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
```

```
4             pixel[] toprow = img[ri];
5             // 0-based indexing
6             pixel[] bottomrow = img[img.height - ri - 1];
7             for (int ci = 0; ci < img.width; ++ci) {
8                  // generic swap call in library
9                  lib.swap(toprow[ci], bottomrow[ci]);
10            }
11       }
12  }
13
14  function int main () {
15       image img = lib.read("meow.png");
16       flip(img);
17       lib.save(img);
18
19       // implicit return 0:
20       // we want this to be able to work with
21       // command-line environments as well
22  }
```

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)
11          -t   --tokens          Print the stream of tokens
12          -a   --ast             Print the parsed Program
13          -s   --semantic        Print the Semantic Program
14          -l   --llvm            Print the generated LLVM code
15          -c   --compile         Compile the desired input and
    output the final LLVM
16          -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:

```
1 import lib
2
3 fun main () : int {
4       lib.print_n("hello world");
5       return 0;
6 }
```

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:

```
1 import lib
2
3 fun main () {
4       lib.print_n("hello world");
5       return 0;
```

```
6 }
```

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":

```
1  fun main () : int {
2       // 2 dimensional array
3       // of integers, initialized as a string
4       // from a "bitmap"
5       var arr : int[[]] = "\
6  |    | ||| |    |    |||    |    | ||| ||| |    ||
7  |    | |   |    |    | |    |    | | | | | |    | |
8  |||||  ||  |    |    | |    | | | | | ||  |    | |
9  |    | |   |    |    | |    || || | | | | |    | |
10 |    | ||| ||| ||| ||| |    |    | ||| | | ||| || ";
11      lib.save("hello.bmp", arr);
12 }
```

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 (expression) = (expression) + 1.
2. (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

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

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

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

⟨*binary-digit*⟩ ::= one of
0 1

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

⟨*n-digit*⟩ ::= 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
‘_’

⟨*decimal-digit-sequence*⟩ ::= ⟨⟩
| ⟨*decimal-digit*⟩ ⟨*decimal-digit-sequence*⟩

⟨*binary-digit-sequence*⟩ ::= ⟨⟩
| ⟨*binary-digit*⟩ ⟨*binary-digit-sequence*⟩

⟨*octal-digit-sequence*⟩ ::= ⟨⟩
| ⟨*octal-digit*⟩ ⟨*octal-digit-sequence*⟩

⟨*hexidecimal-digit-sequence*⟩ ::= ⟨⟩
| ⟨*hexidecimal-digit*⟩ ⟨*hexidecimal-digit-sequence*⟩

⟨*n-digit-sequence*⟩ ::= ⟨⟩
| ⟨*n-digit*⟩ ⟨*n-digit-sequence*⟩

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

⟨*integer-literal*⟩ ::= ⟨*decimal-literal*⟩
| ⟨*binary-literal*⟩
| ⟨*octal-literal*⟩
| ⟨*hexidecimal-literal*⟩
| ⟨*n-digit-literal*⟩

⟨*decimal-literal*⟩ ::= ⟨*decimal-digit-sequence*⟩

⟨*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
^	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

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

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

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

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

⟨*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*⟩

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

7.3.4 Jump Statements

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

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

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

⟨*function_definition*⟩ ::= ⟨*function_declaration*⟩ ⟨*block*⟩

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

7.5 Preprocessor

⟨*preprocessor_directive*⟩ ::= #define ⟨*identifier*⟩ ⟨*expression*⟩
| #ifdef ⟨*identifier*⟩
| #ifndef ⟨*identifier*⟩
| #endif
| #import ⟨*identifier*⟩
| #import "⟨*file_name*⟩"
| #import string ⟨*file_name*⟩

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.

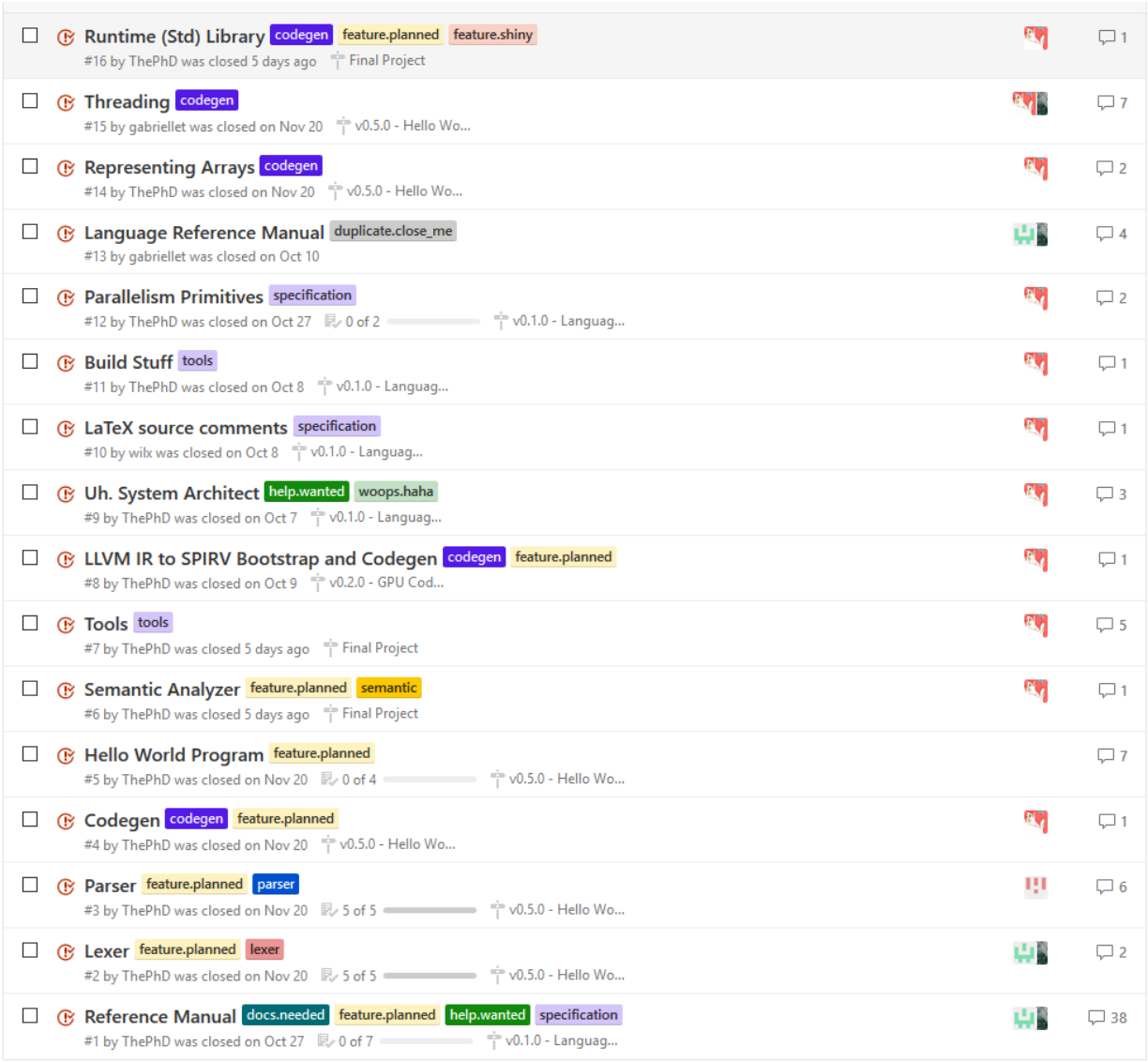


Figure 4.1: Closed issues throughout the project `https://github.com/ThePhD/lepix/issues?q=is%3Aissue+is%3Aclosed`.

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.

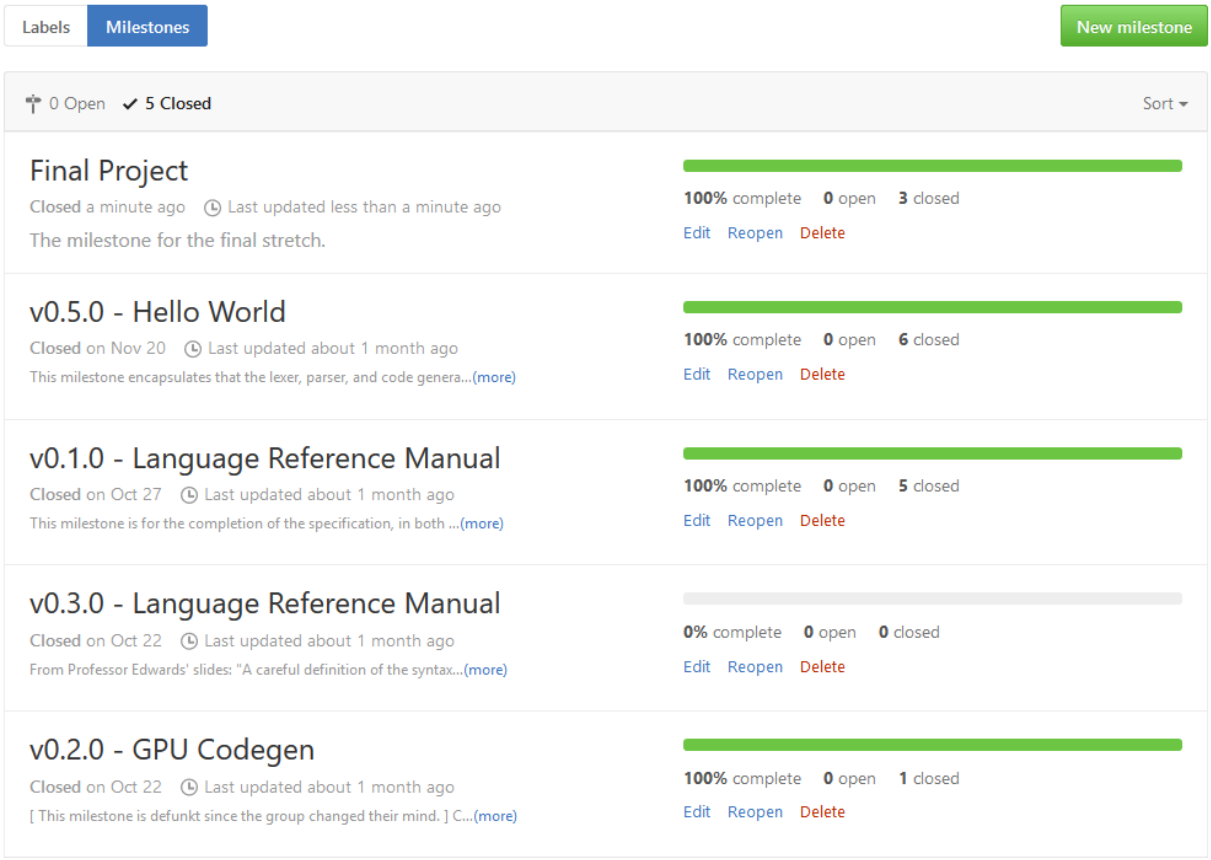


Figure 4.2: Milestones for the project `https://github.com/ThePhD/lepix/milestones?state=closed`.

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...
```

```
385 | |
386 * | commit 83afd3b
387 | | Author: ThePhD <phdofthehouse@gmail.com>
388 | | Date:   4 weeks ago
389 | |
390 | |     super duper docker
391 | |
392 * | commit 78f655d
393 | | Author: ThePhD <phdofthehouse@gmail.com>
394 | | Date:   4 weeks ago
395 | |
396 | |     Goddamn tabs
397 | |
398 * | commit cca9cd2
399 | | Author: ThePhD <phdofthehouse@gmail.com>
400 | | Date:   4 weeks ago
401 | |
402 | |     update travis work
403 | |
404 * | commit c0e85a3
405 | | Author: ThePhD <phdofthehouse@gmail.com>
406 | | Date:   4 weeks ago
407 | |
408 | |     Use a different language to attempt to stay out of the
      python shell
409 | |
410 * | commit 9e00c5f
411 | | Author: ThePhD <phdofthehouse@gmail.com>
412 | | Date:   4 weeks ago
413 | |
414 | |     update tests file and travis CI yaml file
415 | |
416 * | commit 8b71e6d
417 | | Author: ThePhD <phdofthehouse@gmail.com>
418 | | Date:   4 weeks ago
419 | |
420 | |     Add .travis file to start CI
421 | |
422 * | commit 0f25039 (origin/feature/codegen, feature/codegen)
423 | | Author: ThePhD <phdofthehouse@gmail.com>
424 | | Date:   4 weeks ago
425 | |
426 | |     Testing fixture
427 | |
428 * | commit e200757
429 | | Author: ThePhD <phdofthehouse@gmail.com>
430 | | Date:   4 weeks ago
431 | |
```

```
| |     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
|
```

```
654 |     Almost done with AST and pretty printing for all language
      constructs. Only function and variable decls to go
655 |
656 * commit 71a6fcb
657 | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
658 | Date:   5 weeks ago
659 |
660 |     Added simple top level. Edited ast, parser and lexer but
      some errors remain. Pretty printing needs to be set up.
661 |
662 * commit d29fa01
663 | Author: Fatima <fatimakoli14@gmail.com>
664 | Date:   6 weeks ago
665 |
666 |     Parallelblock and jump statements added
667 |
668 *   commit 772f3b9
669 |\  Merge: e039350 2a939a4
670 | | Author: Fatima <fatimakoli14@gmail.com>
671 | | Date:   6 weeks ago
672 | |
673 | |     Merge branch 'master' of https://github.com/ThePhD/lepix
674 | |
675 | * commit 2a939a4
676 | | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
677 | | Date:   6 weeks ago
678 | |
679 | |     Deleted intermediate files and yacc output
680 | |
681 | * commit 80e8614
682 | | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
683 | | Date:   6 weeks ago
684 | |
685 | |     Parser simplified. Multiple expression grammar rules
      collapsed into single rule
686 | |
687 | *   commit 7fd0c99
688 | |\  Merge: bb3f3bb 1dcf3f5
689 | | | Author: ThePhD <phdofthehouse@gmail.com>
690 | | | Date:   6 weeks ago
691 | | |
692 | | |     Merge remote-tracking branch 'origin/master'
693 | | |
694 | * | commit bb3f3bb
695 | | | Author: ThePhD <phdofthehouse@gmail.com>
696 | | | Date:   6 weeks ago
697 | | |
698 | | |     Makin' a bootstrapper....
699 | | |
```

```
| * | 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
|
```

```
795 |     Added top level file (lepix.ml) and deleted intermediate
      files from lex and yacc
796 |
797 *   commit 6d409f9
798 |\  Merge: 542ee64 dbf4809
799 | | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
800 | | Date:   7 weeks ago
801 | |
802 | |     Merge branch 'master' of https://github.com/ThePhD/lepix
803 | |
804 | * commit dbf4809
805 | | Author: fennilin <jackielin13@gmail.com>
806 | | Date:   7 weeks ago
807 | |
808 | |     Create 11-03-16
809 | |
810 * | commit 542ee64
811 |/  Author: Akshaan Kakar <akshaan.crackers@gmail.com>
812 |   Date:   7 weeks ago
813 |
814 |       Added missing tokens to parser
815 |
816 * commit b47e043
817 | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
818 | Date:   7 weeks ago
819 |
820 |     All rules added to parser. No S/R or R/R conflicts. Need
      to defined entry point for compiler (i.e. 'main')
821 |
822 *   commit 980e2a3
823 |\  Merge: 0ecdb5f f6283ac
824 | | Author: Akshaan Kakar <akshaan.crackers@gmail.com>
825 | | Date:   8 weeks ago
826 | |
827 | |     Merge branch 'master' of https://github.com/ThePhD/lepix
828 | |
829 | |     Edited parser.mly
830 | |
831 | * commit f6283ac
832 | | Author: ThePhD <phdofthehouse@gmail.com>
833 | | Date:   8 weeks ago
834 | |
835 | |     preprocessor is still eluding me with a parse error.
      Need to get more info about this.
836 | |
837 | * commit 847c13c
838 | | Author: ThePhD <phdofthehouse@gmail.com>
839 | | Date:   8 weeks ago
840 | |
```

```
| |     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
| |
```

