qualified_id * definition list

type program =
  | Program of definition list

(* Useful destructuring and common operations *)
let binding_type = function
  | (_, qt) -> qt

let binding_name = function
  | (n, _) -> n
```

../source/ast.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
```

```
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Semantic checking for the Lepix compiler that will produce a new
SemanticProgram type with things like locals group into a single type
and type promotions / conversions organized for operators. *)

module StringMap = Map.Make(String)

type s_prefix_op = Ast.prefix_op
type s_binary_op = Ast.binary_op
type s_qualified_id = Ast.qualified_id
type s_id = Ast.id
type s_type_qualifier = Ast.type_qualifier
type s_builtin_type = Ast.builtin_type

type s_type_name =
  | SBuiltinType of s_builtin_type * s_type_qualifier
  | SArray of s_type_name * int * s_type_qualifier
  | SSizedArray of s_type_name * int * int list * s_type_qualifier
  | SFunction of s_type_name * s_type_name list * s_type_qualifier
  | SOverloads of s_type_name list
  | SAlias of s_qualified_id * s_qualified_id

let no_qualifiers = Ast.no_qualifiers

let void_t = SBuiltinType(Ast.Void, Ast.no_qualifiers)
let auto_t = SBuiltinType(Ast.Auto, Ast.no_qualifiers)
let string_t = SBuiltinType(Ast.String, Ast.no_qualifiers)
let bool_t = SBuiltinType(Ast.Bool, Ast.no_qualifiers)
let int32_t = SBuiltinType(Ast.Int(32), Ast.no_qualifiers)
let int64_t = SBuiltinType(Ast.Int(64), Ast.no_qualifiers)
let float64_t = SBuiltinType(Ast.Float(64), Ast.no_qualifiers)

type s_binding = s_id * s_type_name

type s_literal =
  | SBoolLit of bool
```

```
  | SIntLit of int64 * int
  | SFloatLit of float * int
  | SStringLit of string

type s_expression =
  | SObjectInitializer of s_expression list * s_type_name
  | SArrayInitializer of s_expression list * s_type_name
  | SLiteral of s_literal
  | SQualifiedId of s_qualified_id * s_type_name
  | SMember of s_expression * s_qualified_id * s_type_name
  | SCall of s_expression * s_expression list * s_type_name
  | SIndex of s_expression * s_expression list * s_type_name
  | SBinaryOp of s_expression * s_binary_op * s_expression * s_type_name
  | SPrefixUnaryOp of s_prefix_op * s_expression * s_type_name
  | SAssignment of s_expression * s_expression * s_type_name
  | SNoop

type s_locals =
  | SLocals of s_binding list

type s_parameters =
  | SParameters of s_binding list

type s_variable_definition =
  | SVarBinding of s_binding * s_expression

type s_general_statement =
  | SGeneralBlock of s_locals * s_general_statement list
  | SExpressionStatement of s_expression
  | SVariableStatement of s_variable_definition

type s_capture =
  | SParallelCapture of s_binding list

type s_control_initializer =
  | SControlInitializer of s_general_statement * s_expression

type s_parallel_expression =
  | SInvocations of s_expression
  | SThreadCount of s_expression
```

```
type s_statement =
  | SBlock of s_locals * s_statement list
  | SGeneral of s_general_statement
  | SReturn of s_expression
  | SBreak of int
  | SContinue

  | SIfBlock of s_control_initializer (* Init statements
   for an if block *)
   * s_statement (* If code *)

  | SIfElseBlock of s_control_initializer (* Init
   statements for an if-else block *)
   * s_statement (* If code *)
   * s_statement (* Else code *)

  | SWhileBlock of s_control_initializer (* Init statements
    plus ending conditional for a while loop *)
   * s_statement (* code inside the while block, locals
   and statements *)

  | SForBlock of s_control_initializer  (* Init statements
   plus ending conditional for a for loop *)
   * s_expression list (* Post-loop expressions (increment
   /decrement) *)
   * s_statement (* Code inside *)

  | SParallelBlock of s_parallel_expression list (*
   Invocation parameters passed to kickoff function *)
   * s_capture (* Capture list: references to outside
   variables *)
   * s_statement (* Locals and their statements *)

  | SAtomicBlock of s_statement (* code in the atomic block
    *)


type s_function_definition = {
  func_name : s_qualified_id;
  func_parameters : s_parameters;
     func_return_type : s_type_name;
  func_body : s_statement list;
}

```

```
type s_basic_definition =
  | SVariableDefinition of s_variable_definition
  | SFunctionDefinition of s_function_definition

type s_builtin_library =
  | Lib

let builtin_library_names = [
  ( "lib", Lib )
]

type s_loop =
  | SFor
  | SWhile

type s_module =
  | SCode of string
  | SDynamic of string
  | SBuiltin of s_builtin_library

type s_definition =
  | SBasic of s_basic_definition

type s_attributes = {
    attr_parallelism : bool;
  attr_arrays : int;
  attr_strings : bool;
}

type s_environment = {
  env_usings : string list;
  env_symbols : s_type_name StringMap.t;
  env_definitions : s_type_name StringMap.t;
  env_imports : s_module list;
  env_loops : s_loop list;
}

type s_program =
  | SProgram of s_attributes * s_environment * s_definition
    list

(* Helping functions *)
let rec coerce_type_name_of_s_expression injected =
   function
  | SObjectInitializer(a, _) -> SObjectInitializer(a,
```

```
    injected)
  | SArrayInitializer(a, _) -> SArrayInitializer(a, injected)
  | SQualifiedId(a, _) -> SQualifiedId(a, injected)
  | SMember(a, b, _) -> SMember(a, b, injected)
  | SCall(a, b, _) -> SCall(a, b, injected)
  | SIndex(a, b, _) -> SIndex(a, b, injected)
  | SBinaryOp(a, b, c, _) -> SBinaryOp(a, b, c, injected)
  | SPrefixUnaryOp(a, b, _) -> SPrefixUnaryOp(a, b, injected)
  | SAssignment(a, b, _) -> SAssignment(a, b, injected)
  | e -> e

let unqualify = function
  | SBuiltinType(bt, _) -> SBuiltinType(bt, no_qualifiers)
  | SArray(tn, d, _) -> SArray(tn, d, no_qualifiers)
  | SSizedArray(tn, d, il, _) -> SSizedArray(tn, d, il, no_qualifiers)
  | SFunction(tn, pl, _) -> SFunction(tn, pl, no_qualifiers)
  | t -> t

let string_of_qualified_id qid =
  ( String.concat "." qid )

let parameter_bindings = function
  | SParameters( bl ) -> bl

let type_name_of_s_literal = function
  | SBoolLit(_) -> bool_t
  | SIntLit(_, b) -> SBuiltinType( Ast.Int(b), no_qualifiers )
  | SFloatLit(_,b ) -> SBuiltinType( Ast.Float(b), no_qualifiers )
  | SStringLit(_) -> string_t

let rec type_name_of_s_expression = function
  | SObjectInitializer(_, t) -> t
  | SArrayInitializer(_, t) -> t
  | SLiteral(lit) -> type_name_of_s_literal lit
  | SQualifiedId(_, t) -> t
  | SMember(_, _, t) -> t
  | SCall(_, _, t) -> t
  | SIndex(_, _, t) -> t
  | SBinaryOp(_, _, _, t) -> t
```

```
  | SPrefixUnaryOp(_, _, t) -> t
  | SAssignment(_, _, t) -> t
  | SNoop -> void_t

let return_type_name = function
  | SFunction(rt,_, _) -> rt
  | t -> t

let args_type_name = function
  | SFunction(_, args, _) -> args
  | t -> []

let mangled_name_of_type_qualifier = function
  | (_, referencess) -> if referencess then "p!" else "!"

let type_name_of_s_function_definition fdef =
  let bl = parameter_bindings fdef.func_parameters in
  let argst = List.map ( fun (_, t) -> t ) bl in
  let rt = fdef.func_return_type in
  SFunction( rt, argst, no_qualifiers )

let mangled_name_of_builtin_type = function
  | Ast.Void -> "v"
  | Ast.Auto -> "a"
  | Ast.Bool -> "b"
  | Ast.Int(n) -> "i" ^ string_of_int n
  | Ast.Float(n) -> "f" ^ string_of_int n
  | Ast.String -> "s"
  | Ast.Memory -> "m"

let rec mangled_name_of_s_type_name = function
  | SBuiltinType( bt, tq ) ->
    mangled_name_of_type_qualifier tq ^
    mangled_name_of_builtin_type bt
  | SArray( tn, dims, tq ) ->
    mangled_name_of_type_qualifier tq ^ "a" ^ string_of_int
    dims ^ ";" ^ mangled_name_of_s_type_name tn
  | SSizedArray( tn, dims, sizes, tq ) ->
    mangled_name_of_type_qualifier tq ^ "a" ^ string_of_int
    dims ^ ";" ^ mangled_name_of_s_type_name tn
  | SFunction( rt, pl, tq ) ->
    mangled_name_of_type_qualifier tq ^ "r;" ^ ( String.
    concat ";" ( List.map mangled_name_of_s_type_name pl ) )
     ^ ";" ^ mangled_name_of_s_type_name rt
  | _ -> "UNSUPPORTED"
```

```
let mangle_name_args qid tnl =
  string_of_qualified_id qid ^
  if ( List.length tnl ) > 0 then
    "_" ^ ( String.concat "_" ( List.map mangled_name_of_s_type_name tnl ) )
  else
    ""

let mangle_name qid = function
  | SFunction(rt, pl, tq) -> mangle_name_args qid pl
  | _ -> string_of_qualified_id qid
```

../source/semast.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Contains routines for string-ifying various parts of the
infrastructure of the compiler, to make it easy to
```

```
    understand what the fuck we're doing. *)

(* Lexer types: dumping and pretty printing tokens *)

module StringMap = Map.Make(String)

let preparser_token_to_string = function
  | Preparser.HASH -> "HASH"
  | Preparser.IMPORT -> "IMPORT"
  | Preparser.STRING -> "STRING"
  | Preparser.TEXT(s) -> "TEXT(" ^ s ^ ")"
  | Preparser.STRINGLITERAL(s) -> "STRINGLITERAL(" ^ s ^ ")"
  | Preparser.EOF -> "EOF"


let parser_token_to_string = function
  | Parser.LPAREN -> "LPAREN"
  | Parser.RPAREN -> "RPAREN"
  | Parser.LBRACE -> "LBRACE"
  | Parser.RBRACE -> "RBRACE"
  | Parser.LSQUARE -> "LSQUARE"
  | Parser.RSQUARE -> "RSQUARE"
  | Parser.SEMI -> "SEMI"
  | Parser.COMMA -> "COMMA"
  | Parser.PLUSPLUS -> "PLUSPLUS"
  | Parser.MINUSMINUS -> "MINUSMINUS"
  | Parser.PLUS -> "PLUS"
  | Parser.MINUS -> "MINUS"
  | Parser.TIMES -> "TIMES"
  | Parser.DIVIDE -> "DIVIDE"
  | Parser.MODULO -> "MODULO"
  | Parser.PLUSASSIGN -> "PLUSASSIGN"
  | Parser.MINUSASSIGN -> "MINUSASSIGN"
  | Parser.TIMESASSIGN -> "TIMESASSIGN"
  | Parser.DIVIDEASSIGN -> "DIVIDEASSIGN"
  | Parser.MODULOASSIGN -> "MODULOASSIGN"
  | Parser.ASSIGN -> "ASSIGN"
  | Parser.EQ -> "EQ"
  | Parser.NEQ -> "NEQ"
  | Parser.LT -> "LT"
  | Parser.LEQ -> "LEQ"
  | Parser.GT -> "GT"
  | Parser.GEQ -> "GEQ"
  | Parser.AND -> "AND"
```

```
  | Parser.OR -> "OR"
  | Parser.NOT -> "NOT"
  | Parser.DOT -> "DOT"
  | Parser.AMP -> "AMPERSAND"
  | Parser.COLON -> "COLON"
  | Parser.PARALLEL -> "PARALLEL"
  | Parser.INVOCATIONS -> "INVOCATIONS"
  | Parser.THREADCOUNT -> "THREADCOUNT"
  | Parser.ATOMIC -> "ATOMIC"
  | Parser.VAR -> "VAR"
  | Parser.LET -> "LET"
  | Parser.CONST -> "CONST"
  | Parser.FUN -> "FUN"
  | Parser.NAMESPACE -> "NAMESPACE"
  | Parser.IF -> "IF"
  | Parser.ELSE -> "ELSE"
  | Parser.FOR -> "FOR"
  | Parser.TO -> "TO"
  | Parser.BY -> "BY"
  | Parser.WHILE -> "WHILE"
  | Parser.RETURN -> "RETURN"
  | Parser.INT(b) -> "INT" ^ string_of_int b
  | Parser.FLOAT(b) -> "FLOAT" ^ string_of_int b
  | Parser.BOOL -> "BOOL"
  | Parser.STRING -> "STRING"
  | Parser.VOID -> "VOID"
  | Parser.AUTO -> "AUTO"
  | Parser.MEMORY -> "MEMORY"
  | Parser.TRUE -> "TRUE"
  | Parser.FALSE -> "FALSE"
  | Parser.BREAK -> "BREAK"
  | Parser.CONTINUE -> "CONTINUE"
  | Parser.IMPORT -> "IMPORT"
  | Parser.STRINGLITERAL(s) -> "STRINGLITERAL(" ^ s ^ ")"
  | Parser.INTLITERAL(i) -> "INTLITERAL(" ^ Num.
 string_of_num i ^ ")"
  | Parser.FLOATLITERAL(f) -> "FLOATLITERAL(" ^ Num.
 string_of_num f ^ ")"
  | Parser.ID(s) -> "ID(" ^ s ^ ")"
  | Parser.EOF -> "EOF"

let token_range_to_string (x, y) =
  let range_is_wide = ( y - x > 1 ) in
  if range_is_wide then
    ( string_of_int x ^ "-" ^ string_of_int y,
```