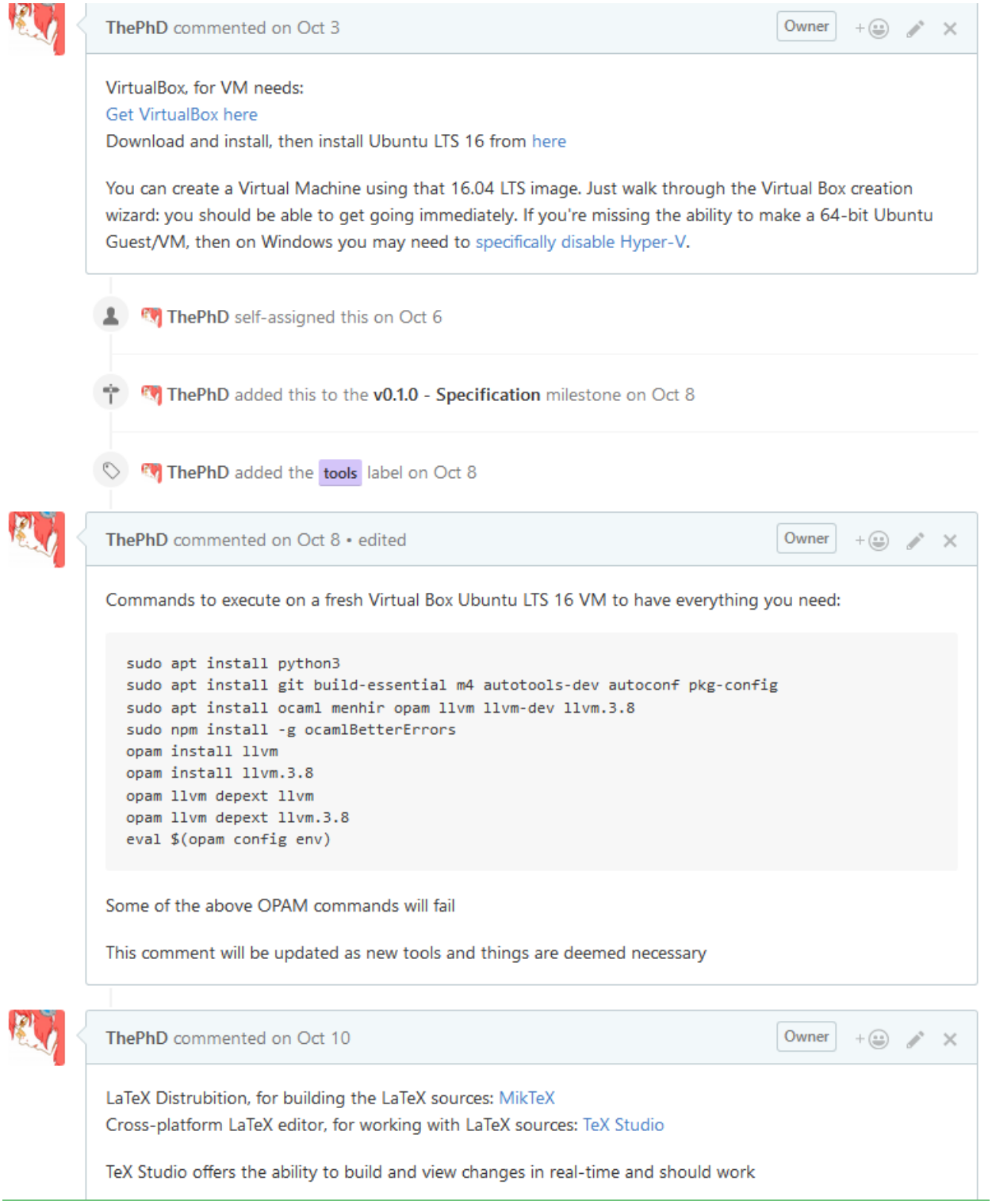
```
| * commit 35483aa
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     Toplevel PDFs we can link to.
| |
| * commit 42168dd
| | Author: ThePhD <phdofthehouse@gmail.com>
| | Date:   8 weeks ago
| |
| |     final specificaiton before submission
| |
| * commit e29b45e
| | 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

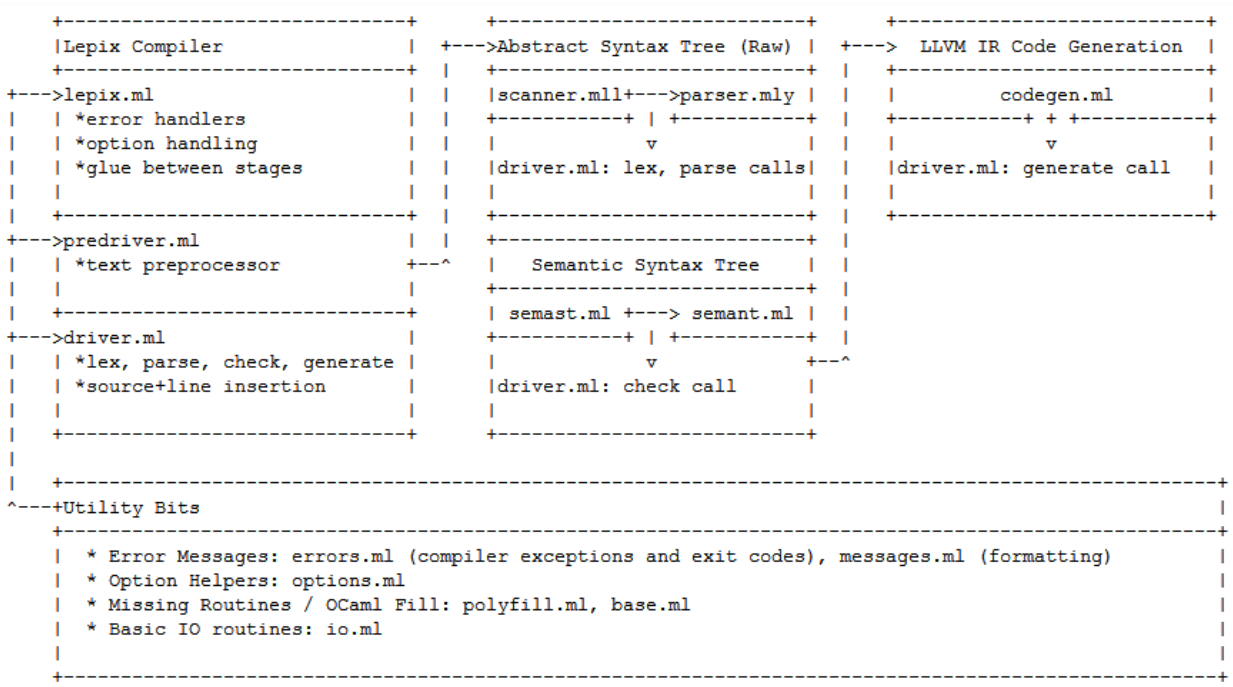


Figure 5.1: Compiler and implementation organization.

way I 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.

5.1.2 Namespaces

Namespaces became trivial to implement. Since namespaces themselves are not allowed to be a type – just a meta-construct for organization – they needed no physical entry in the final Semantic Syntax Tree. All names of definitions were simply folded to be fully qualified. It would behoove me in further implementation to keep track of the original unqualified name, to ensure that I can easily access both versions of the information easily.

The only other thing that would have come in handy to implement is using namespace syntax. As of right now, accessing things in namespaces requires fully qualified names. This could get very annoying very quickly, and therefore some form of a using statement would be good.

5.1.3 Bottom-up Type Derivation

Bottom-up Type Derivation was, essentially, just analyzing the return values of a function. The implementation does not have checkers for expressions that initialize a variable, so the mechanism is not in place for automatic variable deduction, even though the Semantic Syntax Tree will properly tag a variable definition without a type specifier to be of automatically-deduced type.

Thankfully, return values for functions were done. Here, I simply re-ran the semantic analyzer after going through once and getting the return types of all functions and return expressions, and then ran it again to re-gather symbols into the global environment and program attributes. The first Semantic Syntax Tree was thrown out entirely in this process. It is a cheap programmatic way to get the derivations to resolve for all the top-level symbols gathered earlier. A much more effective and composed method would be nice for later, but I had to implement things as fast as possible.

5.1.4 Overloading

Overloading was conceptually difficult to figure out. Many languages have overloading, but a lot of the code examples I had seen were done at runtime. That is, argument arity and strict type matching were done, overloads ranked by that, and then any conversion paths were taken into consideration before determining if a single function call was the best function call. We implemented similar here, without taking into account conversions (because that would require ranking all overloads fully, which is a veritable nightmare to do properly and the source of hundreds – perhaps thousands – of bugs in C++, C#, and other language compilers).

It was important to note that overloading is not part of the original AST, just the Semantic AST as a type for Qualified IDs (only identifiers can be overloaded) and for codegen purposes become mangled names to prevent name lookup confusion (the source of a bug that plagued even past the original deadline and only fixed recently).

5.1.5 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);
```

```
31      return svalue;
32  }
```