```
    range_is_wide )
  else
    ( string_of_int x, range_is_wide )

let token_source_to_string t =
  let (s, _) = token_range_to_string t.Base.
    token_column_range in
  string_of_int t.Base.token_line_number
  ^ ":"
  ^ s

let preparser_token_list_to_string token_list =
  let rec helper = function
  | (token, pos) :: tail ->
    "[" ^ ( preparser_token_to_string token ) ^ ":"
    ^ token_source_to_string pos ^ "] "
    ^ helper tail
  | [] -> "\n" in helper token_list

let parser_token_list_to_string token_list =
  let rec helper = function
  | (token, pos) :: tail ->
    "[" ^ ( parser_token_to_string token ) ^ ":"
    ^ token_source_to_string pos ^ "] "
    ^ helper tail
  | [] -> "\n" in helper token_list


(* Program types:
dumping and Pretty-Printing *)

let string_of_id i = i

let string_of_qualified_id qid = ( String.concat "." ( List
    .map string_of_id qid ) )

let string_of_binary_op = function
  | Ast.Add -> "+"
  | Ast.Sub -> "-"
  | Ast.Mult -> "*"
  | Ast.Div -> "/"
  | Ast.Modulo -> "%"
  | Ast.AddAssign -> "+="
  | Ast.SubAssign -> "-="
  | Ast.MultAssign -> "*="
```

```
  | Ast.DivAssign -> "/="
  | Ast.ModuloAssign -> "%="
  | Ast.Equal -> "=="
  | Ast.Neq -> "!="
  | Ast.Less -> "<"
  | Ast.Leq -> "<="
  | Ast.Greater -> ">"
  | Ast.Geq -> ">="
  | Ast.And -> "&&"
  | Ast.Or -> "||"

let string_of_unary_op = function
  | Ast.Neg -> "-"
  | Ast.Not -> "!"
  | Ast.PreDecrement -> "--"
  | Ast.PreIncrement -> "++"

let rec string_of_expression = function
  | Ast.Literal(Ast.IntLit(l, _)) -> Int64.to_string l
  | Ast.Literal(Ast.BoolLit(true)) -> "true"
  | Ast.Literal(Ast.BoolLit(false)) -> "false"
  | Ast.Literal(Ast.StringLit(s)) -> "\"" ^ s ^ "\""
  | Ast.Literal(Ast.FloatLit(f, _)) -> string_of_float f
  | Ast.QualifiedId(qid) -> string_of_qualified_id qid
  | Ast.BinaryOp(e1, o, e2) ->
    string_of_expression e1 ^ " " ^ string_of_binary_op o ^ " " ^ string_of_expression e2
  | Ast.PrefixUnaryOp(o, e) -> string_of_unary_op o ^ string_of_expression e
  | Ast.Index(e, l) -> string_of_expression e ^ "[" ^ (String.concat ", " (List.map string_of_expression l)) ^ "]"
  | Ast.Member(e, qid) -> string_of_expression e ^ "." ^ string_of_qualified_id qid
  | Ast.Assignment(e1, e2) -> string_of_expression e1 ^ " = " ^ string_of_expression e2
  | Ast.Call(e, el) ->
    string_of_expression e ^ "(" ^ String.concat ", " (List.map string_of_expression el) ^ ")"
  | Ast.Noop -> "{ noop }"
  | Ast.ArrayInitializer(el) -> "[ " ^ String.concat ", " (List.map string_of_expression el) ^ " ]"
  | Ast.ObjectInitializer(el) -> "{ " ^ String.concat ", " (List.map string_of_expression el) ^ " }"

```

```
let string_of_parallel_expression = function
  | Ast.ThreadCount(e) -> "thread_count = " ^ string_of_expression e
  | Ast.Invocations(e) -> "invocations = " ^ string_of_expression e

let rec string_of_expression_list el =
  String.concat ", " ( List.map string_of_expression el )

let rec string_of_builtin_type = function
  | Ast.Auto -> "auto"
  | Ast.Bool -> "bool"
  | Ast.Int(b) -> "int" ^ string_of_int b
  | Ast.Float(b) -> "float" ^ string_of_int b
  | Ast.String -> "string"
  | Ast.Memory -> "memory"
  | Ast.Void -> "void"

let string_of_type_qualifier = function
  | (c, r) -> ( if c then "const" else "" ) ^ ( if r then " &" else "" )

let rec string_of_type_name tn =
  let tqual tq =
    let s = string_of_type_qualifier tq in
    if s = "" then "" else s ^ " "
  in match tn with
  | Ast.BuiltinType(t, tq) -> tqual tq ^ string_of_builtin_type t
  | Ast.Array(t, d, tq) -> tqual tq ^ string_of_type_name t ^ ( String.make d '[' ) ^ ( String.make d ']' )
  | Ast.SizedArray(t, d, il, tq) -> tqual tq ^ string_of_type_name t ^ ( String.make d '[' ) ^ ( String.concat ", " ( List.map string_of_int il ) ) ^ ( String.make d ']' )
  | Ast.Function(r, args, tq) -> tqual tq ^ "(" ^ ( String.concat ", " ( List.map ( fun v -> string_of_type_name v ) args ) ) ^ ")" ^ string_of_type_name r

let string_of_binding = function
  | (n, t) -> n ^ " : " ^ string_of_type_name t

let string_of_variable_definition = function
  | Ast.VarBinding(b, Ast.Noop) -> "var " ^ string_of_binding b
```

```
  | Ast.VarBinding(b, e) -> "var " ^ string_of_binding b ^ " = " ^ string_of_expression e

let string_of_general_statement = function
  | Ast.ExpressionStatement(e) -> string_of_expression e
  | Ast.VariableStatement(v) -> string_of_variable_definition v

let string_of_condition_initializer = function
  | (il, cond) -> ( String.concat "; " ( List.map string_of_general_statement il ) )
  ^ ( if ( List.length il ) > 0 then ";" else "" )
  ^ ( string_of_general_statement cond )

let rec string_of_statement s =
  let string_of_statement_list sl = String.concat "" (List.map string_of_statement sl) in
  match s with
    | Ast.General(b) -> string_of_general_statement b ^ ";\n";
    | Ast.Return(expr) -> "return " ^ string_of_expression expr ^ ";\n";
    | Ast.IfBlock(ilcond, s) -> "if (" ^ string_of_condition_initializer ilcond ^ ")"
      ^ "{\n" ^  string_of_statement_list s ^ "}\n"
    | Ast.IfElseBlock(ilcond, s, s2) -> "if (" ^ string_of_condition_initializer ilcond ^ ")"
      ^ "{\n" ^  string_of_statement_list s ^ "}\n"
      ^ "else {\n" ^ string_of_statement_list s2 ^ "}\n"
    | Ast.ForBlock(gsl, cond, incrl, sl) -> "for (" ^ ( String.concat ", " (List.map string_of_general_statement gsl) ) ^ "; "
      ^ string_of_expression cond ^ "; "
      ^ string_of_expression_list incrl  ^ ") {\n" ^ string_of_statement_list sl ^ "}\n"
    | Ast.ForByToBlock(e1, e2, e3, sl) -> "for (" ^ string_of_expression e1  ^ " to " ^ string_of_expression e2 ^ " by " ^ string_of_expression e3  ^ ") {\n" ^ string_of_statement_list sl ^ "}\n"
    | Ast.WhileBlock(ilcond, s) -> "while (" ^ string_of_condition_initializer ilcond ^ ") {\n"
      ^ string_of_statement_list s ^ "}\n"
    | Ast.Break(n) -> ( if n == 1 then "break" else "break" ^ string_of_int n ) ^ ";\n"
    | Ast.Continue -> "continue;\n"
```

```
    | Ast.ParallelBlock(pel,sl) -> "parallel(" ^ (String.concat ", " (List.map string_of_parallel_expression pel)) ^ ") {"
      ^ "\n" ^ string_of_statement_list sl ^ "}\n"
    | Ast.AtomicBlock(sl) -> "atomic {\n" ^ string_of_statement_list sl ^ "}\n"

let string_of_statement_list sl =
  String.concat "" (List.map string_of_statement sl)

let string_of_function_definition = function
  | ( name, parameters, return_type, body) ->
    "fun " ^ string_of_qualified_id name
      ^ "(" ^ (String.concat ", " (List.map string_of_binding parameters)) ^ ") : "
      ^ string_of_type_name return_type  ^ " {\n"
      ^ string_of_statement_list body
      ^ "}\n"

let rec string_of_basic_definition = function
  | Ast.FunctionDefinition(fdef) -> string_of_function_definition fdef
  | Ast.VariableDefinition(vdef) -> string_of_variable_definition vdef ^ ";\n"

let string_of_import_definition = function
  | Ast.LibraryImport(qid) -> "import " ^ string_of_qualified_id qid ^ "\n"

let rec string_of_definition = function
  | Ast.Import(idef) -> string_of_import_definition idef
  | Ast.Basic(bdef) -> string_of_basic_definition bdef
  | Ast.Namespace(qid, defs) -> "namespace " ^ string_of_qualified_id qid ^ " {\n"
    ^ (String.concat "" (List.map string_of_definition defs) ) ^ "}\n"

let string_of_program = function
  | Ast.Program(p) -> let s = (String.concat "" (List.map string_of_definition p) ) in
  Base.brace_tabulate s 0

(* Semantic Program types:
dumping and pretty printing *)

```

```
let rec string_of_s_type_name tn =
  let tqual tq =
    let s = string_of_type_qualifier tq in
    if s = "" then "" else s ^ " "
  in match tn with
  | Semast.SBuiltinType(t, tq) -> tqual tq ^ string_of_builtin_type t
  | Semast.SArray(t, d, tq) -> tqual tq ^ string_of_s_type_name t ^ ( String.make d '[' ) ^ ( String.make d ']' )
  | Semast.SSizedArray(t, d, il, tq) -> tqual tq ^ string_of_s_type_name t ^ ( String.make d '[' ) ^ ( String.concat ", " ( List.map string_of_int il ) ) ^ ( String.make d ']' )
  | Semast.SFunction(r, args, tq) -> tqual tq ^ "(" ^  ( String.concat ", " ( List.map ( fun v -> string_of_s_type_name v ) args ) ) ^ ")" ^ string_of_s_type_name r
  | Semast.SOverloads(fl) -> "overloads["
    ^ ( String.concat ", " ( List.map string_of_s_type_name fl ) )
    ^ "]"
  | Semast.SAlias(target, source) -> "using " ^ string_of_qualified_id target ^ " -> " ^ string_of_qualified_id source

let string_of_s_binding = function
  | (n, t) -> n ^ " : " ^ string_of_s_type_name t

let string_of_s_locals = function
  | Semast.SLocals(bl) -> if ( List.length bl < 1 ) then "" else ( String.concat ";\n" ( List.map string_of_s_binding bl ) ) ^ ";"

let string_of_s_literal = function
  | Semast.SBoolLit(b) -> string_of_bool b
  | Semast.SIntLit(i, _) -> Int64.to_string i
  | Semast.SFloatLit(f, _) -> string_of_float f
  | Semast.SStringLit(s) -> "\"" ^ s ^ "\""

let rec string_of_s_expression = function
  | Semast.SObjectInitializer(el, tn) ->
    string_of_s_type_name tn ^ "{ "
    ^ String.concat ", " ( List.map string_of_s_expression el )
```

```
      ^ " }"
    | Semast.SArrayInitializer(el, tn) ->
      string_of_s_type_name tn ^ "[ "
      ^ String.concat ", " ( List.map string_of_s_expression el )
      ^ " ]"
    | Semast.SLiteral(l) -> string_of_s_literal l
    | Semast.SQualifiedId(qid, tn) -> "[[ " ^ string_of_s_type_name tn ^ " ]] " ^ string_of_qualified_id qid
    | Semast.SMember(e, qid, tn) -> "[[ " ^ string_of_s_type_name tn ^ " ]] " ^ string_of_s_expression e ^ "." ^ string_of_qualified_id qid
    | Semast.SCall(e, el, tn) -> "[[" ^ string_of_s_type_name tn ^ "]]" ^ string_of_s_expression e ^ "( " ^ ( String.concat ", " ( List.map string_of_s_expression el ) ) ^ " )"
    | Semast.SIndex(e, el, tn) -> "[[ " ^ string_of_s_type_name tn ^ " ]] " ^ string_of_s_expression e ^ "[ " ^ ( String.concat ", " ( List.map string_of_s_expression el ) ) ^ " ]"
    | Semast.SBinaryOp(l, op, r, tn) -> "[[ " ^ string_of_s_type_name tn ^ " ]] " ^ string_of_s_expression l ^ " " ^ string_of_binary_op op ^ " " ^ string_of_s_expression r
    | Semast.SPrefixUnaryOp(op, r, tn) -> "[[ " ^ string_of_s_type_name tn ^ " ]] " ^ string_of_unary_op op ^ string_of_s_expression r
    | Semast.SAssignment(l, r, tn) -> "[[ " ^ string_of_s_type_name tn ^ " ]] " ^ string_of_s_expression l ^ " = " ^ string_of_s_expression r
    | Semast.SNoop -> "(noop)"

let string_of_s_capture = function
    | Semast.SParallelCapture(bl) -> let capturecount = List.length bl in
      if capturecount == 0 then "[[no captures]]\n" else
      "[[captures]] { " ^ ( String.concat ", " ( List.map string_of_s_binding bl ) )
      ^ " }\n"

let string_of_s_variable_definition = function
    | Semast.SVarBinding(b, e) -> "var " ^
```