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 =
```

```
30    let end_index = start_index + len - 1 in
31    if start_index >= end_index then value else
32    let accumulated = ref value
33    in
34    for i = start_index to end_index do
35      accumulated := ( f !accumulated i )
36    done;
37    !accumulated
38
39  let foldi_to f value start_index end_index =
40    foldi f value start_index (end_index - start_index)
41
42  (* Integer *)
43  let rec powi n = function
44    | 0 -> 1
45    | 1 -> n
46    | x -> n * ( powi n  x - 1 )
47
48  let int_of_bool b = if b then 1 else 0
49
50  let int_of_string_base b s =
51    let len = (String.length s) in
52    let acc num i = let c = s.[i] in
53      let v = if c >= '0' || c <= '9' then
54          int_of_char c - int_of_char '0'
55        else
56          if c >= 'A' || c <= 'Z' then
57            int_of_char c - int_of_char 'A' + 10
58          else
59            if c >= 'a' || c <= 'z' then
60              int_of_char c - int_of_char '0' + 10
61            else 0
62      and place = len - 1 - i
63      in
64      num + ( v * ( powi b place ) )
65    in
66    foldi acc 0 0 len
67
68  (* Num *)
69
70  exception BadBase of string
71  exception DigitGreaterThanBase of string
72
73  let num_of_string_base_part b s =
74    if b > 36 || b < 1 then raise(BadBase "num_of_string_base
```

```
      : base cannot be greater than 36 or less than 1") else
75    let n0 = Num.num_of_int 0 in
76    let len = (String.length s) in
77    if len < 1 then n0 else
78    let (mid, starter) = try ( ( String.index s '.' ), 1 )
       with _ -> (len - 1, 0)
79    and nb = Num.num_of_int b
80    in
81    let acc (n, skipval) i = let c = s.[i] in
82      if c = '.' then (n, skipval - 1) else
83      let v = if c >= '0' && c <= '9' then
84          ( int_of_char c ) - ( int_of_char '0' )
85        else
86          if c >= 'A' && c <= 'Z' then
87            ( int_of_char c ) - ( int_of_char 'A' ) + 10
88          else
89            if c >= 'a' && c <= 'z' then
90              int_of_char c - ( int_of_char 'a' ) + 10
91            else 0
92      and place = mid - i - skipval
93      in
94      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
95      let nv = Num.num_of_int v
96      and nplace = Num.num_of_int place
97      in
98      (Num.add_num n ( Num.mult_num nv ( Num.power_num nb
       nplace ) ), skipval)
99    in
100   let (n, _) = foldi acc (n0, starter) 0 len in
101   n
102
103 let num_of_string_base b s =
104   num_of_string_base_part b s
105
106 let num_of_string s =
107   let slen = String.length s in
108   try
109     let eidx = String.index s 'e' in
110     if eidx < 1 then raise(Not_found);
111     let eidxp1 = ( eidx + 1 ) in
112     let nval = num_of_string_base_part 10 (String.sub s 0
       eidx)
```

```
113       and eval = if eidxp1 < slen then
          num_of_string_base_part 10 (String.sub s eidxp1 (slen -
          eidxp1)) else ( Num.num_of_int 0 )
114       in
115       Num.mult_num nval ( Num.power_num (Num.num_of_int 10)
          eval )
116     with
117       | Not_found -> num_of_string_base_part 10 s
118
119   (* Char *)
120   let is_whitespace = function
121     | ' ' -> true
122     | '\t' -> true
123     | '\n' -> true
124     | '\r' -> true
125     | _ -> false
126
127   (* String *)
128   let string_to_list s =
129     let l = ref [] in
130     let acc c =
131       l := c :: !l; ()
132     in
133     String.iter acc s;
134     List.rev !l
135
136   let iteri f start_index len =
137     let end_index = start_index + len - 1 in
138     if start_index < end_index then
139       for i = start_index to end_index do
140         ( f i )
141       done
142
143   type split_option =
144     | RemoveDelimeter
145     | KeepDelimeter
146
147   let string_split_with v s opt =
148     let e = String.length s
149     and vlen = String.length v
150     in
151     if vlen >= e then [s] else
152     let forward_search start =
153       let acc found idx =
154         found && ( s.[start + idx] = v.[idx] )
```

```
155       in
156       foldi acc true 1 (vlen - 1)
157     in
158     let add_sub len slist start =
159       if len < 1 then ( start, slist ) else
160       let fresh = ( String.sub s start len )
161       and last = start + len + vlen in
162       begin match opt with
163         | RemoveDelimeter -> ( last, fresh :: slist )
164         | KeepDelimeter ->
165           let slist = v :: slist in
166           ( last, fresh :: slist )
167       end
168     in
169     let acc (last, slist) start =
170       if (start < last) then (last, slist) else
171       if ( s.[start] = v.[0] ) then
172         if (forward_search start) then
173           let len = start - last in
174           (add_sub len slist last )
175         else
176           (last, slist)
177       else
178         if ( start = ( e - 1 ) ) then
179           let len = e - last in
180           (add_sub len slist last )
181         else
182           (last, slist)
183     in
184     let (_, slist) = ( foldi acc (0, []) 0 e ) in
185     (* Return complete split list *)
186     List.rev slist
187
188   let string_starts_with str pre =
189     let prelen = (String.length pre) in
190     prelen <= (String.length str ) && pre = (String.sub str 0
        prelen)
191
192   let string_split v s =
193     string_split_with v s RemoveDelimeter
```

../source/polyfill.ml

```
1   (* LePiX Language Compiler Implementation
2   Copyright (c) 2016- ThePhD
3
4   Permission is hereby granted, free of charge, to any person
```

```
     obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
        be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Base types and routines. *)
22
23 type token_source = {
24      token_source_name : string;
25   token_number : int;
26      token_line_number : int;
27      token_line_start : int;
28   token_column_range : int * int;
29   token_character_range : int * int;
30 }
31
32 type target =
33   | Pipe
34   | File of string
35
36 let target_to_string = function
37   | Pipe -> "pipe"
38   | File(s) -> "file: " ^ s
```

```
39
40 let target_to_pipe_string i b = match i with
41   | Pipe -> if b then "stdin" else "stdout"
42   | File(s) -> "file: " ^ s
43
44 type action =
45   | Help
46   | Preprocess
47   | Tokens
48   | Ast
49   | Semantic
50   | Llvm
51   | Compile
52
53 let action_to_int = function
54   | Help -> -1
55   | Preprocess -> 0
56   | Tokens -> 1
57   | Ast -> 10
58   | Semantic -> 100
59   | Llvm -> 1000
60   | Compile -> 10000
61
62 let entry_point_name = "main"
63
64 (* Core options *)
65
66 let default_integral_bit_width = 32
67 let default_floating_bit_width = 64
68
69 (* Error message helpers *)
70
71 let line_of_source src token_info =
72   let ( absb, abse ) = token_info.token_character_range
73   and linestart = token_info.token_line_start
74   in
75   let ( lineend, _ ) =
76     let f (endindex, should_skip) idx =
77       let c = src.[idx] in
78       let skip_this = c = '\n' in
79       if should_skip || skip_this then
80         (endindex, true)
81       else
82         (endindex + 1, false)
83     in
```

```
84      Polyfill.foldi f ( linestart, false ) linestart ( (
        String.length src ) - linestart )
85    in
86    let srcline = String.sub src linestart (max 0 (lineend -
        linestart - 1)) in
87    let srclinelen = String.length srcline in
88    let ( srcindent, _ ) =
89      let f (s, should_skip) idx =
90        let c = srcline.[idx] in
91        let nws = not ( Polyfill.is_whitespace c ) in
92        if should_skip || nws then
93          (s, false)
94        else
95          (s ^ ( String.make 1 c ), true)
96      in
97      Polyfill.foldi f ( "", false ) 0 srclinelen
98    in
99    let indentlen = String.length srcindent
100   and tokenlen = lineend - absb
101   in
102   ( srcline, srcindent, (max ( srclinelen - indentlen -
        tokenlen ) 0 ) )
103
104
105 let brace_tabulate str tabs =
106   let len = ( String.length str ) in
107   let lines = Polyfill.string_split_with "\n" str Polyfill.
        KeepDelimeter in
108   let lineslen = ( List.length lines ) in
109   let buf = Buffer.create ( len + ( lineslen * 4 ) ) in
110   let acc ( buf, t ) line =
111     let tmod = 0 - ( Polyfill.int_of_bool ( String.contains
        line '}' ) ) in
112     let t = t + tmod in
113     Buffer.add_string buf (String.make t '\t'); Buffer.
      add_string buf line;
114     let t = t + ( Polyfill.int_of_bool ( String.contains
      line '{' ) ) in
115     (buf, t)
116   in
117   let (buf, _) = List.fold_left acc ( buf, tabs ) lines in
118   Buffer.contents buf
```

../source/base.ml

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

```
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Options / argument parser *)
22
23 type option =
24   | Dash of string
25   | DoubleDash of string
26   | Argument of int * string
27
28 type options_context = {
29      mutable options_help : string -> string;
30 }
31
32 let read_options ocontext sys_argv =
33   let argc = Array.length sys_argv - 1 in
34   (* Skip first argument one (argv 0 is the path
35   of the exec on pretty much all systems) *)
36   let argv = ( Array.sub sys_argv 1 argc )
```

```
37    and action = ref Base.Help
38    and verbose = ref false
39    and input = ref Base.Pipe
40    and output = ref Base.Pipe
41    and specified = ref []
42    and seen_stdin = ref false
43    in
44    (* Our various options *)
45    let update_action a =
46      specified := a :: !specified;
47      if ( Base.action_to_int !action ) < ( Base.
       action_to_int a ) then
48        action := a;
49    in
50    let options = [
51      ( 1, "h", "help", "print the help message",
52        fun _ _ -> ( update_action(Base.Help) )
53      );
54      ( 1, "p", "preprocess", "Preprocess and display source"
       ,
55        fun _ _ -> ( update_action(Base.Preprocess) )
56      );
57      ( 1, "i", "input", "Take input from standard in (
       default: stdin)",
58        fun _ _ -> ( input := Base.Pipe; seen_stdin := true )
59      );
60      ( 2, "o", "output", "Set the output file (default:
       stdout)",
61        fun _ o -> ( output := Base.File(o) )
62      );
63      ( 1, "t", "tokens", "Print the stream of tokens",
64        fun _ _ -> ( update_action(Base.Tokens) )
65      );
66      ( 1, "a", "ast", "Print the parsed Program",
67        fun _ _ -> ( update_action(Base.Ast) )
68      );
69      ( 1, "s", "semantic", "Print the Semantic Program",
70        fun _ _ -> ( update_action(Base.Semantic) )
71      );
72      ( 1, "l", "llvm", "Print the generated LLVM code",
73        fun _ _ -> ( update_action(Base.Llvm) )
74      );
75      ( 1, "c", "compile", "Compile the desired input and
       output the final LLVM",
76        fun _ _ -> ( update_action(Base.Compile) )
```

```
77      );
78      ( 1, "v", "verbose", "Be as explicit as possible with
      all steps",
79        fun _ _ -> ( verbose := true )
80      );
81    ]
82    and position_option arg_index positional_index arg =
83      if Sys.file_exists arg then
84        input := Base.File( arg )
85      else
86        raise(Errors.OptionFileNotFound(arg))
87    in
88    let help tabulation =
89      let value_text = "<value>" in
90      let value_text_len = String.length value_text in
91      let longest_option =
92        let acc len o = match o with
93          | (sz, _, long, _, _ ) ->
94            let newlen = (String.length long)
95              + if sz = 2 then 1 + value_text_len else 0
96            in
97            if newlen > len then newlen else len
98        in
99        let l = 1 + ( List.fold_left acc 1 options ) in
100       if l < value_text_len then value_text_len else l
101     in
102     let concat_options t =
103       let builder s o = match o with
104         | ( sz, short, long, desc, _ ) ->
105           let long_len = String.length long in
106           let spacing_size = longest_option - long_len - (
      if sz = 2 then 1 + value_text_len else 0 ) in
107           let spacing_string = (String.make spacing_size '
      ') in
108           s
109           ^ "\n" ^ t ^ "-" ^ short
110           ^ "\t--" ^ long ^ (if sz = 1 then "" else " " ^
      value_text)
111           ^ spacing_string
112           ^ desc
113       in
114       (List.fold_left builder "" options)
115     in
116     let (_, input_short, input_long, _, _) = List.nth
      options 2 in
```

```
117       let msg = "Help:"
118         ^ "\n" ^ tabulation ^ "lepix [options] filename [
      filenames...]"
119         ^ "\n" ^ tabulation ^ "\t" ^ "filename | filenames
      can have one option -" ^ input_short ^ " or --" ^
      input_long
120         ^ "\n" ^ tabulation ^ "options:"
121         ^ ( concat_options (tabulation ^ "\t") )
122       in
123       msg
124     in
125     ocontext.options_help <- help;
126     (* Exit early if possible *)
127     if argc < 1 then
128       (!input, !output, !action, !specified, !verbose)
129     else
130
131     let to_option idx arg =
132       let arglen = String.length arg in
133       match arg with
134         | _ when Polyfill.string_starts_with arg "--" ->
      DoubleDash((String.sub arg 2 (arglen - 2)))
135         | _ when Polyfill.string_starts_with arg "-" -> Dash
      ((String.sub arg 1 (arglen - 1)))
136         | _ -> Argument(idx, arg)
137     in
138     (* Convert all arguments to the Option type first *)
139     let options_argv = Array.mapi to_option argv in
140
141     (* Function for each argument *)
142     let f (index, positional_index, skip_next) option_arg =
143       if skip_next then (1 + index, positional_index, false)
      else
144       let execute_on_match_sub_option ( opt_failure,
      should_block ) opt_string pred = match opt_failure with
145         (* There is some failure, so just propogate it
      through *)
146         | Some(x) -> ( opt_failure, should_block )
147         (* There is no failure, so now work with the list *)
148         | None -> begin match List.filter pred options with
149           (* We use filter instead of find because find is
      dumb and throws an
150           exception instead of just returning an optional
      because
151           whoever designed the OCaml standard library is an
```

```
absolute
152         bell end. *)
153         | ( 1, _, _, _, f ) :: tail -> (* Only needs 1
argument *)
154           (f opt_string "");
155           ( opt_failure, should_block )
156         | ( 2, _, _, _, f ) :: tail -> (* Needs 2 arguments
, look ahead by 1 *)
157           if ( index + 1 ) >= argc  then
158             raise(Errors.MissingOption(opt_string));
159           let nextarg = ( options_argv.(1 + index) ) in
160           let _ = match nextarg with
161             | Argument(idx, s) -> (f opt_string s)
162             | _ -> raise(Errors.BadOption(opt_string))
163           in
164           ( opt_failure, true )
165         | _ -> (* Unhandled case: return new failure string
 *)
166           ( Some opt_string, should_block )
167       end
168     and on_failure dashes opt arglist arg =
169       let msg = dashes ^ opt
170         ^ if ( List.length arglist ) > 1 then " ( in " ^
dashes ^ arg ^ " )" else ""
171       in
172       raise(Errors.BadOption(msg))
173     in
174     let ( should_skip_next, was_positional ) = match
option_arg with
175       | Dash(arg) ->
176         (* if it has a dash only *)
177         (* each letter can be its own thing *)
178         let perletter (opt_failure, should_break) c =
179           let opt_string = ( String.make 1 c ) in
180           let short_pred (_, short, _, _, _ ) =
181             short = opt_string
182           in
183           execute_on_match_sub_option (opt_failure,
should_break) opt_string short_pred
184         in
185         (* look at every character. If there's 1 match
among them, go crazy *)
186         let arglist = (Polyfill.string_to_list arg) in
187         let (opt_failure, causes_skip) = ( List.fold_left
perletter ( None, false ) arglist ) in
```

```
188           begin match opt_failure with
189             | None ->
190               ( causes_skip, 0 )
191             | Some(opt) -> let _ = (on_failure "--" opt
      arglist arg) in
192               ( causes_skip, 0 )
193           end
194         | DoubleDash(arg) ->
195           (* if it has a double dash... *)
196           (* each comma-delimeted word can be its own option
      *)
197           let perword (opt_failure, problems) opt_string =
198             let long_pred (_, _, long, _, _ ) =
199               long = opt_string
200             in
201             execute_on_match_sub_option (opt_failure,
      problems) opt_string long_pred
202           in
203           (* look at word character. If there's 1 match among
       them, go crazy *)
204           let arglist = Polyfill.string_split "," arg in
205           let (opt_failure, causes_skip) = ( List.fold_left
      perword ( None, false ) arglist ) in
206           begin match opt_failure with
207             | None ->
208               ( causes_skip, 0 )
209             | Some(opt) -> let _ = (on_failure "--" opt
      arglist arg) in
210               ( causes_skip, 0 )
211           end
212         (* otherwise, it's just a positional argument *)
213         | Argument(idx, arg) ->
214           (position_option index positional_index arg);
215           ( skip_next, 1 )
216       in
217       (1 + index, positional_index + was_positional,
      should_skip_next)
218     in
219     (* Iterate over the arguments *)
220     let _ = Array.fold_left f (0, 0, false) options_argv in
221     (* Return tuple of input, output, action *)
222     ( !input, !output, !action, !specified, !verbose )
```

../source/options.ml

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

```
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Message formatters and helpers. *)
22
23
24 let preprocessing_error pcontext =
25   let ( t, info ) = pcontext.Predriver.token in
26   let ( source_line, source_indentation,
       columns_after_indent ) =
27     ( Base.line_of_source pcontext.Predriver.source_code
       info )
28   in
29   let column_range = info.Base.token_column_range in
30   let (column_text, is_columns_wide) = Representation.
       token_range_to_string column_range in
31   let msg = "Preprocessing Error in " ^ pcontext.Predriver.
       source_name ^ ":"
32   ^ "\n" ^ "\t" ^ "Unrecognizable parse pattern at token #"
```

```
      ^ ( string_of_int pcontext.Predriver.token_count )
33      ^ ": [id " ^ string_of_int info.Base.token_number ^ ":"
        ^ Representation.preparser_token_to_string t ^ "]"
34    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int info.Base.
      token_line_number
35    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: "
      else "Column: " ) ^ column_text
36    ^ "\n"
37    ^ "\n" ^ source_line
38    ^ "\n" ^ source_indentation ^ ( String.make
      columns_after_indent ' ' ) ^ "^~~~"
39    in
40    msg
41
42  let preprocessing_lexer_error pcontext core_msg c s e =
43    let ( t, info ) = pcontext.Predriver.token in
44    let ( source_line, source_indentation,
      columns_after_indent ) =
45      ( Base.line_of_source pcontext.Predriver.source_code
      info )
46    in
47    let column_range = info.Base.token_column_range in
48    let (column_text, is_columns_wide) = Representation.
      token_range_to_string column_range in
49    let msg = "Preprocessing Lexing Error in " ^ pcontext.
      Predriver.source_name ^ ":"
50    ^ "\n" ^ "\t" ^ core_msg ^ " at token#" ^ ( string_of_int
       pcontext.Predriver.token_count )
51      ^ ": [id " ^ string_of_int info.Base.token_number ^ ":"
        ^ Representation.preparser_token_to_string t ^ "]"
52    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int info.Base.
      token_line_number
53    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: "
      else "Column: " ) ^ column_text
54    ^ "\n"
55    ^ "\n" ^ source_line
56    ^ "\n" ^ source_indentation ^ ( String.make
      columns_after_indent ' ' ) ^ "^~~~"
57    in
58    msg
59
60  let lexer_error context core_msg c s e =
61    let abspos = s.Lexing.pos_cnum in
62    let endabspos = e.Lexing.pos_cnum in
63    let relpos = 1 + abspos - s.Lexing.pos_bol in
```

```
64    let endrelpos = 1 + endabspos - e.Lexing.pos_bol in
65    let (column_text, is_columns_wide) = Representation.
       token_range_to_string ( relpos, endrelpos ) in
66    let msg = "Lexing Error in " ^ context.Driver.source_name
        ^ ":"
67    ^ "\n" ^ "\t" ^ core_msg ^ " at character: " ^  c
68    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int s.Lexing.
       pos_lnum
69    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: "
       else "Column: " ) ^ column_text
70    in
71    msg
72
73  let parser_error context core_msg =
74    let ( t, info ) = context.Driver.token in
75    let ( source_line, source_indentation,
       columns_after_indent ) =
76      ( Base.line_of_source context.Driver.source_code info )
77    in
78    let column_range = info.Base.token_column_range in
79    let (column_text, is_columns_wide) = Representation.
       token_range_to_string column_range in
80    let msg = "Parsing Error in " ^ context.Driver.
       source_name ^ ":"
81    ^ "\n" ^ "\t" ^ core_msg ^ " at token #" ^ (
       string_of_int context.Driver.token_count )
82      ^ ": [id " ^ string_of_int info.Base.token_number ^ ":"
        ^ Representation.parser_token_to_string t ^ "]"
83    ^ "\n" ^ "\t" ^ "Line: " ^ string_of_int info.Base.
       token_line_number
84    ^ "\n" ^ "\t" ^ ( if is_columns_wide then "Columns: "
       else "Column: " ) ^ column_text
85    ^ "\n"
86    ^ "\n" ^ source_line
87    ^ "\n" ^ source_indentation ^ ( String.make
       columns_after_indent ' ' ) ^ "^~~"
88    in
89    msg
```

../source/message.ml