```
    string_of_s_binding b ^ " = " ^ string_of_s_expression e

let rec string_of_s_general_statement = function
  | Semast.SGeneralBlock(locals, gsl) -> "{\n" ^ string_of_s_locals locals ^ "\n" ^ ( String.concat ";\n" (List.map string_of_s_general_statement gsl) ) ^ "\n}"
  | Semast.SExpressionStatement(sexpr) -> string_of_s_expression sexpr
  | Semast.SVariableStatement(v) -> string_of_s_variable_definition v

let string_of_s_general_statement_list gsl =
  String.concat ";\n" (List.map string_of_s_general_statement gsl)

let string_of_s_parallel_expression = function
  | Semast.SInvocations(e) -> string_of_s_expression e
  | Semast.SThreadCount(e) -> string_of_s_expression e

let rec string_of_s_statement s =
  let initializer_begin = function
    | Semast.SGeneralBlock(locals, gsl) ->
      let precount = List.length gsl in
      if precount > 1 then
        "{ "
        ^ string_of_s_locals locals
        ^ "\n" ^ string_of_s_general_statement_list gsl
        ^ "\n"
      else
        ""
    | _ -> ""
  in
  let initializer_end = function
    | Semast.SGeneralBlock(locals, gsl) ->
      let precount = List.length gsl in
      if precount > 1 then
        "}\n"
      else
        ""
    | _ -> ""
  in match s with
  | Semast.SBlock(locals, sl) -> "{\n" ^ string_of_s_locals locals ^ "\n" ^ ( String.concat "\n" (List.map string_of_s_statement sl) ) ^ "\n}\n"
  | Semast.SGeneral(g) -> ( string_of_s_general_statement
```

```
g ) ^ ";"
| Semast.SReturn(sexpr) -> "return " ^
string_of_s_expression sexpr ^ ";"
| Semast.SBreak(n) -> if n < 2 then "break;" else "break
" ^ string_of_int n ^ ";"
| Semast.SContinue -> "continue;"
| Semast.SAtomicBlock(s) -> "atomic {"
  ^ string_of_s_statement s
  ^ "}\n"
| Semast.SParallelBlock( pel, captures, s) ->
  "parallel(" ^ (String.concat ", " (List.map
string_of_s_parallel_expression pel)) ^ " ) {"
  ^ "\n" ^ string_of_s_capture captures
  ^ "\n" ^ string_of_s_statement s
  ^ "}\n"
| Semast.SIfBlock(Semast.SControlInitializer(inits, cond)
, s) ->
  initializer_begin inits
  ^ "if (" ^ string_of_s_expression cond ^ ") {"
  ^ "\n" ^ string_of_s_statement s
  ^ "}\n"
  ^ initializer_end inits
| Semast.SIfElseBlock(Semast.SControlInitializer(inits,
cond), is, es) ->
  initializer_begin inits
  ^ "if (" ^ string_of_s_expression cond ^ ") {"
  ^ "\n" ^ string_of_s_statement is
  ^ "}\n"
  ^ "else {"
  ^ "\n" ^ string_of_s_statement es
  ^ "}\n"
  ^ initializer_end inits
| Semast.SWhileBlock(Semast.SControlInitializer(inits,
cond), s) ->
  initializer_begin inits
  ^ "while (" ^ string_of_s_expression cond ^ ") {"
  ^ "\n" ^ string_of_s_statement s
  ^ "}\n"
  ^ initializer_end inits
| Semast.SForBlock(Semast.SControlInitializer(inits, cond
), increxprl, s) ->
  let incrl = String.concat ", " ( List.map
string_of_s_expression increxprl ) in
  initializer_begin inits
  ^ "for (;" ^ string_of_s_expression cond ^ "; " ^ incrl
```

```
    ^ ") {"
    ^ "\n" ^ string_of_s_statement s
    ^ "}\n"
    ^ initializer_end inits

let string_of_s_statement_list sl =
  ( String.concat "\n" ( List.map string_of_s_statement sl ) ) ^ "\n"

let string_of_s_block = function
  | (locals, sl) -> "{\n" ^ string_of_s_locals locals
    ^ "\n" ^ string_of_s_statement_list sl
    ^ "\n}\n"

let string_of_s_parameters = function
  | Semast.SParameters(parameters) -> String.concat ", " ( List.map string_of_s_binding parameters)

let string_of_s_function_definition f =
  "fun " ^ string_of_qualified_id f.Semast.func_name
  ^ "(" ^ string_of_s_parameters f.Semast.func_parameters ^ ") : "
  ^ string_of_s_type_name f.Semast.func_return_type  ^ " {\n"
  ^ string_of_s_statement_list f.Semast.func_body
  ^ "}\n"

let string_of_s_basic_definition = function
  | Semast.SVariableDefinition(v) -> string_of_s_variable_definition v
  | Semast.SFunctionDefinition(f) -> string_of_s_function_definition f

let string_of_s_builtin_library = function
  | Semast.Lib -> "lib"

let string_of_s_module = function
  | Semast.SCode(s) -> "import [[code]] " ^ s
  | Semast.SDynamic(s) -> "import [[dynamic]] " ^ s
  | Semast.SBuiltin(bltin) -> "import [[builtin]] " ^ string_of_s_builtin_library bltin

let rec string_of_s_definition = function
  | Semast.SBasic(b) -> string_of_s_basic_definition b ^ "\n"
```

```

let string_of_s_program = function
  | Semast.SProgram( attr, env, sdl ) ->
  let symbolacc k tn l =
    let entry = ( string_of_s_type_name tn ) ^ " | " ^ k in
    entry :: l
  and importacc m =
    string_of_s_module m
  in
  let implist = List.map importacc env.Semast.env_imports
  and symbollist = StringMap.fold symbolacc env.Semast.env_symbols []
  and defsymbollist = StringMap.fold symbolacc env.Semast.env_definitions []
  in
  let i = "imports:\n\t" ^ ( String.concat "\n\t" implist )
  and s = "symbols:\n\t" ^ ( String.concat "\n\t" symbollist )
  and d = "code symbols:\n\t" ^ ( String.concat "\n\t" defsymbollist )
  and a = "strings: " ^ string_of_bool attr.Semast.attr_strings
    ^ "\narrays: " ^ string_of_int attr.Semast.attr_arrays
    ^ "\nparallelism: " ^ string_of_bool attr.Semast.attr_parallelism
  in
  let p = String.concat "" (List.map string_of_s_definition sdl) in
    Base.brace_tabulate ( a ^ "\n\n" ^ i ^ "\n\n" ^ s ^ "\n\n" ^ d ^ "\n\n" ^ p ) 0
```

../source/representation.ml

```
(* LePiX - LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:
```

```
The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Ocamllex Scanner for LePiX Preprocessor *)

let whitespace = [' ' '\t' '\r']
let newline = ['\n']

rule token = parse
| newline as c { Lexing.new_line lexbuf; let b = Buffer.create 1024 in Buffer.add_char b c; sub_token b lexbuf }
| '#'          { let t = Preparser.HASH in t :: hash_token lexbuf  }
| _ as c       { let b = Buffer.create 1024 in Buffer.add_char b c; sub_token b lexbuf }
| eof          { [Preparser.EOF] }

and sub_token text_buf = parse
| newline as c  { Lexing.new_line lexbuf; Buffer.add_char text_buf c; sub_token text_buf lexbuf }
| '#'           { let s = Buffer.contents text_buf in let t = Preparser.TEXT(s) in let l = Preparser.HASH :: ( hash_token lexbuf ) in t :: l  }
| _ as c        { Buffer.add_char text_buf c; sub_token text_buf lexbuf }
| eof           { let s = Buffer.contents text_buf in let t = Preparser.TEXT(s) in [t; Preparser.EOF] }

and hash_token = parse
| newline    { Lexing.new_line lexbuf; token lexbuf }
| whitespace { hash_token lexbuf }
```

```
| '"'         { let b = ( Buffer.create 128 ) in string_literal b lexbuf }
| "import"    { Preparser.IMPORT :: hash_token lexbuf }
| "string"    { Preparser.STRING :: hash_token lexbuf }
| eof         { [Preparser.EOF] }
| _ as c      { raise (Errors.UnknownCharacter(String.make 1 c, ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p lexbuf ) )) }

and string_literal string_buf = parse
| newline as c { Lexing.new_line lexbuf; Buffer.add_char string_buf c; string_literal string_buf lexbuf }
| '"'            { let sl = Preparser.STRINGLITERAL( Buffer.contents string_buf ) in [sl] }
| "\\\"" as s  { Buffer.add_string string_buf s; string_literal string_buf lexbuf }
| _ as c       { Buffer.add_char string_buf c; string_literal string_buf lexbuf }
```

../source/prescanner.mll

```
%{
(* LePiX - LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
```

```
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Parser for the LePiX preprocessor: compatible with both ocamlyacc and
menhir, as we have developed against both for testing purposes. *)

%}

%token HASH
%token IMPORT STRING
%token <string> TEXT
%token <string> STRINGLITERAL
%token EOF

%start source
%type<Preast.pre_source> source
%%

blob:
| HASH IMPORT STRINGLITERAL { Preast.ImportSource($3) }
| HASH IMPORT STRING STRINGLITERAL { Preast.ImportString($4) }
| TEXT { Preast.Text($1) }

blob_list: { [] }
| blob_list blob { $2 :: $1 }

source:
| blob_list EOF { List.rev $1 }
```

../source/preparser.mly

```
(* LePiX - LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies
```

```
of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Ocamllex Scanner for LePiX Preprocessor *)

{
  open Parser
}

let whitespace = [' ' '\t' '\r']
let newline = ['\n']
let binary_digit = '0' | '1'
let hex_digit = ['0'-'9'] | ['A'-'F'] | ['a'-'f']
let octal_digit = ['0'-'7']
let decimal_digit = ['0'-'9']
let uppercase_letter = ['A'-'Z']
let lowercase_letter = ['a'-'z']
let n_digit = decimal_digit | uppercase_letter | lowercase_letter

rule token = parse
| whitespace { token lexbuf }
| newline  { Lexing.new_line lexbuf; token lexbuf }
| "/*"     { multi_comment 0 lexbuf }
| "//"     { single_comment lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
```

```
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['     { LSQUARE }
| ']'     { RSQUARE }
| ';'       { SEMI }
| ':'     { COLON }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| "+="      { PLUSASSIGN }
| "-="      { MINUSASSIGN }
| "*="      { TIMESASSIGN }
| "/="      { DIVIDEASSIGN }
| "%="      { MODULOASSIGN }
| "++"      { PLUSPLUS }
| "--"      { MINUSMINUS }
| "%"       { MODULO }
| '='       { ASSIGN }
| '&'       { AMP }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| '.'       { DOT }
| '&'       { AMP }
| "||"      { OR }
| "!"       { NOT }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "by"      { BY }
| "to"      { TO }
| "return"  { RETURN }
| "auto"    { AUTO }
| "int" ((decimal_digit+)? as s) { let bits = if s = ""
    then Base.default_integral_bit_width else (
    int_of_string s ) in INT(bits) }
| "float" ((decimal_digit+)? as s) { let bits = if s = ""
    then Base.default_floating_bit_width else (
```

```
    int_of_string s ) in FLOAT(bits) }
| "bool"    { BOOL }
| "string" { STRING }
| "void"    { VOID }
| "memory" { MEMORY }
| "true"    { TRUE }
| "false"   { FALSE }
| "var"     { VAR }
| "let"     { LET }
| "const"   { CONST }
| "fun"      { FUN }
| "parallel" { PARALLEL }
| "break" { BREAK }
| "continue" { CONTINUE }
| "invocations" { INVOCATIONS }
| "thread_count" { THREADCOUNT }
| "atomic" { ATOMIC }
| "namespace" { NAMESPACE }
| "import" { IMPORT }
| '"'       { string_literal ( Buffer.create 128 ) lexbuf }
| decimal_digit+ as lxm { INTLITERAL(Num.num_of_string lxm) }
| "0c" | "0C" { octal_int_literal lexbuf }
| "0x" | "0X" { hex_int_literal lexbuf }
| "0b" | "0B" { binary_int_literal lexbuf }
| ( "0n" | "0N" ) ( decimal_digit+ as b ) ("n" | "N") { n_int_literal (int_of_string b) lexbuf }
| '.' ['0'-'9']+ ('e' ('+'|'-')? ['0'-'9']+)? as s { FLOATLITERAL( Polyfill.num_of_string s ) }
| ['0'-'9']+ ( '.' ['0'-'9']* ('e' ('+'|'-')? ['0'-'9']+)? | ('e' ('+'|'-')? ['0'-'9']+)?) as s { try FLOATLITERAL( Polyfill.num_of_string_base 10 s) with _ -> raise ( Errors.BadNumericLiteral(s, ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p lexbuf ) )) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as s { ID(s) }
| eof { EOF }
| _ as c { raise (Errors.UnknownCharacter(String.make 1 c, ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p lexbuf ) )) }


and octal_int_literal = parse
| octal_digit+ as s { try INTLITERAL( Polyfill.
    num_of_string_base 8 s ) with _ -> raise (Errors.
```