```
1  (* LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  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) }
39      );
40    }
41    and pcontext = {
42          Predriver.source_name = "";
43      Predriver.source_code = "";
44      Predriver.original_source_code = "";
45      Predriver.token_count = 0;
46      Predriver.token = ( Preparser.EOF,
47        { Base.token_source_name = ""; Base.token_number = 0;
48        Base.token_line_number = 0; Base.token_line_start =
      0;
49        Base.token_column_range = (0, 0); Base.
      token_character_range = (0, 0) }
50      );
51       }
52    in
53    let ocontext = {
54      Options.options_help = fun (s) -> ( "" );
55    } in
56    (* Call options Parser for Driver *)
57    let _ =
58    try
59      let ( i, o, a, s, v ) = ( Options.read_options ocontext
      Sys.argv ) in
60        input := i;
61        output := o;
62        action := a;
63        specified := s;
64        verbose := v
65    with
66      | err -> let _ = match err with
67        | Errors.BadOption(s) ->
68          let msg = "Options Error:"
69          ^ "\n" ^ "\t" ^ "Unrecognized option: " ^ s
70          ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
71          prerr_endline msg
72        | Errors.NoOption ->
73          let msg = "Options Error:"
74          ^ "\n" ^ "\t" ^ "No inputs or options specified"
75          ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
76          prerr_endline msg
77        | Errors.MissingOption(o) ->
78          let msg = "Options Error:"
79          ^ "\n" ^ "\t" ^ "Flag " ^ o ^ " needs an additional
```

```
      argument after it that is not dashed"
80          ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
81          prerr_endline msg
82        | Errors.OptionFileNotFound(f) ->
83          let msg = "Options Error:"
84          ^ "\n" ^ "\t" ^ "File " ^ f ^ " was not found"
85          ^ "\n" ^ ( ocontext.Options.options_help "\t" ) in
86          prerr_endline msg
87        | err ->
88          let msg = "Unknown Error during Option parsing:"
89          ^ "\n" ^ "\t" ^ "Contact the compiler vendor for
    more details and possibly include source code, or try
    simplifying the program"
90          in
91          prerr_endline msg;
92          raise(err)
93      in
94      (* Exit if arguments are wrong *)
95      ignore( ( exit Errors.option_error_exit_code ) )
96    in
97    (* Perform actual lexing and parsing using the Driver
     here *)
98    try
99      let allactions = !specified in
100     let source_name = ( Base.target_to_pipe_string !input
    true ) in
101     let output_to_target ( s ) = match !output with
102       | Base.Pipe -> ( print_endline s )
103       | Base.File(f) -> ( Io.write_file_text s f )
104     in
105     let print_predicate b =
106       fun v -> ( v = b )
107     in
108     let print_help () =
109       let msg = ( ocontext.Options.options_help "\t" ) in
110       print_endline msg
111     in
112     if !action = Base.Help then begin
113       print_help ()
114     end else
115     (* Since we do the actions in these functions multiple
    times,
116     We refactor them out here to make our lives easier
    while we tweak
117      stuff *)
```

```
118     let get_source () =
119       let pre_source_text = match !input with
120         | Base.Pipe -> Io.read_text stdin
121         | Base.File(f) -> ( Io.read_file_text f )
122       in
123       let source_text = Predriver.pre_process pcontext !
    input pre_source_text in
124       context.Driver.source_name <- source_name;
125       context.Driver.original_source_code <-
    pre_source_text;
126       context.Driver.source_code <- source_text;
127       source_text
128     in
129     let dump_tokens f tokenstream =
130       if ( List.exists (print_predicate Base.Tokens)
    allactions ) then f( Representation.
    parser_token_list_to_string tokenstream )
131     and dump_ast f program =
132       if ( List.exists (print_predicate Base.Ast)
    allactions ) then f( Representation.string_of_program
    program )
133     and dump_semantic f semanticprogram =
134       if ( List.exists (print_predicate Base.Semantic)
    allactions ) then f( Representation.string_of_s_program
    semanticprogram )
135     and dump_module f m =
136       f( Llvm.string_of_llmodule m )
137     in
138     let _ = match !action with
139       | Base.Help -> print_help ()
140       | Base.Preprocess ->
141         let source_text = get_source () in
142         output_to_target( source_text )
143       | Base.Tokens ->
144         let source_text = get_source () in
145         let lexbuf = Lexing.from_string source_text in
146         let tokenstream = Driver.lex source_name lexbuf in
147         ( dump_tokens output_to_target tokenstream )
148       | Base.Ast ->
149         let source_text = get_source () in
150         let lexbuf = Lexing.from_string source_text in
151         let tokenstream = Driver.lex source_name lexbuf in
152         ( dump_tokens print_endline tokenstream );
153         let program = Driver.parse context tokenstream in
154         ( dump_ast output_to_target program)
```

```
155       | Base.Semantic ->
156         let source_text = get_source () in
157         let lexbuf = Lexing.from_string source_text in
158         let tokenstream = Driver.lex source_name lexbuf in
159         ( dump_tokens print_endline tokenstream );
160         let program = Driver.parse context tokenstream in
161         ( dump_ast print_endline program );
162         let semanticprogram = Driver.analyze program in
163         ( dump_semantic output_to_target semanticprogram )
164       | Base.Llvm ->
165         let source_text = get_source () in
166         let lexbuf = Lexing.from_string source_text in
167         let tokenstream = Driver.lex source_name lexbuf in
168         ( dump_tokens print_endline tokenstream );
169         let program = Driver.parse context tokenstream in
170         ( dump_ast print_endline program );
171         let semanticprogram = Driver.analyze program in
172         ( dump_semantic print_endline semanticprogram );
173         let m = Codegen.generate semanticprogram in
174         if !verbose then ( dump_module print_endline m );
175         ( dump_module output_to_target m )
176       | Base.Compile ->
177         let source_text = get_source () in
178         let lexbuf = Lexing.from_string source_text in
179         let tokenstream = Driver.lex source_name lexbuf in
180         ( dump_tokens print_endline tokenstream );
181         let program = Driver.parse context tokenstream in
182         ( dump_ast print_endline program );
183         let semanticprogram = Driver.analyze program in
184         ( dump_semantic print_endline semanticprogram );
185         let m = Codegen.generate semanticprogram in
186         Llvm_analysis.assert_valid_module m;
187         ( dump_module output_to_target m )
188     in
189     ()
190   with
191     | err -> let _ = match err with
192       (* Preprocessor-Specific Errors *)
193       (* Preprocessing Parser Errors *)
194       | Preparser.Error ->
195         let msg = Message.preprocessing_error pcontext in
196         prerr_endline msg
197       | Errors.PreUnknownCharacter( c, (s, e) ) ->
198         let msg = Message.preprocessing_lexer_error
      pcontext "Unrecognized character in program" c s e in
```

```
199         prerr_endline msg
200
201       (* General Compiler Errors *)
202       (* Lexer Errors *)
203       | Errors.UnknownCharacter( c, (s, e) ) ->
204         let msg = Message.lexer_error context "Unrecognized
     character in program" c s e in
205         prerr_endline msg
206
207       | Errors.BadNumericLiteral( c, (s, e) ) ->
208         let msg = Message.lexer_error context "Bad
    character in numeric literal" c s e in
209         prerr_endline msg
210
211       (* Parser Errors *)
212       | Parser.Error
213       | Parsing.Parse_error ->
214         let msg = Message.parser_error context "
    Unrecognizable parse pattern" in
215         prerr_endline msg
216       | Errors.MissingEoF ->
217         let msg = "Parsing Error in" ^ context.Driver.
    source_name ^ ":"
218         ^ "\n" ^ "\t" ^ "Missing EoF at end of token stream
     (bad lexer input?)"
219         in
220         prerr_endline msg
221
222       (* Semantic Analyzer and Codegen Errors *)
223       (* Semantic Errors *)
224       (* TODO: positional information should be tracked
    through the AST and SemAST,
225       all the way to codegen, as well... *)
226       | Errors.BadFunctionCall(s) ->
227         let msg = "Bad Function Call error: " ^ s
228         in
229         prerr_endline msg
230       | Errors.FunctionAlreadyExists(s) ->
231         let msg = "Function Already Exists error: " ^ s
232         in
233         prerr_endline msg
234       | Errors.VariableAlreadyExists(s) ->
235         let msg = "Variable Already Exists error: " ^ s
236         in
237         prerr_endline msg
```

```
238         | Errors.TypeMismatch(s) ->
239           let msg = "Mismatched types error: " ^ s
240           in
241           prerr_endline msg
242         | Errors.IdentifierNotFound(s) ->
243           let msg = "Identifier Not Found error: " ^ s
244           in
245           prerr_endline msg
246         | Errors.InvalidFunctionSignature(s, n) ->
247           let msg = "Invalid signature: " ^ s ^ " in " ^ n
248           in
249           prerr_endline msg
250         | Errors.InvalidMainSignature(s) ->
251           let msg = "Invalid signature: " ^ s
252           in
253           prerr_endline msg
254         | Errors.InvalidBinaryOperation(s)
255         | Errors.InvalidUnaryOperation(s) ->
256           let msg = "Invalid operation: " ^ s
257           in
258           prerr_endline msg
259
260         (* Direct Codegen Errors *)
261         | Errors.UnknownVariable(s) ->
262           let msg = "Codegen (LLVM IR) error: " ^ s
263           in
264           prerr_endline msg
265         | Errors.UnknownFunction(s) ->
266           let msg = "Codegen (LLVM IR) error: " ^ s
267           in
268           prerr_endline msg
269         | Errors.VariableLookupFailure(name, _) ->
270           let msg = "Codegen (LLVM IR) error: could not
    properly find variable with the name " ^ name
271           in
272           prerr_endline msg
273         | Errors.FunctionLookupFailure(name, mangledname) ->
274           let msg = "Codegen (LLVM IR) error: looking for the
     function with the name " ^ name ^ " (mangled name: " ^
    mangledname ^ ")"
275           in
276           prerr_endline msg
277         | Errors.BadPrintfArgument ->
278           let msg = "Codegen (LLVM IR) error: lib.print and
    related functions only take either a string, an integer,
```

```
      or a floating point argument"
279         in
280         prerr_endline msg
281
282       (* Common Errors *)
283       (* Missing File/Bad File Name, Bad System Calls *)
284       | Sys_error(s) ->
285         let msg = "Sys_error: \n\t" ^ s
286         in
287         prerr_endline msg
288
289       (* Unsupported features *)
290       | Errors.Unsupported(s) ->
291         let msg = "Unsupported (ran out of implementation
    time): \n\t" ^ s
292         in
293         prerr_endline msg
294
295       (* Unknown Errors *)
296       | err ->
297         let msg = "Unknown Error during Compilation:"
298         ^ "\n" ^ "\t" ^ "Contact the compiler vendor for
    more details and possibly include source code, or try
    simplifying the program"
299         in
300         prerr_endline msg;
301         raise(err)
302     in
303     ignore( exit Errors.compiler_error_exit_code )
```

../source/lepix.ml

```
1  (* LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
   ), to deal in the Software
6  without restriction, including without limitation the
   rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
   of the Software, and to
8  permit persons to whom the Software is furnished to do so,
   subject to the following
9  conditions:
10
```

```
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* A listing of exceptions and the methods that power them
22 to make the parser more expressive *)
23
24 (* Driver and Related class of errors *)
25 let option_error_exit_code = 1
26
27 (* Option Errors *)
28 exception NoOption
29 exception BadOption of string
30 exception MissingOption of string
31 exception OptionFileNotFound of string
32
33 (* Compiler class of Errors *)
34 let compiler_error_exit_code = 2
35 (* Lexer Errors *)
36 exception PreUnknownCharacter of string * ( Lexing.position
       * Lexing.position )
37 exception UnknownCharacter of string * ( Lexing.position *
       Lexing.position )
38 exception BadNumericLiteral of string * ( Lexing.position *
       Lexing.position )
39
40 (* Parser Errors *)
41 exception MissingEoF
42 exception BadToken
43
44 (* Semantic and Codegen Errors *)
45 exception Unsupported of string
46 exception FunctionAlreadyExists of string
```

```
47 exception VariableAlreadyExists of string
48 exception IdentifierNotFound of string
49 exception TypeMismatch of string
50 exception BadFunctionCall of string
51 exception InvalidMainSignature of string
52 exception InvalidFunctionSignature of string * string
53 exception InvalidBinaryOperation of string
54 exception InvalidUnaryOperation of string
55
56 (* Codegen Errors *)
57 exception UnknownVariable of string
58 exception UnknownFunction of string
59 exception BadPrintfArgument
60 exception FunctionLookupFailure of string * string
61 exception VariableLookupFailure of string * string
```

../source/errors.ml

```
1  (* LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
```

```
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Drives the typical lexing and parsing algorithm
22 while adding pertinent source, line and character
       information. *)
23
24 type context = {
25       mutable source_name : string;
26       mutable source_code : string;
27   mutable original_source_code : string;
28   mutable token_count : int;
29       mutable token : Parser.token * Base.token_source;
30 }
31
32 let lex sourcename lexbuf =
33   let rec acc lexbuf tokennumber tokens =
34     let next_token = Scanner.token lexbuf
35     and startp = Lexing.lexeme_start_p lexbuf
36     and endp = Lexing.lexeme_end_p lexbuf
37     in
38     let line = startp.Lexing.pos_lnum
39     and relpos = (1 + startp.Lexing.pos_cnum - startp.
       Lexing.pos_bol)
40     and endrelpos = (1 + endp.Lexing.pos_cnum - endp.Lexing
       .pos_bol)
41     and abspos = startp.Lexing.pos_cnum
42     and endabspos = endp.Lexing.pos_cnum
43     in
44     let create_token token =
45       let t = ( token, { Base.token_source_name =
       sourcename; Base.token_number = tokennumber;
46           Base.token_line_number = line; Base.
       token_line_start = startp.Lexing.pos_bol;
47           Base.token_column_range = (relpos, endrelpos); Base
       .token_character_range = (abspos, endabspos) }
48       ) in
49       t
50     in
51     match next_token with
52     | Parser.EOF as token -> ( create_token token ) ::
       tokens
53     | token -> ( create_token token ) :: ( acc lexbuf ( 1 +
       tokennumber ) tokens )
54   in
55   acc lexbuf 0 []
```

```
56
57 let parse context token_list =
58   (* Keep a reference to the original token list
59   And use that to dereference rather than whatever crap we
       get from
60   the channel *)
61   let tokenlist = ref(token_list) in
62   let tokenizer _ = match !tokenlist with
63   (* Break each token down into pieces, info and all*)
64   | (token, info) :: rest ->
65     context.source_name <- info.Base.token_source_name;
66     context.token_count <- 1 + context.token_count;
67     context.token <- ( token, info );
68     (* Shift the list down by one by referencing
69     the beginning of the rest of the list *)
70     tokenlist := rest;
71     (* return token we care about *)
72     token
73   (* The parser stops calling the tokenizer when
74   it hits EOF: if it reaches the empty list, WE SCREWED UP
       *)
75   | [] -> raise (Errors.MissingEoF)
76   in
77   (* Pass in an empty channel built off a cheap string
78   and then ignore the fuck out of it in our 'tokenizer'
79   internal function *)
80   let program = Parser.program tokenizer (Lexing.
       from_string "") in
81   program
82
83 let analyze program =
84   (* TODO: other important checks and semantic analysis
       here
85   that will create a proper checked program type*)
86   let sem = Semant.check program in
87   sem
```

../source/driver.ml

```
1 (* LePiX - LePiX Language Compiler Implementation
2 Copyright (c) 2016- ThePhD
3
4 Permission is hereby granted, free of charge, to any person
      obtaining a copy of this
5 software and associated documentation files (the "Software"
    ), to deal in the Software
6 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
```

```
41      and endabspos = endp.Lexing.pos_cnum
42      in
43      let create_token token =
44        let t = ( token, { Base.token_source_name =
      sourcename; Base.token_number = tokennumber;
45          Base.token_line_number = line; Base.
      token_line_start = startp.Lexing.pos_bol;
46          Base.token_column_range = (relpos, endrelpos); Base
      .token_character_range = (abspos, endabspos) }
47        ) in
48        t
49      in
50      let rec matcher = function
51        | [] -> raise (Errors.MissingEoF)
52        | Preparser.EOF :: [] -> ( create_token Preparser.EOF
       ) :: tokens
53        | token :: [] -> ( create_token token ) :: ( acc
      lexbuf tokens ( 1 + tokennumber ) )
54        | token :: rest -> ( create_token token ) :: (
      matcher rest )
55      in matcher next_token
56    in acc lexbuf [] 0
57
58  let pre_parse context token_list =
59    (* Keep a reference to the original token list
60    And use that to dereference rather than whatever crap we
       get from
61    the channel *)
62    let tokenlist = ref(token_list) in
63    let tokenizer _ = match !tokenlist with
64    (* Break each token down into pieces, info and all*)
65    | (token, info) :: rest ->
66      context.source_name <- info.Base.token_source_name;
67      context.token_count <- 1 + context.token_count;
68      context.token <- ( token, info );
69      (* Shift the list down by one by referencing
70      the beginning of the rest of the list *)
71      tokenlist := rest;
72      (* return token we care about *)
73      token
74    (* The parser stops calling the tokenizer when
75    it hits EOF: if it reaches the empty list, WE SCREWED UP
       *)
76    | [] -> raise (Errors.MissingEoF)
77    in
```

```
78    (* Pass in an empty channel built off a cheap string
79    and then ignore the fuck out of it in our 'tokenizer'
80    internal function *)
81    let past = Preparser.source tokenizer (Lexing.from_string
        "") in
82    past
83
84  let rec pre_process context source source_text =
85    let source_name = Base.target_to_string source in
86    context.source_name <- source_name;
87    context.source_code <- source_text;
88    let reldir = match source with
89      | Base.Pipe -> ( Sys.getcwd () )
90      | Base.File(f) -> Filename.dirname f
91    in
92    let generate v p = match p with
93      | Preast.Text(s) -> v ^ s
94      | Preast.ImportString(f) -> v ^ "\"" ^ Io.
       read_file_text (Filename.concat reldir f) ^ "\""
95      | Preast.ImportSource(f) -> let realf = (Filename.
       concat reldir f) in
96        let ftext = Io.read_file_text realf in
97        let processedtext = ( pre_process context ( Base.File
       (f) ) ftext ) in
98        v ^ processedtext
99    in
100   let tokenstream = pre_lex source_name ( Lexing.
       from_string source_text ) in
101   (*TODO: debug shit tokens at a later date*)
102   (*print_endline ( Representation.
       preparser_token_list_to_string tokenstream );*)
103   let past = pre_parse context tokenstream in
104   List.fold_left generate "" past
```

../source/predriver.ml

```
1  (* LePiX - LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  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
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Types and routines for the abstract syntax tree and
22 representation of a LePiX program. *)
23
24 type id = string
25
26 type qualified_id = id list
27
28 type builtin_type =
29   | Auto
30   | Void
31   | Bool
32   | Int of int
33   | Float of int
34   | String
35   | Memory
36
37 type constness = bool
38 type referenceness = bool
39
40 type type_qualifier = constness * referenceness
41
42 type type_name =
43   | BuiltinType of builtin_type * type_qualifier
44   | Array of type_name * int * type_qualifier
45   | SizedArray of type_name * int * int list *
       type_qualifier
46   | Function of type_name * type_name list * type_qualifier
47
48 let no_qualifiers = (false, false)
49
```

```
50 let void_t = BuiltinType(Void, no_qualifiers)
51 let string_t = BuiltinType(String, no_qualifiers)
52 let int32_t = BuiltinType(Int(Base.
       default_integral_bit_width), no_qualifiers)
53 let float64_t = BuiltinType(Float(64), no_qualifiers)
54
55 type binding = id * type_name
56
57 let add_const (id, t) = match t with
58   | BuiltinType(bt, tq) -> let ( _, refness ) = tq in
59     (id, BuiltinType(bt, (true, refness)))
60   | Array(tn, d, tq) -> let (_, refness) = tq in
61     (id, Array(tn, d, (true, refness)))
62   | SizedArray(tn, d, il, tq) -> let (_, refness) = tq in
63     (id, SizedArray(tn, d, il, (true, refness)))
64   | Function(tn, pl, tq) -> let (_, refness) = tq in
65     (id, Function(tn, pl, (true, refness)))
66
67 type binary_op = Add | Sub | Mult | Div | Modulo
68   | AddAssign | SubAssign | MultAssign | DivAssign |
      ModuloAssign
69   | Equal | Neq | Less | Leq | Greater | Geq
70   | And | Or
71
72 type prefix_op =
73   | Neg | Not | PreIncrement | PreDecrement
74
75 type postfix_op =
76   PostIncrement | PostDecrement
77
78 type literal =
79   | BoolLit of bool
80   | IntLit of int64 * int
81   | FloatLit of float * int
82   | StringLit of string
83
84 type expression =
85   | Literal of literal
86   | ObjectInitializer of expression list
87   | ArrayInitializer of expression list
88   | QualifiedId of qualified_id
89   | Member of expression * qualified_id
90   | Call of expression * expression list
91   | Index of expression * expression list
92   | BinaryOp of expression * binary_op * expression
```

```
93    | PrefixUnaryOp of prefix_op * expression
94    | Assignment of expression * expression
95    | Noop
96
97  type parallel_expression =
98    | Invocations of expression
99    | ThreadCount of expression
100
101 type variable_definition =
102   | VarBinding of binding * expression
103
104 type general_statement =
105   | ExpressionStatement of expression
106   | VariableStatement of variable_definition
107
108 type control_initializer = general_statement list *
      general_statement
109
110 type statement =
111   | General of general_statement
112   | Return of expression
113   | Break of int
114   | Continue
115   | ParallelBlock of parallel_expression list * statement
      list
116   | AtomicBlock of statement list
117   | IfBlock of control_initializer * statement list
118   | IfElseBlock of control_initializer * statement list *
      statement list
119   | WhileBlock of control_initializer * statement list
120   | ForBlock of general_statement list * expression *
      expression list * statement list
121   | ForByToBlock of expression * expression * expression *
      statement list
122
123 type function_definition =
124   qualified_id (* Name *)
125   * binding list (* Parameters *)
126   * type_name (* Return Type *)
127   * statement list (* Body *)
128
129 type basic_definition =
130   | VariableDefinition of variable_definition
131   | FunctionDefinition of function_definition
132
```

```
133 type import_definition =
134   | LibraryImport of qualified_id
135
136 type definition =
137   | Import of import_definition
138   | Basic of basic_definition
139   | Namespace of qualified_id * definition list
140
141 type program =
142   | Program of definition list
143
144 (* Useful destructuring and common operations *)
145 let binding_type = function
146   | (_, qt) -> qt
147
148 let binding_name = function
149   | (n, _) -> n
```

../source/ast.ml