```
    BadNumericLiteral(s, ( Lexing.lexeme_start_p lexbuf,
    Lexing.lexeme_end_p lexbuf ) )) }
| _ as c { raise (Errors.BadNumericLiteral(String.make 1 c,
     ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p
    lexbuf ) )) }


and hex_int_literal = parse
| hex_digit+ as s { try INTLITERAL( Polyfill.
    num_of_string_base 16 s ) with _ -> raise (Errors.
    BadNumericLiteral(s, ( Lexing.lexeme_start_p lexbuf,
    Lexing.lexeme_end_p lexbuf ) )) }
| _ as c { raise (Errors.BadNumericLiteral(String.make 1 c,
     ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p
    lexbuf ) )) }


and binary_int_literal = parse
| binary_digit+ as s { try INTLITERAL( Polyfill.
    num_of_string_base 2 s ) with _ -> raise (Errors.
    BadNumericLiteral(s, ( Lexing.lexeme_start_p lexbuf,
    Lexing.lexeme_end_p lexbuf ) )) }
| _ as c { raise (Errors.BadNumericLiteral(String.make 1 c,
     ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p
    lexbuf ) )) }


and n_int_literal base = parse
| n_digit+ as s    { INTLITERAL( Polyfill.
    num_of_string_base base s ) }
| _ as c { raise (Errors.BadNumericLiteral(String.make 1 c,
     ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p
    lexbuf ) )) }


and string_literal string_buffer = parse
| newline as c { Lexing.new_line lexbuf; Buffer.add_char
    string_buffer c; string_literal string_buffer lexbuf }
| '"' { let v = STRINGLITERAL( Buffer.contents
    string_buffer ) in v }
| "\\\"" as s { Buffer.add_string string_buffer s;
    string_literal string_buffer lexbuf }
| _ as c { Buffer.add_char string_buffer c; string_literal
    string_buffer lexbuf }

```

```

and multi_comment level = parse
| newline { Lexing.new_line lexbuf; multi_comment level
    lexbuf }
| "*/" { if level = 0 then token lexbuf else multi_comment
    (level-1) lexbuf }
| "/*" { multi_comment (level+1) lexbuf }
| _    { multi_comment level lexbuf }


and single_comment = parse
| newline { Lexing.new_line lexbuf; token lexbuf }
| _    { single_comment lexbuf }
```

../source/scanner.mll

```
%{
(* LePiX - LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person
    obtaining a copy of this
software and associated documentation files (the "Software"
    ), to deal in the Software
without restriction, including without limitation the
    rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
permit persons to whom the Software is furnished to do so,
    subject to the following
conditions:

The above copyright notice and this permission notice shall
    be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
```

```
(* Parser for the LePiX language: compatible with both
    ocamlyacc and
menhir, as we have developed against both for testing
    purposes. *)

%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token LSQUARE RSQUARE COLON
%token DOT
%token PARALLEL INVOCATIONS ATOMIC THREADCOUNT
%token PLUSPLUS MINUSMINUS
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token MODULO
%token PLUSASSIGN MINUSASSIGN TIMESASSIGN DIVIDEASSIGN
%token MODULOASSIGN
%token NOT AND OR EQ NEQ LT LEQ GT GEQ
%token TRUE FALSE
%token VAR LET
%token FUN TO BY
%token RETURN CONTINUE BREAK IF ELSE FOR WHILE
%token AMP CONST
%token <int> INT
%token <int> FLOAT
%token BOOL VOID STRING MEMORY
%token AUTO
%token NAMESPACE
%token IMPORT
%token <string> ID
%token <string> STRINGLITERAL
%token <Num.num> INTLITERAL
%token <Num.num> FLOATLITERAL
%token EOF

%right ASSIGN
%right PLUSASSIGN MINUSASSIGN
%right TIMESASSIGN DIVIDEASSIGN MODULOASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%right NOT NEG
```

```
%left LSQUARE
%left LPAREN
%left MINUSMINUS
%left PLUSPLUS

%start program
%type<Ast.program> program
%%

qualified_id_builder:
| ID { [$1] }
| qualified_id_builder DOT ID { $3 :: $1 }

qualified_id:
| qualified_id_builder { List.rev $1 }

builtin_type:
| AUTO { Ast.Auto }
| VOID { Ast.Void }
| BOOL { Ast.Bool }
| INT { Ast.Int($1) }
| FLOAT { Ast.Float($1) }
| STRING { Ast.String }
| MEMORY { Ast.Memory }

array_spec:
| LSQUARE RSQUARE { 1 }
| LSQUARE array_spec RSQUARE { 1 + $2 }

int_literal_list:
| INTLITERAL { [ ( Num.int_of_num $1 ) ] }
| INTLITERAL int_literal_list { ( Num.int_of_num $1 ) :: $2
  }

sized_array_spec:
| LSQUARE int_literal_list RSQUARE { ( 1, $2 ) }
| LSQUARE sized_array_spec RSQUARE { let ( d, el ) = $2 in
  (1 + d, el) }

type_category: { (false, false) }
| AMP { (false, true) }
| CONST AMP { (true, true) }
| CONST { (true, false) }

sub_type_name:
```

```
| type_category builtin_type             { Ast.BuiltinType($2, $1) }
| type_category builtin_type array_spec  { Ast.Array(Ast.BuiltinType($2, Ast.no_qualifiers), $3, $1) }
| type_category builtin_type sized_array_spec  { let (d, el) = $3 in
  if d <> ( List.length el ) then
    raise(Parsing.Parse_error)
  else
    Ast.SizedArray(Ast.BuiltinType($2, Ast.no_qualifiers), d, el, $1)
}

sub_type_name_list_builder: { [] }
| sub_type_name { [$1] }
| sub_type_name_list_builder COMMA sub_type_name { $3 :: $1 }

sub_type_name_list:
| sub_type_name_list_builder { List.rev $1 }

type_name:
| sub_type_name { $1 }
| type_category LPAREN sub_type_name_list RPAREN sub_type_name { Ast.Function($5, $3, $1) }

expression_comma_list:                      { [] }
| expression                                { [$1] }
| expression COMMA expression_comma_list { $1 :: $3 }

op_expression:
| expression TIMES expression { Ast.BinaryOp($1, Ast.Mult, $3) }
| expression DIVIDE expression { Ast.BinaryOp($1, Ast.Div, $3)  }
| expression PLUS expression { Ast.BinaryOp($1, Ast.Add, $3) }
| expression MINUS expression { Ast.BinaryOp($1, Ast.Sub, $3) }
| expression MODULO expression { Ast.BinaryOp($1, Ast.Modulo, $3) }
| expression TIMESASSIGN expression { Ast.BinaryOp($1, Ast.MultAssign, $3) }
| expression DIVIDEASSIGN expression { Ast.BinaryOp($1, Ast.DivAssign, $3)  }
```

```
| expression PLUSASSIGN expression { Ast.BinaryOp($1, Ast.AddAssign, $3) }
| expression MINUSASSIGN expression { Ast.BinaryOp($1, Ast.SubAssign, $3) }
| expression MODULOASSIGN expression { Ast.BinaryOp($1, Ast.ModuloAssign, $3) }
| expression LT expression { Ast.BinaryOp($1, Ast.Less, $3) }
| expression GT expression { Ast.BinaryOp($1, Ast.Greater, $3) }
| expression LEQ expression { Ast.BinaryOp($1, Ast.Leq, $3) }
| expression GEQ expression { Ast.BinaryOp($1, Ast.Geq, $3) }
| expression NEQ expression { Ast.BinaryOp($1, Ast.Neq, $3) }
| expression EQ expression { Ast.BinaryOp($1, Ast.Equal, $3) }
| expression AND expression { Ast.BinaryOp($1, Ast.And, $3) }
| expression OR expression { Ast.BinaryOp($1, Ast.Or, $3) }
| expression ASSIGN expression { Ast.Assignment($1, $3) }
| MINUS expression %prec NEG { Ast.PrefixUnaryOp(Ast.Neg, $2) }
| NOT expression { Ast.PrefixUnaryOp(Ast.Not, $2) }
| PLUSPLUS expression { Ast.PrefixUnaryOp(Ast.PreIncrement, $2) }
| MINUSMINUS expression { Ast.PrefixUnaryOp(Ast.PreDecrement, $2) }

value_expression:
| INTLITERAL { let v = match $1 with
  | Num.Int(i) -> Ast.IntLit( (Int64.of_int i ), Base.default_integral_bit_width)
  | Num.Big_int(bi) ->
    begin try
      Ast.IntLit( Int64.of_int( Big_int.int_of_big_int bi ), 32 )
    with
      _ -> Ast.IntLit( Big_int.int64_of_big_int bi, 64 )
    end
  | n -> Ast.FloatLit( Num.float_of_num n, Base.default_floating_bit_width )
  in
  Ast.Literal(v)
```

```
}
| FLOATLITERAL { Ast.Literal(Ast.FloatLit( Num.float_of_num
    ( $1 ), Base.default_floating_bit_width )) }
| STRINGLITERAL { Ast.Literal(Ast.StringLit($1)) }
| TRUE { Ast.Literal(Ast.BoolLit(true)) }
| FALSE { Ast.Literal(Ast.BoolLit(false)) }
| LSQUARE expression_comma_list RSQUARE { Ast.
    ArrayInitializer( $2 ) }
| LBRACE expression_comma_list RBRACE { Ast.
    ObjectInitializer( $2 ) }

postfix_expression:
| expression LSQUARE expression_comma_list RSQUARE { Ast.
    Index($1, $3) }
| expression LPAREN expression_comma_list RPAREN { Ast.Call
    ($1, $3) }

expression:
| qualified_id { Ast.QualifiedId($1) }
| value_expression { $1 }
| value_expression DOT qualified_id { Ast.Member($1, $3) }
| op_expression { $1 }
| postfix_expression { $1 }
| postfix_expression DOT qualified_id { Ast.Member($1, $3)
    }
| LPAREN expression RPAREN { $2 }
| LPAREN expression RPAREN DOT qualified_id { Ast.Member($2
    , $5) }

type_spec:
| COLON type_name { $2 }

maybe_type_spec: { Ast.BuiltinType(Ast.Auto, Ast.
    no_qualifiers) }
| COLON type_name { $2 }

binding:
| ID type_spec { ($1, $2) }

binding_list: { [] }
| binding { [$1] }
| binding COMMA binding_list { $1 :: $3 }

var_binding:
| ID type_spec { ($1, $2) }
```

```
| ID { ($1, Ast.BuiltinType(Ast.Auto, Ast.no_qualifiers)) }

variable_definition:
| VAR var_binding ASSIGN expression { Ast.VarBinding($2, $4
    ) }
| LET var_binding ASSIGN expression { Ast.VarBinding(Ast.
    add_const($2), $4) }
| VAR var_binding { Ast.VarBinding($2, Ast.Noop) }
| LET var_binding { Ast.VarBinding(Ast.add_const($2), Ast.
    Noop) }

statement_list_builder: { [] }
| statement_list_builder statement { $2 :: $1 }

statement_list :
| statement_list_builder { List.rev $1 }

parallel_binding:
| INVOCATIONS ASSIGN expression { Ast.Invocations($3) }
| THREADCOUNT ASSIGN expression { Ast.ThreadCount($3) }

parallel_binding_list_builder: { [] }
| parallel_binding { [$1] }
| parallel_binding_list_builder COMMA parallel_binding { $3
    :: $1 }

parallel_binding_list:
| parallel_binding_list_builder { List.rev $1 }

sub_general_statement:
| expression { Ast.ExpressionStatement($1) }
| variable_definition { Ast.VariableStatement($1) }

general_statement:
| sub_general_statement SEMI { Ast.General($1) }

control_initializer_builder:
| sub_general_statement { ( [$1], 1 ) }
| control_initializer_builder SEMI sub_general_statement {
    let (l, c) = $1 in ( $3 :: l, 1 + c ) }

control_initializer:
| control_initializer_builder { let ( il, c ) = $1 in if c
   < 2 then ([], List.hd il) else (List.rev ( List.tl il ),
    List.hd il) }
```

```

sub_general_statement_list_builder: { [] }
| sub_general_statement { [$1] }
| sub_general_statement_list_builder COMMA
    sub_general_statement { $3 :: $1 }

sub_general_statement_list:
| sub_general_statement_list_builder { List.rev $1 }

statement:
| general_statement { $1 }
| IF LPAREN control_initializer RPAREN LBRACE
    statement_list RBRACE { Ast.IfBlock($3,$6) }
| IF LPAREN control_initializer RPAREN LBRACE
    statement_list RBRACE ELSE LBRACE statement_list RBRACE
    { Ast.IfElseBlock($3,$6,$10)  }
| WHILE LPAREN control_initializer RPAREN LBRACE
    statement_list RBRACE { Ast.WhileBlock($3, $6) }
| FOR LPAREN sub_general_statement_list SEMI expression
    SEMI expression_comma_list RPAREN LBRACE statement_list
    RBRACE { Ast.ForBlock($3, $5, $7, $10) }
| FOR LPAREN expression TO expression BY expression RPAREN
    LBRACE statement_list RBRACE { Ast.ForByToBlock($3, $5,
    $7, $10) }
| RETURN expression SEMI { Ast.Return($2) }
| RETURN SEMI { Ast.Return(Ast.Noop) }
| BREAK SEMI { Ast.Break(1) }
| BREAK INTLITERAL SEMI { Ast.Break( Num.int_of_num $2 ) }
| CONTINUE SEMI { Ast.Continue }
| PARALLEL LPAREN parallel_binding_list RPAREN LBRACE
    statement_list RBRACE  { Ast.ParallelBlock($3, $6) }
| PARALLEL LBRACE statement_list RBRACE  { Ast.
    ParallelBlock([Ast.ThreadCount(Ast.Literal(Ast.IntLit(
    Int64.of_int(-1), Base.default_integral_bit_width)));
    Ast.Invocations(Ast.Literal(Ast.IntLit(Int64.of_int(-1),
     Base.default_integral_bit_width)))], $3) }
| ATOMIC LBRACE statement_list RBRACE { Ast.AtomicBlock($3)
     }

function_definition:
| FUN ID LPAREN binding_list RPAREN maybe_type_spec LBRACE
    statement_list RBRACE { ([$2], $4, $6, $8) }

import_definition:
| IMPORT qualified_id { $2 }
```

```
definition_list : { [] }
| definition_list import_definition { Ast.Import(Ast.
    LibraryImport($2)) :: $1 }
| definition_list function_definition { Ast.Basic(Ast.
    FunctionDefinition($2)) :: $1 }
| definition_list variable_definition SEMI { Ast.Basic(Ast.
    VariableDefinition($2)) :: $1 }
| definition_list NAMESPACE qualified_id LBRACE
    definition_list RBRACE { Ast.Namespace($3, List.rev $5)
    :: $1 }

program:
| definition_list EOF { Ast.Program(List.rev $1) }
```

../source/parser.mly

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person
     obtaining a copy of this
software and associated documentation files (the "Software"
    ), to deal in the Software
without restriction, including without limitation the
    rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
permit persons to whom the Software is furnished to do so,
    subject to the following
conditions:

The above copyright notice and this permission notice shall
     be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
```

```
(* Drives the typical lexing and parsing algorithm
while adding pertinent source, line and character information. *)

type context = {
     mutable source_name : string;
     mutable source_code : string;
  mutable original_source_code : string;
  mutable token_count : int;
     mutable token : Parser.token * Base.token_source;
}

let lex sourcename lexbuf =
  let rec acc lexbuf tokennumber tokens =
    let next_token = Scanner.token lexbuf
    and startp = Lexing.lexeme_start_p lexbuf
    and endp = Lexing.lexeme_end_p lexbuf
    in
    let line = startp.Lexing.pos_lnum
    and relpos = (1 + startp.Lexing.pos_cnum - startp.Lexing.pos_bol)
    and endrelpos = (1 + endp.Lexing.pos_cnum - endp.Lexing.pos_bol)
    and abspos = startp.Lexing.pos_cnum
    and endabspos = endp.Lexing.pos_cnum
    in
    let create_token token =
      let t = ( token, { Base.token_source_name = sourcename; Base.token_number = tokennumber;
        Base.token_line_number = line; Base.token_line_start = startp.Lexing.pos_bol;
        Base.token_column_range = (relpos, endrelpos); Base.token_character_range = (abspos, endabspos) }
      ) in
      t
    in
    match next_token with
    | Parser.EOF as token -> ( create_token token ) :: tokens
    | token -> ( create_token token ) :: ( acc lexbuf ( 1 + tokennumber ) tokens )
  in
  acc lexbuf 0 []
```

```
let parse context token_list =
  (* Keep a reference to the original token list
  And use that to dereference rather than whatever crap we get from
  the channel *)
  let tokenlist = ref(token_list) in
  let tokenizer _ = match !tokenlist with
  (* Break each token down into pieces, info and all*)
  | (token, info) :: rest ->
    context.source_name <- info.Base.token_source_name;
    context.token_count <- 1 + context.token_count;
    context.token <- ( token, info );
    (* Shift the list down by one by referencing
    the beginning of the rest of the list *)
    tokenlist := rest;
    (* return token we care about *)
    token
  (* The parser stops calling the tokenizer when
  it hits EOF: if it reaches the empty list, WE SCREWED UP *)
  | [] -> raise (Errors.MissingEoF)
  in
  (* Pass in an empty channel built off a cheap string
  and then ignore the fuck out of it in our 'tokenizer'
  internal function *)
  let program = Parser.program tokenizer (Lexing.from_string "") in
  program

let analyze program =
  (* TODO: other important checks and semantic analysis here
  that will create a proper checked program type*)
  let sem = Semant.check program in
  sem
```

../source/driver.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
```

```
merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
permit persons to whom the Software is furnished to do so,
    subject to the following
conditions:

The above copyright notice and this permission notice shall
     be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Semantic checking for the Lepix compiler that will
    produce a new
SemanticProgram type with things like locals group into a
    single type
and type promotions / conversions organized for operators.
    *)

module StringMap = Map.Make(String)

let extract_binding = function
  | Ast.VarBinding(b, _) -> b

let extract_s_binding = function
  | Semast.SVarBinding(b, _) -> b

let extract_s_binding_name = function
  | (n, _) -> n

let extract_s_binding_type = function
  | (_, tn) -> tn

let create_s_attributes () = {
  Semast.attr_parallelism = false;
```

```
  Semast.attr_arrays = 0;
  Semast.attr_strings = false;
}

let create_s_environment () = {
  Semast.env_usings = [];
  Semast.env_symbols = StringMap.empty;
  Semast.env_definitions = StringMap.empty;
  Semast.env_imports = [];
  Semast.env_loops = [];
}

let enter_block envl locals parameters =
  let acc_symbols m l =
    let ( n, tn ) = (extract_s_binding_name l, extract_s_binding_type l) in
    StringMap.add n tn m
  in
  let symbols = List.fold_left acc_symbols StringMap.empty parameters in
  let symbols = List.fold_left acc_symbols symbols locals in
  let env = {
    Semast.env_usings = [];
    Semast.env_symbols = symbols;
    Semast.env_definitions = StringMap.empty;
    Semast.env_imports = [];
    Semast.env_loops = [];
  } in
  env :: envl

let enter_parameter_block envl parameters =
  let acc_symbols m l =
    let ( n, tn ) = (extract_s_binding_name l, extract_s_binding_type l) in
    StringMap.add n tn m
  in
  let m k v1 v2 = match (v1, v2) with
    | Some(x), Some(_) -> Some(x)
    | Some(x) as s, None -> s
    | None, ( Some(y) as s ) -> s
    | _ -> None
  in
  let symbols = List.fold_left acc_symbols StringMap.empty parameters in
```

```
  let env = List.hd envl in
  let env = { env with
    Semast.env_symbols = ( StringMap.merge m symbols env.Semast.env_symbols );
  } in
  env :: envl

let lookup_id name mapl =
  let rec find = function
    | [] -> None
    | h :: tl -> try Some ( StringMap.find name h )
      with | _ -> find tl
  in
  find mapl

let env_lookup_id name envl =
  let mapl = ( List.map ( fun env -> env.Semast.env_symbols ) envl ) in
  lookup_id name mapl

let accumulate_string_type_bindings syms (n, qt) =
  let rec list_cmp v1 v2 = match v1, v2 with
    | hl::tll, hr::tlr -> hl = hr && list_cmp tll tlr
    | [], [] -> true
    | _, [] -> false
    | [], _ -> false
  in
  try
    let check_f l =
      let qt_argst = Semast.args_type_name qt in
      let pred t =
        let argst = Semast.args_type_name t in
        ( List.length argst ) = ( List.length qt_argst )
        && ( list_cmp argst qt_argst )
      in
      begin try
        let _ = List.find pred l in
        raise(Errors.FunctionAlreadyExists("an id with name " ^ n ^ " and type " ^ ( Representation.string_of_s_type_name qt ) ^ " is already present"))
      with
        Not_found -> Semast.SOverloads( qt :: l )
      end
    in
    let v = StringMap.find n syms in
```

```
      let vt = match v with
        | Semast.SOverloads(tl) -> check_f tl
        | Semast.SFunction(_,_,_) as t -> check_f [t]
        | _ -> raise(Not_found)
      in
      StringMap.add n vt syms
    with Not_found ->
      StringMap.add n qt syms

let import_builtin_module symbols = function
  | Semast.Lib -> begin
      let c_bindings = [
        ("lib.print", Semast.SFunction( Semast.void_t, [Semast.int32_t], Semast.no_qualifiers ) );
        ("lib.print", Semast.SFunction( Semast.void_t, [Semast.string_t], Semast.no_qualifiers ) );
        ("lib.print", Semast.SFunction( Semast.void_t, [Semast.float64_t], Semast.no_qualifiers ) );
        ("lib.print_n", Semast.SFunction( Semast.void_t, [Semast.int32_t], Semast.no_qualifiers ) );
        ("lib.print_n", Semast.SFunction( Semast.void_t, [Semast.string_t], Semast.no_qualifiers ) );
        ("lib.print_n", Semast.SFunction( Semast.void_t, [Semast.float64_t], Semast.no_qualifiers ) );
      ]
      in
      print_endline "Here!";
      let symbols = List.fold_left accumulate_string_type_bindings symbols c_bindings in
      (symbols)
    end

let rec type_name_of_ast_type_name = function
  | Ast.BuiltinType(bt, tq) -> Semast.SBuiltinType( bt, tq )
  | Ast.Array(tn, d, tq) -> Semast.SArray( ( type_name_of_ast_type_name tn ), d, tq )
  | Ast.SizedArray(tn, d, dims, tq) -> Semast.SSizedArray( ( type_name_of_ast_type_name tn ), d, dims, tq )
  | Ast.Function( tn, pl, tq) -> Semast.SFunction( ( type_name_of_ast_type_name tn ), ( List.map type_name_of_ast_type_name pl ), tq )

let type_name_of_ast_literal attrs envl astlit =
  let t = match astlit with
```

```
    | Ast.BoolLit(_) -> Ast.BuiltinType( Ast.Bool, Ast.no_qualifiers )
    | Ast.IntLit(_, b) -> Ast.BuiltinType( Ast.Int(b), Ast.no_qualifiers )
    | Ast.FloatLit(_, b) -> Ast.BuiltinType( Ast.Float(b), Ast.no_qualifiers )
    | Ast.StringLit(_) -> Ast.BuiltinType( Ast.String, Ast.no_qualifiers )
  in
  type_name_of_ast_type_name t

let check_binary_op_common_type lt bop rt = match (lt, rt) with
  | (Semast.SBuiltinType(Ast.Int(n), ltq), Semast.SBuiltinType(Ast.Int(m), rtq)) -> Semast.SBuiltinType(Ast.Int(max n m), Semast.no_qualifiers)
  | (Semast.SBuiltinType(Ast.Float(n), ltq), Semast.SBuiltinType(Ast.Int(m), rtq)) -> Semast.SBuiltinType(Ast.Float(max n m), Semast.no_qualifiers)
  | (Semast.SBuiltinType(Ast.Int(n), ltq), Semast.SBuiltinType(Ast.Float(m), rtq)) -> Semast.SBuiltinType(Ast.Float(max n m), Semast.no_qualifiers)
  | (Semast.SBuiltinType(Ast.Float(n), ltq), Semast.SBuiltinType(Ast.Float(m), rtq)) -> Semast.SBuiltinType(Ast.Float(max n m), Semast.no_qualifiers)
  | (Semast.SBuiltinType(Ast.Bool, ltq), Semast.SBuiltinType(Ast.Bool, rtq)) -> Semast.SBuiltinType(Ast.Bool, Semast.no_qualifiers)
  | _ -> raise(Errors.InvalidBinaryOperation("cannot perform a binary operation on two non-numeric types"))

let check_unary_op uop sr = match sr with
  | Semast.SBuiltinType(Ast.Int(n), ltq) -> Semast.SBuiltinType(Ast.Int(n), Semast.no_qualifiers)
  | Semast.SBuiltinType(Ast.Float(n), ltq) -> Semast.SBuiltinType(Ast.Float(n), Semast.no_qualifiers)
  | _ -> raise(Errors.InvalidUnaryOperation("cannot perform a unary operation on this type"))

let overload_resolution args = function
  | Semast.SFunction(__, tnl, _) -> let argslen = ( List.length args ) in
    let overloadlen = ( List.length tnl ) in
    argslen = overloadlen &&
    if ( argslen > 0 ) then ( List.exists ( fun a -> List.
```

```
    mem a tnl ) args )
      else true
    | _ -> raise(Errors.TypeMismatch("cannot resolve an overload that includes a non-function in its type listing"))

let check_function_overloads args overloadlist =
  begin try
    let ft = List.find ( overload_resolution args ) overloadlist
    in
    ( ft , Semast.return_type_name ft )
  with _ ->
    let argslist = "( " ^ ( String.concat "," ( List.map Representation.string_of_s_type_name args ) ) ^ " )" in
    raise(Errors.BadFunctionCall("could not resolve the specific overload for this set of " ^ ( Representation.string_of_s_type_name ( Semast.SOverloads(overloadlist) ) ) ^ " using " ^ argslist ))
  end

let rec type_name_of_ast_expression attrs envl astexpr =
  let t = match astexpr with
    | Ast.Literal(lit) -> type_name_of_ast_literal attrs envl lit
    | Ast.QualifiedId(qid)-> let qualname = Semast.string_of_qualified_id qid in
      begin match ( env_lookup_id qualname envl ) with
        | None -> raise(Errors.IdentifierNotFound("identifier '" ^ qualname ^ "' not found"))
        | Some(stn) -> stn
      end
    | Ast.Call(e, args) -> let ft = type_name_of_ast_expression attrs envl e in
      (* TODO: check arguments to make sure it matches *)
      begin match ft with
        | Semast.SFunction(rt, pl, tq) ->
          rt
        | Semast.SOverloads(fl) ->
          let sargs = List.map ( type_name_of_ast_expression attrs envl ) args in
          let (ft, r) = check_function_overloads sargs fl in
          r
        | _ -> raise(Errors.TypeMismatch("expected a
```