```
1  (* LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
```

```
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Semantic checking for the Lepix compiler that will
       produce a new
22 SemanticProgram type with things like locals group into a
       single type
23 and type promotions / conversions organized for operators.
       *)
24
25 module StringMap = Map.Make(String)
26
27 type s_prefix_op = Ast.prefix_op
28 type s_binary_op = Ast.binary_op
29 type s_qualified_id = Ast.qualified_id
30 type s_id = Ast.id
31 type s_type_qualifier = Ast.type_qualifier
32 type s_builtin_type = Ast.builtin_type
33
34 type s_type_name =
35   | SBuiltinType of s_builtin_type * s_type_qualifier
36   | SArray of s_type_name * int * s_type_qualifier
37   | SSizedArray of s_type_name * int * int list *
       s_type_qualifier
38   | SFunction of s_type_name * s_type_name list *
       s_type_qualifier
39   | SOverloads of s_type_name list
40   | SAlias of s_qualified_id * s_qualified_id
41
42 let no_qualifiers = Ast.no_qualifiers
43
44 let void_t = SBuiltinType(Ast.Void, Ast.no_qualifiers)
45 let auto_t = SBuiltinType(Ast.Auto, Ast.no_qualifiers)
46 let string_t = SBuiltinType(Ast.String, Ast.no_qualifiers)
47 let bool_t = SBuiltinType(Ast.Bool, Ast.no_qualifiers)
48 let int32_t = SBuiltinType(Ast.Int(32), Ast.no_qualifiers)
49 let int64_t = SBuiltinType(Ast.Int(64), Ast.no_qualifiers)
50 let float64_t = SBuiltinType(Ast.Float(64), Ast.
       no_qualifiers)
51
52 type s_binding = s_id * s_type_name
53
54 type s_literal =
55   | SBoolLit of bool
```

```
56    | SIntLit of int64 * int
57    | SFloatLit of float * int
58    | SStringLit of string
59
60  type s_expression =
61    | SObjectInitializer of s_expression list * s_type_name
62    | SArrayInitializer of s_expression list * s_type_name
63    | SLiteral of s_literal
64    | SQualifiedId of s_qualified_id * s_type_name
65    | SMember of s_expression * s_qualified_id * s_type_name
66    | SCall of s_expression * s_expression list * s_type_name
67    | SIndex of s_expression * s_expression list *
       s_type_name
68    | SBinaryOp of s_expression * s_binary_op * s_expression
       * s_type_name
69    | SPrefixUnaryOp of s_prefix_op * s_expression *
       s_type_name
70    | SAssignment of s_expression * s_expression *
       s_type_name
71    | SNoop
72
73  type s_locals =
74    | SLocals of s_binding list
75
76  type s_parameters =
77    | SParameters of s_binding list
78
79  type s_variable_definition =
80    | SVarBinding of s_binding * s_expression
81
82  type s_general_statement =
83    | SGeneralBlock of s_locals * s_general_statement list
84    | SExpressionStatement of s_expression
85    | SVariableStatement of s_variable_definition
86
87  type s_capture =
88    | SParallelCapture of s_binding list
89
90  type s_control_initializer =
91    | SControlInitializer of s_general_statement *
       s_expression
92
93  type s_parallel_expression =
94    | SInvocations of s_expression
95    | SThreadCount of s_expression
```

```
96
97  type s_statement =
98    | SBlock of s_locals * s_statement list
99    | SGeneral of s_general_statement
100   | SReturn of s_expression
101   | SBreak of int
102   | SContinue
103
104   | SIfBlock of s_control_initializer (* Init statements
        for an if block *)
105     * s_statement (* If code *)
106
107   | SIfElseBlock of s_control_initializer (* Init
        statements for an if-else block *)
108     * s_statement (* If code *)
109     * s_statement (* Else code *)
110
111   | SWhileBlock of s_control_initializer (* Init statements
        plus ending conditional for a while loop *)
112     * s_statement (* code inside the while block, locals
        and statements *)
113
114   | SForBlock of s_control_initializer  (* Init statements
        plus ending conditional for a for loop *)
115     * s_expression list (* Post-loop expressions (increment
        /decrement) *)
116     * s_statement (* Code inside *)
117
118   | SParallelBlock of s_parallel_expression list (*
        Invocation parameters passed to kickoff function *)
119     * s_capture (* Capture list: references to outside
        variables *)
120     * s_statement (* Locals and their statements *)
121
122   | SAtomicBlock of s_statement (* code in the atomic block
        *)
123
124
125 type s_function_definition = {
126   func_name : s_qualified_id;
127   func_parameters : s_parameters;
128      func_return_type : s_type_name;
129   func_body : s_statement list;
130 }
131
```

```
132 type s_basic_definition =
133   | SVariableDefinition of s_variable_definition
134   | SFunctionDefinition of s_function_definition
135
136 type s_builtin_library =
137   | Lib
138
139 let builtin_library_names = [
140   ( "lib", Lib )
141 ]
142
143 type s_loop =
144   | SFor
145   | SWhile
146
147 type s_module =
148   | SCode of string
149   | SDynamic of string
150   | SBuiltin of s_builtin_library
151
152 type s_definition =
153   | SBasic of s_basic_definition
154
155 type s_attributes = {
156     attr_parallelism : bool;
157   attr_arrays : int;
158   attr_strings : bool;
159 }
160
161 type s_environment = {
162   env_usings : string list;
163   env_symbols : s_type_name StringMap.t;
164   env_definitions : s_type_name StringMap.t;
165   env_imports : s_module list;
166   env_loops : s_loop list;
167 }
168
169 type s_program =
170   | SProgram of s_attributes * s_environment * s_definition
      list
171
172 (* Helping functions *)
173 let rec coerce_type_name_of_s_expression injected =
    function
174   | SObjectInitializer(a, _) -> SObjectInitializer(a,
```

```
      injected)
175   | SArrayInitializer(a, _) -> SArrayInitializer(a,
      injected)
176   | SQualifiedId(a, _) -> SQualifiedId(a, injected)
177   | SMember(a, b, _) -> SMember(a, b, injected)
178   | SCall(a, b, _) -> SCall(a, b, injected)
179   | SIndex(a, b, _) -> SIndex(a, b, injected)
180   | SBinaryOp(a, b, c, _) -> SBinaryOp(a, b, c, injected)
181   | SPrefixUnaryOp(a, b, _) -> SPrefixUnaryOp(a, b,
      injected)
182   | SAssignment(a, b, _) -> SAssignment(a, b, injected)
183   | e -> e
184
185 let unqualify = function
186   | SBuiltinType(bt, _) -> SBuiltinType(bt, no_qualifiers)
187   | SArray(tn, d, _) -> SArray(tn, d, no_qualifiers)
188   | SSizedArray(tn, d, il, _) -> SSizedArray(tn, d, il,
      no_qualifiers)
189   | SFunction(tn, pl, _) -> SFunction(tn, pl, no_qualifiers
      )
190   | t -> t
191
192 let string_of_qualified_id qid =
193   ( String.concat "." qid )
194
195 let parameter_bindings = function
196   | SParameters( bl ) -> bl
197
198 let type_name_of_s_literal = function
199   | SBoolLit(_) -> bool_t
200   | SIntLit(_, b) -> SBuiltinType( Ast.Int(b),
      no_qualifiers )
201   | SFloatLit(_,b ) -> SBuiltinType( Ast.Float(b),
      no_qualifiers )
202   | SStringLit(_) -> string_t
203
204 let rec type_name_of_s_expression = function
205   | SObjectInitializer(_, t) -> t
206   | SArrayInitializer(_, t) -> t
207   | SLiteral(lit) -> type_name_of_s_literal lit
208   | SQualifiedId(_, t) -> t
209   | SMember(_, _, t) -> t
210   | SCall(_, _, t) -> t
211   | SIndex(_, _, t) -> t
212   | SBinaryOp(_, _, _, t) -> t
```

```
213     | SPrefixUnaryOp(_, _, t) -> t
214     | SAssignment(_, _, t) -> t
215     | SNoop -> void_t
216
217   let return_type_name = function
218     | SFunction(rt,_, _) -> rt
219     | t -> t
220
221   let args_type_name = function
222     | SFunction(_, args, _) -> args
223     | t -> []
224
225   let mangled_name_of_type_qualifier = function
226     | (_, referencess) -> if referencess then "p!" else "!"
227
228   let type_name_of_s_function_definition fdef =
229     let bl = parameter_bindings fdef.func_parameters in
230     let argst = List.map ( fun (_, t) -> t ) bl in
231     let rt = fdef.func_return_type in
232     SFunction( rt, argst, no_qualifiers )
233
234   let mangled_name_of_builtin_type = function
235     | Ast.Void -> "v"
236     | Ast.Auto -> "a"
237     | Ast.Bool -> "b"
238     | Ast.Int(n) -> "i" ^ string_of_int n
239     | Ast.Float(n) -> "f" ^ string_of_int n
240     | Ast.String -> "s"
241     | Ast.Memory -> "m"
242
243   let rec mangled_name_of_s_type_name = function
244     | SBuiltinType( bt, tq ) ->
        mangled_name_of_type_qualifier tq ^
        mangled_name_of_builtin_type bt
245     | SArray( tn, dims, tq ) ->
        mangled_name_of_type_qualifier tq ^ "a" ^ string_of_int
        dims ^ ";" ^ mangled_name_of_s_type_name tn
246     | SSizedArray( tn, dims, sizes, tq ) ->
        mangled_name_of_type_qualifier tq ^ "a" ^ string_of_int
        dims ^ ";" ^ mangled_name_of_s_type_name tn
247     | 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
248     | _ -> "UNSUPPORTED"
```

```
249
250 let mangle_name_args qid tnl =
251   string_of_qualified_id qid ^
252   if ( List.length tnl ) > 0 then
253     "_" ^ ( String.concat "_" ( List.map
     mangled_name_of_s_type_name tnl ) )
254   else
255     ""
256 let mangle_name qid = function
257   | SFunction(rt, pl, tq) -> mangle_name_args qid pl
258   | _ -> string_of_qualified_id qid
```

../source/semast.ml

```
1 (* LePiX Language Compiler Implementation
2 Copyright (c) 2016- ThePhD
3
4 Permission is hereby granted, free of charge, to any person
     obtaining a copy of this
5 software and associated documentation files (the "Software"
    ), to deal in the Software
6 without restriction, including without limitation the
    rights to use, copy, modify,
7 merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
8 permit persons to whom the Software is furnished to do so,
    subject to the following
9 conditions:
10
11 The above copyright notice and this permission notice shall
     be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
    MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
    , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
    CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Contains routines for string-ifying various parts of the
22 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"
```