```
    function type, but received something else."))
      end
    | Ast.Noop -> Semast.void_t
    | Ast.BinaryOp(l, bop, r) ->
      let sl = type_name_of_ast_expression attrs envl l
      and sr = type_name_of_ast_expression attrs envl r
      in
      check_binary_op_common_type sl bop sr
    | Ast.PrefixUnaryOp(uop, r) ->
      let sr = type_name_of_ast_expression attrs envl r in
      check_unary_op uop sr
    | Ast.Assignment(lhs, rhs) -> let lhst =
    type_name_of_ast_expression attrs envl lhs in
      lhst
    | Ast.Member(_, _) -> raise(Errors.Unsupported("member
    access is not supported"))
    | _ -> raise(Errors.Unsupported("expression conversion
    currently unsupported"))
  in
  (* TODO: some type checks to make sure weird things like
    void& aren't put in place... *)
  t

let process_ast_import prefix symbols defs imports =
    function
  | Ast.LibraryImport(qid) ->
    let qualname = Semast.string_of_qualified_id qid in
    let (v, impsymbols) = match List.filter ( fun (n, _) ->
     n = qualname ) Semast.builtin_library_names with
      | (_, bltin) :: [] -> let b = Semast.SBuiltin(bltin)
    in
        let ( bltinsymbols ) = import_builtin_module
    symbols bltin in
        ( b, bltinsymbols )
      | _ -> ( Semast.SDynamic(qualname), symbols )
    in
    ( prefix, impsymbols, defs, v :: imports )


let generate_global_env = function
| Ast.Program(ast_definitions) ->
  let rec acc ( prefix, symbols, defs, imports ) def =
    match def with
      | Ast.Import(imp) -> process_ast_import prefix
    symbols defs imports imp
```

```
      | Ast.Basic(Ast.FunctionDefinition((qid, args, rt, _))) ->
        let argst = List.map Ast.binding_type args in
        let qualname = prefix ^ Semast.string_of_qualified_id qid in
        let qt = Ast.Function(rt, argst, Ast.no_qualifiers) in
        let sqt = ( type_name_of_ast_type_name qt ) in
        print_endline "here";
        let nsymbols = accumulate_string_type_bindings symbols (qualname, sqt)
        and ndefs = accumulate_string_type_bindings defs (qualname, sqt)
        in
        ( prefix, nsymbols, ndefs, imports )
      | Ast.Basic(Ast.VariableDefinition(v)) ->
        let (name, qt) = extract_binding v in
        let qualname = prefix ^ name in
        if StringMap.mem prefix symbols then raise (Errors.VariableAlreadyExists(qualname)) else
        let nsymbols = ( StringMap.add qualname ( type_name_of_ast_type_name qt ) symbols )
        and ndefs = ( StringMap.add qualname ( type_name_of_ast_type_name qt ) defs )
        in
        ( prefix, nsymbols, ndefs, imports )
      | Ast.Namespace(n, dl) ->
        let qualname = prefix ^ Semast.string_of_qualified_id n in
        let (_, innersymbols, innerdefs, innerimports ) = List.fold_left acc ( qualname ^ ".", symbols, defs, imports ) dl in
        ( prefix, innersymbols, innerdefs, innerimports )
  in
  let (_, symbols, defs, imports) = List.fold_left acc ("", StringMap.empty, StringMap.empty, []) ast_definitions in
  let attrs = create_s_attributes () in
  let env = {
    Semast.env_usings = [];
    Semast.env_symbols = symbols;
    Semast.env_definitions = defs;
    Semast.env_imports = imports;
    Semast.env_loops = [];
  }
```

```
  in
  ( attrs , env )

let check_qualified_identifier attrs envl sl t =
  (attrs , envl , Semast.SQualifiedId(sl , t))

let check_function_call attrs envl target args =
  let (t , rt) = match Semast.type_name_of_s_expression target with
    | Semast.SFunction(tn , tnl , tq) as f -> f , tn
    | Semast.SOverloads(fl) ->
      let args_t = ( List.map Semast.type_name_of_s_expression args ) in
      check_function_overloads args_t fl
    | _ -> raise(Errors.BadFunctionCall("cannot invoke an expression which does not result in a function type of some sort"))
  in
  (attrs , envl , Semast.SCall(( Semast.coerce_type_name_of_s_expression t target ), args , rt ))

let generate_s_binding prefix attrs envl = function
  | (name, tn) -> (attrs , envl , (Semast.string_of_qualified_id ( prefix @ [name] ), type_name_of_ast_type_name tn))

let gather_ast_locals attrs envl sl pl =
  let acc locals = function
    | Ast.General(Ast.VariableStatement(v)) ->
      let (_, _, sb) = generate_s_binding [] attrs envl ( extract_binding v ) in
      sb :: locals
    | _ -> locals
  in
  let l = List.rev( List.fold_left acc [] sl ) in
  if ( List.length l ) > 0 then begin
    let envl = ( enter_block envl l pl ) in
    (true , attrs , envl , l)
  end else
    let envl = ( enter_parameter_block envl pl ) in
    (false , attrs , envl , l)

let generate_s_literal attrs envl = function
  | Ast.BoolLit(b) -> (attrs , envl , Semast.SBoolLit(b))
  | Ast.IntLit(i , b) -> (attrs , envl , Semast.SIntLit(i , b))
```

```
  | Ast.FloatLit(f, b) -> (attrs, envl, Semast.SFloatLit(f, b))
  | Ast.StringLit(s) -> (attrs, envl, Semast.SStringLit(s))

let rec generate_s_expression attrs envl astexpr =
  let acc_s_expression (attrs, envl, sel) e =
    let (attrs, envl, se) = generate_s_expression attrs envl e in
    (attrs, envl, se :: sel)
  in
  let ( attrs, envl, se ) = match astexpr with
    | Ast.Literal(lit) ->
      let ( attrs, envl, slit ) = generate_s_literal attrs envl lit in
      ( attrs, envl, Semast.SLiteral( slit ) )
    | Ast.QualifiedId(sl) ->
      let t = type_name_of_ast_expression attrs envl astexpr in
      check_qualified_identifier attrs envl sl t
    | Ast.Call(e, el) ->
      let ( attrs, envl, target ) = ( generate_s_expression attrs envl e ) in
      let ( attrs, envl, args ) = ( List.fold_left acc_s_expression (attrs, envl, []) el ) in
      let args = List.rev args in
      check_function_call attrs envl target args
    | Ast.BinaryOp(lhs, bop, rhs) ->
      let ( attrs, envl, slhs ) = ( generate_s_expression attrs envl lhs ) in
      let ( attrs, envl, srhs ) = ( generate_s_expression attrs envl rhs ) in
      let rhst = ( Semast.type_name_of_s_expression srhs ) in
      let lhst = ( Semast.type_name_of_s_expression slhs ) in
      ( attrs, envl, Semast.SBinaryOp( slhs, bop, srhs, ( check_binary_op_common_type lhst bop rhst ) ) )
    | Ast.PrefixUnaryOp(uop, rhs) ->
      let ( attrs, envl, srhs ) = ( generate_s_expression attrs envl rhs ) in
      let rhst = ( Semast.type_name_of_s_expression srhs ) in
      ( attrs, envl, Semast.SPrefixUnaryOp( uop, srhs, ( check_unary_op uop rhst ) ) )
    | Ast.Assignment(lhs, rhs) ->
```

```
      let ( attrs, envl, slhs ) = ( generate_s_expression attrs envl lhs ) in
      let ( attrs, envl, srhs ) = ( generate_s_expression attrs envl rhs ) in
      let lhst = ( Semast.type_name_of_s_expression slhs ) in
      ( attrs, envl, Semast.SAssignment( slhs, srhs, lhst ) )
    | Ast.Noop -> ( attrs, envl, Semast.SNoop )
    | _ -> raise(Errors.Unsupported("expression generation for this type is current unsupported"))
  in
  let t = Semast.type_name_of_s_expression se in
  let attrs = match t with
    | Semast.SArray(_,d,_)
    | Semast.SSizedArray(_,d,_,_) -> {
        Semast.attr_strings = attrs.Semast.attr_strings;
        Semast.attr_arrays = max d attrs.Semast.attr_arrays;
        Semast.attr_parallelism = attrs.Semast.attr_parallelism;
      }
    | Semast.SBuiltinType( Ast.String, _ ) -> {
        Semast.attr_strings = true;
        Semast.attr_arrays = attrs.Semast.attr_arrays;
        Semast.attr_parallelism = attrs.Semast.attr_parallelism;
      }
    | _ -> attrs
  in
  ( attrs, envl, se )

let generate_s_variable_definition prefix attrs envl = function
  | Ast.VarBinding(b, e) ->
    let (attrs, envl, sb) = generate_s_binding prefix attrs envl b in
    let (attrs, envl, se) = generate_s_expression attrs envl e in
    (attrs, envl, Semast.SVarBinding(sb, se))

let generate_s_general_statement attrs envl = function
  | Ast.ExpressionStatement(e) ->
    let (attrs, envl, se ) = generate_s_expression attrs envl e in
```

```
      ( attrs, envl, Semast.SExpressionStatement( se ) )
    | Ast.VariableStatement(v) ->
      let (attrs, envl, sv ) = generate_s_variable_definition [] attrs envl v in
      ( attrs, envl, Semast.SVariableStatement( sv ) )

let generate_s_statement attrs envl = function
    | Ast.General(g) ->
      let ( attrs, envl, sgs ) = generate_s_general_statement attrs envl g in
      ( attrs, envl, Semast.SGeneral( sgs ) )
    | Ast.Return(e) ->
      let ( attrs, envl, se ) = generate_s_expression attrs envl e in
      ( attrs, envl, Semast.SReturn( se ) )
    | _ -> raise(Errors.Unsupported("statement type not supported"))

let check_returns name ssl rt =
    (* Todo: recursively inspect all inner blocks for return types as well *)
    let acc rl = function
      | Semast.SReturn(se) -> ( Semast.type_name_of_s_expression se ) :: rl
      | _ -> rl
    in
    let returns = List.fold_left acc [] ssl in
    let returnlength = List.length returns in
    if name = "main" then begin
      let sret0 = Semast.SReturn(Semast.SLiteral(Semast.SIntLit(Int64.zero, 32))) in
      let mainpred = function
        | Semast.SBuiltinType(Ast.Auto, _) ->
          ()
        | Semast.SBuiltinType(Ast.Int(32), (_, r)) ->
          if r then raise(Errors.InvalidMainSignature("Cannot return a reference from main"));
          ()
        | _ ->
          raise(Errors.InvalidMainSignature("You can only return an int from main"))
      in
      let ssl = if returnlength < 1 then begin ssl @ [sret0] end else ssl
      in
```

```
      let ssl = match rt with
        | Semast.SBuiltinType(Ast.Int(32), (_, r)) ->
          if r then raise(Errors.InvalidMainSignature("Cannot return a reference to and integer"));
          let _ = List.iter mainpred returns in
          ssl
        | _ -> let _ = List.iter mainpred returns in
          ssl
      in
      (ssl, Semast.int32_t)
    end
    else begin
      let generalpred rt r = match rt with
        | Semast.SBuiltinType(Ast.Auto, _) -> r
        | _ -> let r = Semast.unqualify r in
          let urt = Semast.unqualify rt in
          if r <> urt then raise(Errors.InvalidFunctionSignature("return types do not match across all returns", name))
          else rt
      in
      let rt = List.fold_left generalpred rt returns in
      let (ssl, rt) = match rt with
        | Semast.SBuiltinType(Ast.Auto, _) ->
          if returnlength < 1 then
            ( ssl @ [Semast.SReturn(Semast.SNoop)], Semast.void_t )
          else
            (ssl, rt)
        | Semast.SBuiltinType(Ast.Void, _) ->
          if returnlength < 1 then
            ( ssl @ [Semast.SReturn(Semast.SNoop)], Semast.void_t )
          else
            (ssl, rt)
        | _ ->
          if returnlength < 1 then
            raise(Errors.InvalidFunctionSignature("function was expected to return a value: returned no value", name))
          else
            (ssl, rt)
      in
      (ssl, rt)
    end
```

```
let generate_s_function_definition prefix attrs envl astfdef =
  let acc_ast_statements (attrs, envl, ssl) s =
    let ( attrs, envl, ss ) = ( generate_s_statement attrs envl s ) in
    ( attrs, envl, ss :: ssl )
  in
  let acc_ast_parameters (attrs, envl, pl) p =
    let (attrs, envl, sp) = generate_s_binding [] attrs envl p in
    (attrs, envl, sp :: pl)
  in
  let (qid, astparameters, astrt, body) = astfdef in
  let fqid = prefix @ qid in
  let fqn = Semast.string_of_qualified_id fqid in
  let (attrs, envl, parameters) = List.fold_left acc_ast_parameters (attrs, envl, []) astparameters in
  let rt = type_name_of_ast_type_name astrt in
  let (has_locals, attrs, envl, bl) = gather_ast_locals attrs envl body parameters in
  let (attrs, envl, ssl) = List.fold_left acc_ast_statements (attrs, envl, []) body in
  let ssl = List.rev ssl in
  let (ssl, rt) = check_returns fqn ssl rt in
  let sfuncdef = if has_locals then
  {
    Semast.func_name = fqid;
    Semast.func_parameters = Semast.SParameters(parameters);
    Semast.func_return_type = rt;
    Semast.func_body = [Semast.SBlock(Semast.SLocals(bl), ssl)];
  }
  else
  {
    Semast.func_name = fqid;
    Semast.func_parameters = Semast.SParameters(parameters);
    Semast.func_return_type = rt;
    Semast.func_body = ssl;
  }
  in
  ( attrs, envl, Semast.SFunctionDefinition( sfuncdef ) )
```

```

let generate_s_basic_definition prefix attrs envl = function
  | Ast.FunctionDefinition(fdef) ->
    let (attrs, envl, sfdef) = generate_s_function_definition prefix attrs envl fdef in
    (attrs, envl, Semast.SBasic(sfdef))
  | Ast.VariableDefinition(vdef) ->
    let (attrs, envl, svdef) = generate_s_variable_definition prefix attrs envl vdef in
    (attrs, envl, Semast.SBasic(Semast.SVariableDefinition(svdef)))


let define_libraries attrs env =
  let fi = Semast.SFunction(Semast.void_t, [Semast.int32_t], Semast.no_qualifiers) in
  let ff = Semast.SFunction(Semast.void_t, [Semast.float64_t], Semast.no_qualifiers) in
  let fs = Semast.SFunction(Semast.void_t, [Semast.string_t], Semast.no_qualifiers) in
  let fo = Semast.SOverloads([fi;ff;fs]) in
  let lib_printn_defint = {
    Semast.func_name = ["lib"; "print_n"];
    Semast.func_parameters = Semast.SParameters([("i", Semast.int32_t)]);
    Semast.func_return_type = Semast.void_t;
    Semast.func_body = [
      Semast.SGeneral(Semast.SExpressionStatement(
        Semast.SCall(Semast.SQualifiedId(["lib"; "print"], fo), [Semast.SQualifiedId(["i"], Semast.int32_t)], Semast.void_t)
      ));
      Semast.SGeneral(Semast.SExpressionStatement(
        Semast.SCall(Semast.SQualifiedId(["lib"; "print"], fo), [Semast.SLiteral(Semast.SStringLit("\n"))], Semast.void_t)
      ));
      Semast.SReturn(Semast.SNoop);
    ];
  }
  and lib_printn_deffloat = {
    Semast.func_name = ["lib"; "print_n"];
    Semast.func_parameters = Semast.SParameters([("i", Semast.float64_t)]);
```

```
    Semast.func_return_type = Semast.void_t;
    Semast.func_body = [
      Semast.SGeneral(Semast.SExpressionStatement(
        Semast.SCall(Semast.SQualifiedId(["lib"; "print"], fo), [Semast.SQualifiedId(["i"], Semast.float64_t)], Semast.void_t)
      ));
      Semast.SGeneral(Semast.SExpressionStatement(
        Semast.SCall(Semast.SQualifiedId(["lib"; "print"], fo), [Semast.SLiteral(Semast.SStringLit("\n"))], Semast.void_t)
      ));
      Semast.SReturn(Semast.SNoop);
    ];
  }
  and lib_printn_defstr = {
    Semast.func_name = ["lib"; "print_n"];
    Semast.func_parameters = Semast.SParameters([("i", Semast.string_t)]);
    Semast.func_return_type = Semast.void_t;
    Semast.func_body = [
      Semast.SGeneral(Semast.SExpressionStatement(
        Semast.SCall(Semast.SQualifiedId(["lib"; "print"], fo), [Semast.SQualifiedId(["i"], Semast.string_t)], Semast.void_t)
      ));
      Semast.SGeneral(Semast.SExpressionStatement(
        Semast.SCall(Semast.SQualifiedId(["lib"; "print"], fo), [Semast.SLiteral(Semast.SStringLit("\n"))], Semast.void_t)
      ));
      Semast.SReturn(Semast.SNoop);
    ];
  }
  in
  let libdefs = [
    Semast.SBasic(Semast.SFunctionDefinition(lib_printn_defstr));
    Semast.SBasic(Semast.SFunctionDefinition(lib_printn_defint));
    Semast.SBasic(Semast.SFunctionDefinition(lib_printn_deffloat));
  ] in
  let acc defs = function
    | Semast.SBasic(Semast.SFunctionDefinition(fdef)) ->
```

```
      let n = (Semast.string_of_qualified_id fdef.Semast.func_name )
      and qt = ( Semast.type_name_of_s_function_definition fdef )
      in
      if ( StringMap.mem n defs ) then defs else accumulate_string_type_bindings defs (n, qt)
    | Semast.SBasic(Semast.SVariableDefinition(Semast.SVarBinding((n, tn), _))) ->
      ( StringMap.add n tn defs )
  in
  let ndefs = List.fold_left acc env.Semast.env_definitions libdefs in
  ( { env with Semast.env_definitions = ndefs; }, libdefs)

let direct_code_inject attrs globalenv imp sdl =
  let s = match imp with
    | Ast.LibraryImport(qid) -> Semast.string_of_qualified_id qid
  in
  match s with
    | "lib" -> let (globalenv, library_defs) = define_libraries attrs globalenv in
      (globalenv, library_defs @ sdl)
    | _ -> (globalenv, sdl)

let generate_semantic attrs globalenv = function
| Ast.Program(dl) ->
  let envl = [globalenv] in
  let rec acc_ast_definitions (prefix, attrs, envl, sdl) = function
    | Ast.Import(imp) ->
      let globalenv = List.hd ( List.rev envl ) in
      let (globalenv, sdl) = direct_code_inject attrs globalenv imp sdl in
      let tail = List.tl ( List.rev envl ) in
      (prefix, attrs, ( globalenv :: tail ), sdl)
    | Ast.Namespace(n, dl) -> let qualname = prefix @ n in
      let (_, attrs, envl, nssdl) = List.fold_left acc_ast_definitions (qualname, attrs, envl, sdl) dl in
      ( prefix, attrs, envl, nssdl )
    | Ast.Basic(b) ->
      let (attrs, envl, sb) = ( generate_s_basic_definition prefix attrs envl b ) in
      ( prefix, attrs, envl, sb :: sdl )
```

```
  in
  let (_, attrs, envl, sdefs) = List.fold_left acc_ast_definitions ([], attrs, envl, []) dl in
  let globalenv = List.hd ( List.rev envl ) in
  Semast.SProgram( attrs, globalenv, List.rev sdefs )

let modify_symbols = function
| Semast.SProgram(attrs, env, sdls) ->
  let rec acc ( symbols, defs ) def =
    match def with
      | Semast.SBasic(Semast.SFunctionDefinition(f)) ->
        let qualname = Semast.string_of_qualified_id f.Semast.func_name in
        let sqt = ( Semast.type_name_of_s_function_definition f ) in
        let nsymbols = accumulate_string_type_bindings symbols (qualname, sqt)
        and ndefs = accumulate_string_type_bindings defs (qualname, sqt)
        in
        ( nsymbols, ndefs )
      | Semast.SBasic(Semast.SVariableDefinition(v)) ->
        ( symbols, defs )
  in
  let (symbols, defs) = List.fold_left acc (StringMap.empty, StringMap.empty) sdls in
  let m k l r = match (l, r) with
    | Some(l), Some(_) -> Some(l)
    | None, ( Some(r) as s ) -> s
    | ( Some(l) as s ), None -> s
    | _ -> None
  in
  let env = { env with
    Semast.env_symbols = StringMap.merge m symbols env.Semast.env_symbols;
    Semast.env_definitions = defs;
  }
  in
  Semast.SProgram( attrs, env, sdls )

let check astprogram =
  (* Pass 1: Gather globals inside of all the namespaces
  so they can be referenced even before they're defined ( just so long as
  they're in the same lateral global scope, not necessarily
```

```
  in vertical order) *)
  let ( attrs, env ) = generate_global_env astprogram in
  (* Pass 2: Generate the actual Semantic Tree based on what
  is inside the AST program... *)
  print_endline "Pass 2";
  let sprog = generate_semantic attrs env astprogram in
  (* Pass 3: Update any symbols that were resolved during
  bottom-up type derivation... *)
  print_endline "Pass 3";
  let Semast.SProgram(attrs, env, _) = modify_symbols sprog in
  (* Pass 4: Finalize everything with new information *)
  print_endline "Pass 4";
  let sprog = generate_semantic attrs env astprogram in
  sprog
```

../source/semant.ml

```
(* LePiX Language Compiler Implementation
Copyright (c) 2016- ThePhD

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify,
merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be included in all copies
or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
```

```
CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)

(* Code generation: translate takes a semantically checked AST and produces
LLVM IR:
http://llvm.org/docs/tutorial/index.html
http://llvm.moe/ocaml/ *)

(* Linked code after the c bindings from the makefile
compiled the ll for the c bindings *)

module StringMap = Map.Make(String)

type li_universe = {
  lu_attrs : Semast.s_attributes;
  lu_env : Semast.s_environment;
  lu_module : Llvm.llmodule;
  lu_context : Llvm.llcontext;
  lu_builder : Llvm.llbuilder;
  lu_variables : Llvm.llvalue StringMap.t;
  lu_functions : Llvm.llvalue StringMap.t;
  lu_named_values : Llvm.llvalue StringMap.t;
  lu_named_params : Llvm.llvalue StringMap.t;
  lu_handlers : ( li_universe -> ( Semast.s_expression list ) -> ( li_universe * Llvm.llvalue list ) ) StringMap.t;
}

let create_li_universe = function | Semast.SProgram(attrs, env, _) ->
  let context = Llvm.global_context() in
  let builder = Llvm.builder context in
  let m = Llvm.create_module context "lepix" in
  {
    lu_attrs = attrs;
    lu_env = env;
    lu_module = m;
    lu_context = context;
    lu_builder = builder;
    lu_variables = StringMap.empty;
    lu_functions = StringMap.empty;
    lu_named_values = StringMap.empty;
    lu_named_params = StringMap.empty;
    lu_handlers = StringMap.empty;
  }
```

```
let rec llvm_type_of_s_type_name lu st =
  let f32_t   = Llvm.float_type   lu.lu_context
  and f64_t   = Llvm.double_type  lu.lu_context
  (* for 'char' type to printf -- even if they resolve to same type, we differentiate*)
  and char_t  = Llvm.i8_type      lu.lu_context
  and i16_t   = Llvm.i16_type     lu.lu_context
  and i32_t   = Llvm.i32_type     lu.lu_context
  and i64_t   = Llvm.i64_type        lu.lu_context
  (* LLVM treats booleans as 1-bit integers, not distinct types with their own true / false *)
  and bool_t  = Llvm.i1_type      lu.lu_context
  and void_t  = Llvm.void_type    lu.lu_context
  in
  let p_char_t  = Llvm.pointer_type    char_t
  in
  match st with
  (* TODO: handle reference-ness (e.g., make it behave like a pointer here) *)
  | Semast.SBuiltinType( Ast.Bool, tq ) -> bool_t
  | Semast.SBuiltinType( Ast.Int(n), tq ) -> begin match n with
    | 64 -> i64_t
    | 32 -> i32_t
    | 16 -> i16_t
    | _ -> Llvm.integer_type lu.lu_context n
  end
  | Semast.SBuiltinType( Ast.Float(n), tq ) -> begin match n with
    | 64 -> f64_t
    | 32 -> f32_t
    | 16 -> (* LLVM actually has support for this, but shitty OCaml bindings *)
      (* TODO: Proper Error *)
      raise( Failure "Cannot have a Half Float because OCaml binding for LLVM is garbage" )
    | _ -> (* TODO: Proper Error *)
      raise( Failure "Unallowed Float Width" )
  end
  | Semast.SBuiltinType( Ast.String, tq ) -> p_char_t
  | Semast.SBuiltinType( Ast.Void, tq ) -> void_t
  | Semast.SArray(t, d, tq) -> Llvm.array_type ( llvm_type_of_s_type_name lu t) d
  | Semast.SSizedArray(t, d, szs, tq) -> Llvm.array_type (
```

```
      llvm_type_of_s_type_name lu t) d
    | Semast.SFunction(rt, argst, tq) ->
      let lrt = llvm_type_of_s_type_name lu rt
      and largst = Array.map ( llvm_type_of_s_type_name lu ) ( Array.of_list argst )
      in
      Llvm.function_type lrt largst
    | _ -> (* TODO: Proper Error *)
      raise(Errors.Unsupported("This type is not convertible to an LLVM type"))

let should_reference_pointer = function
    | Semast.SBuiltinType(Ast.String, _) -> true
    | Semast.SArray(_, _, _) -> true
    | Semast.SSizedArray(_, _, _, _) -> true
    | Semast.SFunction(_, _, _) -> true
    | _ -> false

let find_argument_handler lu target =
  let hn = Llvm.value_name target in
  try Some( StringMap.find hn lu.lu_handlers )
  with _ -> None

let llvm_lookup_function lu name t =
  let mname = Semast.mangle_name [name] t in
  match Llvm.lookup_function mname lu.lu_module with
    | Some(v) -> v
    | None -> raise( Errors.FunctionLookupFailure( name, mname ) )

let llvm_lookup_variable lu name t =
  match Llvm.lookup_global name lu.lu_module with
    | Some(v) -> v
    | None -> raise( Errors.VariableLookupFailure( name, name ) )

let dump_s_qualified_id lu qid t =
  let fqn = Semast.string_of_qualified_id qid in
  let lookup n =
    try
      let v = StringMap.find n lu.lu_named_values in
      Some(v)
    with | Not_found ->
      try
        let v = StringMap.find n lu.lu_named_params in
```

```
        Some(v)
    with | Not_found -> try
        let v = StringMap.find n lu.lu_variables in
        Some(v)
    with | Not_found -> None
  in
  let lookup_func n =
    try
      let v = StringMap.find n lu.lu_named_values in
      Some(v)
    with | Not_found ->
      try
        let v = StringMap.find n lu.lu_named_params in
        Some(v)
    with | Not_found -> try
        let v = StringMap.find n lu.lu_functions in
        Some(v)
    with | Not_found -> None
  in
  let overload_lookup qid ft =
    let mangled = Semast.mangle_name qid ft in
    begin match lookup_func mangled with
      | Some(v) as s -> s
      | None -> begin match lookup_func fqn with
        | Some(v) as s -> s
        | None -> None
        end
    end
  in
  let overload_acc op ft =
    match op with
      | Some(v) as s -> s
      | None -> overload_lookup qid ft
  in
  let idval = match t with
    | Semast.SFunction(rt, tnl, tq) as ft ->
      begin match overload_lookup qid ft with
        | Some(v) -> v
        | None -> raise (Errors.UnknownFunction(fqn))
      end
    | Semast.SOverloads(fl)->
      begin match List.fold_left overload_acc None fl with
        | Some(v) -> v
        | None -> raise(Errors.UnknownFunction(fqn))
      end
```

```
    | _ -> match lookup fqn with
        | Some(v) -> v
        | None -> raise (Errors.UnknownVariable fqn)
  in
  (lu, idval)

let dump_s_literal lu lit =
  let f64_t   = Llvm.double_type   lu.lu_context
  and bool_t  = Llvm.i1_type       lu.lu_context
  in
  let v = match lit with
    | Semast.SBoolLit(value) -> Llvm.const_int bool_t (if value then 1 else 0) (* bool_t is still an integer, must convert *)
    | Semast.SIntLit(value, b) -> Llvm.const_of_int64 (Llvm.integer_type lu.lu_context b) value true (* bool for signedness *)
    | Semast.SStringLit(value) ->
      let str = Llvm.build_global_stringptr value "str_lit" lu.lu_builder in
      str
    | Semast.SFloatLit(value, b) -> Llvm.const_float f64_t value
  in
  (lu, v)

let dump_temporary_value lu ev v =
  let v = match ( Llvm.classify_type ( Llvm.type_of v ) ) with
    | Llvm.TypeKind.Pointer ->
      if ( should_reference_pointer ev ) then
        v
      else
        Llvm.build_load v "tmp" lu.lu_builder
    | _ -> v
  in
  v

let dump_s_expression_temporary_gen f lu e =
  let (lu, v) = ( f lu e ) in
  let v = dump_temporary_value lu ( Semast.type_name_of_s_expression e ) v in
  (lu, v)

let dump_arguments_gen f lu el =
```

```
  let acc_expr (lu, vl) e =
    let (lu, v) = dump_s_expression_temporary_gen f lu e in
    ( lu, v :: vl )
  in
  let (lu, args) = List.fold_left acc_expr (lu, []) el in
  (lu, args)

let rec dump_s_expression lu e =
  match e with
  | Semast.SLiteral(lit) -> dump_s_literal lu lit
  | Semast.SQualifiedId(qid, t) ->
    let (lu, v) = dump_s_qualified_id lu qid t in
    (lu, v)
  | Semast.SCall(e, el, t) ->
    let (lu, target) = dump_s_expression lu e in
    let oparghandler = find_argument_handler lu target in
    let ( lu, args ) = match oparghandler with
      | None ->
        let ( lu, args ) = ( dump_arguments_gen ( dump_s_expression ) lu el ) in
        (lu, List.rev args)
      | Some(h) ->
        let (lu, args) = ( h lu el ) in
        ( lu, args )
    in
    let arr_args = Array.of_list args in
    let v = match t with
      | Semast.SBuiltinType(Ast.Void, _) -> Llvm.build_call target arr_args "" lu.lu_builder
      | _ -> Llvm.build_call target arr_args "tmp.call" lu.lu_builder
    in
    (lu, v)
  | Semast.SBinaryOp(l, bop, r, t) ->
    let (lu, lv) = dump_s_expression lu l in
    let (lu, rv) = dump_s_expression lu r in
    let opf = match bop with
      | Ast.Add -> Llvm.build_add
      | Ast.Sub -> Llvm.build_sub
      | Ast.Mult -> Llvm.build_mul
      | Ast.Div -> Llvm.build_sdiv
      | _ -> raise(Errors.Unsupported("This binary operation type is not supported for code generation"))
    in
    let v = opf lv rv "tmp.bop" lu.lu_builder in
```

```
    (lu, v)
  | _ -> raise(Errors.Unsupported("This expression is not supported for code generation"))

let dump_s_expression_temporary lu e =
  let (lu, v) = ( dump_s_expression lu e ) in
  let v = dump_temporary_value lu ( Semast.type_name_of_s_expression e ) v in
  (lu, v)

let dump_s_locals lu locals =
  let acc lu (n, tn) =
    let lty = llvm_type_of_s_type_name lu tn in
    let v = Llvm.build_alloca lty n lu.lu_builder in
    { lu with lu_named_values = StringMap.add (n) v lu.lu_named_values }
  in
  let Semast.SLocals(bl) = locals in
  let lu = List.fold_left acc lu bl in
  lu

let dump_s_parameters lu llfunc parameters =
  let paramarr = Llvm.params llfunc in
  let paraml = Array.to_list paramarr in
  let Semast.SParameters(bl) = parameters in
  let nameparam i p =
    let (n, _) = ( List.nth bl i ) in
    ( Llvm.set_value_name n p )
  in
  let _ = Array.iteri nameparam paramarr in
  let acc lu (n, tn) =
    let v = List.find ( fun p -> ( ( Llvm.value_name p ) = n ) ) paraml in
    { lu with lu_named_params = StringMap.add n v lu.lu_named_params }
  in
  let lu = List.fold_left acc lu bl in
  lu

let dump_store lu lhs lhst rhs rhst =
  let _ = Llvm.build_store rhs lhs lu.lu_builder in
  lhs

let dump_assignment lu lhse rhse lhst  =
  let rhst = Semast.type_name_of_s_expression rhse in
```

```
  let (lu, rhs) = dump_s_expression_temporary lu rhse in
  let (lu, lhs) = dump_s_expression lu lhse in
  let v = dump_store lu lhs lhst rhs rhst in
  ( lu, v )

let dump_s_variable_definition lu = function
  | Semast.SVarBinding((n, tn), rhse) -> let lhse = Semast.SQualifiedId([n], tn) in
    let lhst = tn in
    let rhst = Semast.type_name_of_s_expression rhse in
    let (lu, rhs) = dump_s_expression_temporary lu rhse in
    let (lu, lhs) = dump_s_expression lu lhse in
    let v = dump_store lu lhs lhst rhs rhst in
    ( lu, v )

let rec dump_s_general_statement lu gs =
  let acc lu bgs =
    dump_s_general_statement lu bgs
  in
  match gs with
  | Semast.SGeneralBlock( locals, gsl ) ->
    let lu = dump_s_locals lu locals in
    let lu = List.fold_left acc lu gsl in
    lu
  | Semast.SExpressionStatement(e) ->
    let (lu, _) = dump_s_expression lu e in
    lu
  | Semast.SVariableStatement(vdef) ->
    let (lu, _) = dump_s_variable_definition lu vdef in
    lu

let rec dump_s_statement lu s =
  let acc lu s =
    dump_s_statement lu s
  in
  match s with
  | Semast.SBlock( locals, sl ) ->
    let lu = dump_s_locals lu locals in
    let lu = List.fold_left acc lu sl in
    lu
  | Semast.SGeneral(gs) ->
    dump_s_general_statement lu gs
  | Semast.SReturn(e) ->
    let lu = match e with
      | Semast.SNoop ->
```

```
        let _ = Llvm.build_ret_void lu.lu_builder in
        lu
      | e -> let (lu, v) = dump_s_expression_temporary lu e in
        let _ = Llvm.build_ret v lu.lu_builder in
        lu
    in
    lu
  | _ -> raise(Errors.Unsupported("This statement type is unsupported"))

let dump_s_variable_definition_global lu = function
  | Semast.SVarBinding((n, tn), e) ->
    let (lu, rhs) = dump_s_expression lu e in
    let v = Llvm.define_global n rhs lu.lu_module in
    let lu = { lu with
      lu_variables = StringMap.add n v lu.lu_variables
    } in
    (*
    let lty = llvm_type_of_s_type_name lu v in
    let v = Llvm.declare_global lty k lu.lu_module in
    { lu with lu_variables = StringMap.add k v lu.lu_variables }
    let v = llvm_lookup_variable lu n tn in
    let _ = Llvm.set_initializer v rhs in
    let lu = { lu with lu_variables = StringMap.add n v lu.lu_variables } in
    *)
    ( lu, v )

let dump_s_function_definition lu f =
  let acc lu s =
    dump_s_statement lu s
  in
  (* Generate the function with its signature *)
  (* Which means we just look it up in the llvm module *)
  let ft = Semast.type_name_of_s_function_definition f in
  let n = Semast.string_of_qualified_id f.Semast.func_name in
  let llfunc = llvm_lookup_function lu n ft in
  (* generate the body *)
  let entryblock = Llvm.append_block lu.lu_context "entry" llfunc in
    Llvm.position_at_end entryblock lu.lu_builder;
  let lu = dump_s_parameters lu llfunc f.Semast.
```

```
      func_parameters in
    let lu = List.fold_left acc lu f.Semast.func_body in
    let lu = { lu with
      lu_named_params = StringMap.empty;
    } in
    lu

let dump_s_basic_definition lu = function
    | Semast.SVariableDefinition(v) -> let (lu, _) =
      dump_s_variable_definition_global lu v in
      lu
    | Semast.SFunctionDefinition(f) ->
      dump_s_function_definition lu f

let dump_s_definition lu = function
    | Semast.SBasic(b) -> dump_s_basic_definition lu b

let dump_array_prelude lu =
    (* Unfortunately, unsupported... *)
    lu

let dump_parallelism_prelude lu =
    (* Unfortunately, unsupported... *)
    lu

let dump_global_string lu n v =
    let rhs = Llvm.const_stringz lu.lu_context v in
    let v = Llvm.define_global n rhs lu.lu_module in
    (lu, v)

let dump_builtin_lib lu =
    let char_t  = Llvm.i8_type      lu.lu_context
    and i32_t   = Llvm.i32_type     lu.lu_context
    (* LLVM treats booleans as 1-bit integers, not distinct
      types with their own true / false *)
    in
    let p_char_t  = Llvm.pointer_type    char_t
    and llzero = Llvm.const_int i32_t 0
    in
    let f_acc lu (n, lv) =
      { lu with lu_functions = StringMap.add n lv lu.
      lu_functions }
    in
    let print_lib lu =
      let printf_t = Llvm.var_arg_function_type i32_t [|
```

```
p_char_t |] in
 let printf_func = Llvm.declare_function "printf" printf_t lu.lu_module in
 let (_, int_format_str) = dump_global_string lu "__ifmt" "%d"
 and (_, str_format_str) = dump_global_string lu "__sfmt" "%s"
 and (_, float_format_str) = dump_global_string lu "__ffmt" "%f"
 in
 let handler_name = "printf" in
 let handler lu el =
   let ( lu, exprl ) = ( dump_arguments_gen ( dump_s_expression ) lu el ) in
   if (List.length el) < 1 then ( lu, exprl ) else
   let hdt = Semast.type_name_of_s_expression ( List.hd el ) in
   let insertion = match hdt with
     | Semast.SBuiltinType(Ast.String, _) -> str_format_str
     | Semast.SBuiltinType(Ast.Float(n), _) -> float_format_str
     | Semast.SBuiltinType(Ast.Int(n), _) -> int_format_str
     | _ -> raise(Errors.BadPrintfArgument)
   in
   let fptr = Llvm.build_gep insertion [| llzero; llzero |] "tmp.fmt" lu.lu_builder in
   ( lu, fptr :: exprl )
 in
 let libprintfuncs = [
   (( Semast.mangle_name ["lib"; "print"] ( Semast.SFunction(Semast.void_t, [Semast.string_t], Semast.no_qualifiers)) ), printf_func);
   (( Semast.mangle_name ["lib"; "print"] ( Semast.SFunction(Semast.void_t, [Semast.float64_t], Semast.no_qualifiers)) ), printf_func);
   (( Semast.mangle_name ["lib"; "print"] ( Semast.SFunction(Semast.void_t, [Semast.int32_t], Semast.no_qualifiers)) ), printf_func);
 ] in
 let lu = List.fold_left f_acc lu libprintfuncs in
 let lu = { lu with
   lu_handlers = ( StringMap.add handler_name ( handler ) lu.lu_handlers )
```

```
    } in
    lu
  in
  let math_lib lu =
    lu
  in
  let lu = print_lib lu in
  let lu = math_lib lu in
  lu

let dump_builtin_module lu = function
  | Semast.Lib -> dump_builtin_lib lu

let dump_module_import lu = function
  | Semast.SBuiltin(lib) -> dump_builtin_module lu lib
  | Semast.SCode(_) -> lu
  | Semast.SDynamic(_) -> lu

let dump_declarations lu =
  let rec declare k lu t = match t with
    | Semast.SOverloads(fl) ->
      ( List.fold_left (declare k) lu fl )
    | Semast.SFunction(rt,args,tq) as ft -> let lty =
  llvm_type_of_s_type_name lu t in
      let mk = Semast.mangle_name [k] ft in
      let v = Llvm.declare_function mk lty lu.lu_module in
      { lu with lu_functions = StringMap.add mk v lu.
  lu_functions }
    | _ -> lu
  in
  let acc_def k t lu =
    declare k lu t
  in
  let toplevel = lu.lu_env.Semast.env_definitions in
  let lu = StringMap.fold acc_def toplevel lu in
  lu

let dump_prelude lu sprog =
  let lu = dump_array_prelude lu in
  let lu = dump_parallelism_prelude lu in
  let lu = List.fold_left dump_module_import lu lu.lu_env.
  Semast.env_imports in
  let lu = dump_declarations lu in
  lu

```

```
let generate sprog =
  let acc_def lu d =
    let lu = dump_s_definition lu d in
    lu
  in
  let lu = create_li_universe sprog in
  let lu = dump_prelude lu sprog in
  let lu = match sprog with
    | Semast.SProgram(_, _, defs) -> ( List.fold_left
   acc_def lu defs )
  in
  lu.lu_module
```

../source/codegen.ml