```
66    | Parser.OR -> "OR"
67    | Parser.NOT -> "NOT"
68    | Parser.DOT -> "DOT"
69    | Parser.AMP -> "AMPERSAND"
70    | Parser.COLON -> "COLON"
71    | Parser.PARALLEL -> "PARALLEL"
72    | Parser.INVOCATIONS -> "INVOCATIONS"
73    | Parser.THREADCOUNT -> "THREADCOUNT"
74    | Parser.ATOMIC -> "ATOMIC"
75    | Parser.VAR -> "VAR"
76    | Parser.LET -> "LET"
77    | Parser.CONST -> "CONST"
78    | Parser.FUN -> "FUN"
79    | Parser.NAMESPACE -> "NAMESPACE"
80    | Parser.IF -> "IF"
81    | Parser.ELSE -> "ELSE"
82    | Parser.FOR -> "FOR"
83    | Parser.TO -> "TO"
84    | Parser.BY -> "BY"
85    | Parser.WHILE -> "WHILE"
86    | Parser.RETURN -> "RETURN"
87    | Parser.INT(b) -> "INT" ^ string_of_int b
88    | Parser.FLOAT(b) -> "FLOAT" ^ string_of_int b
89    | Parser.BOOL -> "BOOL"
90    | Parser.STRING -> "STRING"
91    | Parser.VOID -> "VOID"
92    | Parser.AUTO -> "AUTO"
93    | Parser.MEMORY -> "MEMORY"
94    | Parser.TRUE -> "TRUE"
95    | Parser.FALSE -> "FALSE"
96    | Parser.BREAK -> "BREAK"
97    | Parser.CONTINUE -> "CONTINUE"
98    | Parser.IMPORT -> "IMPORT"
99    | Parser.STRINGLITERAL(s) -> "STRINGLITERAL(" ^ s ^ ")"
100   | Parser.INTLITERAL(i) -> "INTLITERAL(" ^ Num.
      string_of_num i ^ ")"
101   | Parser.FLOATLITERAL(f) -> "FLOATLITERAL(" ^ Num.
      string_of_num f ^ ")"
102   | Parser.ID(s) -> "ID(" ^ s ^ ")"
103   | Parser.EOF -> "EOF"
104
105 let token_range_to_string (x, y) =
106   let range_is_wide = ( y - x > 1 ) in
107   if range_is_wide then
108     ( string_of_int x ^ "-" ^ string_of_int y,
```

```
      range_is_wide )
109   else
110     ( string_of_int x, range_is_wide )
111
112 let token_source_to_string t =
113   let (s, _) = token_range_to_string t.Base.
      token_column_range in
114   string_of_int t.Base.token_line_number
115   ^ ":"
116   ^ s
117
118 let preparser_token_list_to_string token_list =
119   let rec helper = function
120   | (token, pos) :: tail ->
121     "[" ^ ( preparser_token_to_string token ) ^ ":"
122     ^ token_source_to_string pos ^ "] "
123     ^ helper tail
124   | [] -> "\n" in helper token_list
125
126 let parser_token_list_to_string token_list =
127   let rec helper = function
128   | (token, pos) :: tail ->
129     "[" ^ ( parser_token_to_string token ) ^ ":"
130     ^ token_source_to_string pos ^ "] "
131     ^ helper tail
132   | [] -> "\n" in helper token_list
133
134
135 (* Program types:
136 dumping and Pretty-Printing *)
137
138 let string_of_id i = i
139
140 let string_of_qualified_id qid = ( String.concat "." ( List
      .map string_of_id qid ) )
141
142 let string_of_binary_op = function
143   | Ast.Add -> "+"
144   | Ast.Sub -> "-"
145   | Ast.Mult -> "*"
146   | Ast.Div -> "/"
147   | Ast.Modulo -> "%"
148   | Ast.AddAssign -> "+="
149   | Ast.SubAssign -> "-="
150   | Ast.MultAssign -> "*="
```

```
151     | Ast.DivAssign -> "/="
152     | Ast.ModuloAssign -> "%="
153     | Ast.Equal -> "=="
154     | Ast.Neq -> "!="
155     | Ast.Less -> "<"
156     | Ast.Leq -> "<="
157     | Ast.Greater -> ">"
158     | Ast.Geq -> ">="
159     | Ast.And -> "&&"
160     | Ast.Or -> "||"
161
162   let string_of_unary_op = function
163     | Ast.Neg -> "-"
164     | Ast.Not -> "!"
165     | Ast.PreDecrement -> "--"
166     | Ast.PreIncrement -> "++"
167
168   let rec string_of_expression = function
169     | Ast.Literal(Ast.IntLit(l, _)) -> Int64.to_string l
170     | Ast.Literal(Ast.BoolLit(true)) -> "true"
171     | Ast.Literal(Ast.BoolLit(false)) -> "false"
172     | Ast.Literal(Ast.StringLit(s)) -> "\"" ^ s ^ "\""
173     | Ast.Literal(Ast.FloatLit(f, _)) -> string_of_float f
174     | Ast.QualifiedId(qid) -> string_of_qualified_id qid
175     | Ast.BinaryOp(e1, o, e2) ->
176       string_of_expression e1 ^ " " ^ string_of_binary_op o ^
        " " ^ string_of_expression e2
177     | Ast.PrefixUnaryOp(o, e) -> string_of_unary_op o ^
      string_of_expression e
178     | Ast.Index(e, l) -> string_of_expression e ^ "[" ^ (
      String.concat ", " (List.map string_of_expression l)) ^
      "]"
179     | Ast.Member(e, qid) -> string_of_expression e ^ "." ^
      string_of_qualified_id qid
180     | Ast.Assignment(e1, e2) -> string_of_expression e1 ^ " =
        " ^ string_of_expression e2
181     | Ast.Call(e, el) ->
182       string_of_expression e ^ "(" ^ String.concat ", " (List
      .map string_of_expression el) ^ ")"
183     | Ast.Noop -> "{ noop }"
184     | Ast.ArrayInitializer(el) -> "[ " ^ String.concat ", " (
      List.map string_of_expression el) ^ " ]"
185     | Ast.ObjectInitializer(el) -> "{ " ^ String.concat ", "
      (List.map string_of_expression el) ^ " }"
186
```

```
187 let string_of_parallel_expression = function
188   | Ast.ThreadCount(e) -> "thread_count = " ^
      string_of_expression e
189   | Ast.Invocations(e) -> "invocations = " ^
      string_of_expression e
190
191 let rec string_of_expression_list el =
192   String.concat ", " ( List.map string_of_expression el )
193
194 let rec string_of_builtin_type = function
195   | Ast.Auto -> "auto"
196   | Ast.Bool -> "bool"
197   | Ast.Int(b) -> "int" ^ string_of_int b
198   | Ast.Float(b) -> "float" ^ string_of_int b
199   | Ast.String -> "string"
200   | Ast.Memory -> "memory"
201   | Ast.Void -> "void"
202
203 let string_of_type_qualifier = function
204   | (c, r) -> ( if c then "const" else "" ) ^ ( if r then "
     &" else "" )
205
206 let rec string_of_type_name tn =
207   let tqual tq =
208     let s = string_of_type_qualifier tq in
209     if s = "" then "" else s ^ " "
210   in match tn with
211   | Ast.BuiltinType(t, tq) -> tqual tq ^
      string_of_builtin_type t
212   | Ast.Array(t, d, tq) -> tqual tq ^ string_of_type_name t
      ^ ( String.make d '[' ) ^ ( String.make d ']' )
213   | 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 ']' )
214   | Ast.Function(r, args, tq) -> tqual tq ^ "(" ^  ( String
      .concat ", " ( List.map ( fun v -> string_of_type_name v
      ) args ) ) ^ ")" ^ string_of_type_name r
215
216 let string_of_binding = function
217   | (n, t) -> n ^ " : " ^ string_of_type_name t
218
219 let string_of_variable_definition = function
220   | Ast.VarBinding(b, Ast.Noop) -> "var " ^
      string_of_binding b
```

```
221     | Ast.VarBinding(b, e) -> "var " ^ string_of_binding b ^
        " = " ^ string_of_expression e
222
223   let string_of_general_statement = function
224     | Ast.ExpressionStatement(e) -> string_of_expression e
225     | Ast.VariableStatement(v) ->
        string_of_variable_definition v
226
227   let string_of_condition_initializer = function
228     | (il, cond) -> ( String.concat "; " ( List.map
        string_of_general_statement il ) )
229     ^ ( if ( List.length il ) > 0 then ";" else "" )
230     ^ ( string_of_general_statement cond )
231
232   let rec string_of_statement s =
233     let string_of_statement_list sl = String.concat "" (List.
        map string_of_statement sl) in
234     match s with
235       | Ast.General(b) -> string_of_general_statement b ^ ";\
        n";
236       | Ast.Return(expr) -> "return " ^ string_of_expression
        expr ^ ";\n";
237       | Ast.IfBlock(ilcond, s) -> "if (" ^
        string_of_condition_initializer ilcond ^ ")"
238         ^ "{\n" ^  string_of_statement_list s ^ "}\n"
239       | Ast.IfElseBlock(ilcond, s, s2) -> "if (" ^
        string_of_condition_initializer ilcond ^ ")"
240         ^ "{\n" ^  string_of_statement_list s ^ "}\n"
241         ^ "else {\n" ^ string_of_statement_list s2 ^ "}\n"
242       | Ast.ForBlock(gsl, cond, incrl, sl) -> "for (" ^ (
        String.concat ", " (List.map string_of_general_statement
        gsl) ) ^ "; "
243         ^ string_of_expression cond ^ "; "
244         ^ string_of_expression_list incrl  ^ ") {\n" ^
        string_of_statement_list sl ^ "}\n"
245       | 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"
246       | Ast.WhileBlock(ilcond, s) -> "while (" ^
        string_of_condition_initializer ilcond ^ ") {\n"
247         ^ string_of_statement_list s ^ "}\n"
248       | Ast.Break(n) -> ( if n == 1 then "break" else "break"
        ^ string_of_int n ) ^ ";\n"
249       | Ast.Continue -> "continue;\n"
```

```
250     | Ast.ParallelBlock(pel,sl) -> "parallel(" ^ (String.
      concat ", " (List.map string_of_parallel_expression pel)
      ) ^ " ) {"
251       ^ "\n" ^ string_of_statement_list sl ^ "}\n"
252     | Ast.AtomicBlock(sl) -> "atomic {\n" ^
      string_of_statement_list sl ^ "}\n"
253
254 let string_of_statement_list sl =
255   String.concat "" (List.map string_of_statement sl)
256
257 let string_of_function_definition = function
258   | ( name, parameters, return_type, body) ->
259     "fun " ^ string_of_qualified_id name
260       ^ "(" ^ (String.concat ", " (List.map
      string_of_binding parameters)) ^ ") : "
261       ^ string_of_type_name return_type  ^ " {\n"
262       ^ string_of_statement_list body
263       ^ "}\n"
264
265 let rec string_of_basic_definition = function
266   | Ast.FunctionDefinition(fdef) ->
      string_of_function_definition fdef
267   | Ast.VariableDefinition(vdef) ->
      string_of_variable_definition vdef ^ ";\n"
268
269 let string_of_import_definition = function
270   | Ast.LibraryImport(qid) -> "import " ^
      string_of_qualified_id qid ^ "\n"
271
272 let rec string_of_definition = function
273   | Ast.Import(idef) -> string_of_import_definition idef
274   | Ast.Basic(bdef) -> string_of_basic_definition bdef
275   | Ast.Namespace(qid, defs) -> "namespace " ^
      string_of_qualified_id qid ^ " {\n"
276     ^ (String.concat "" (List.map string_of_definition defs
      ) ) ^ "}\n"
277
278 let string_of_program = function
279   | Ast.Program(p) -> let s = (String.concat "" (List.map
      string_of_definition p) ) in
280   Base.brace_tabulate s 0
281
282 (* Semantic Program types:
283 dumping and pretty printing *)
284
```

```
285 let rec string_of_s_type_name tn =
286   let tqual tq =
287     let s = string_of_type_qualifier tq in
288     if s = "" then "" else s ^ " "
289   in match tn with
290   | Semast.SBuiltinType(t, tq) -> tqual tq ^
      string_of_builtin_type t
291   | Semast.SArray(t, d, tq) -> tqual tq ^
      string_of_s_type_name t ^ ( String.make d '[' ) ^ (
      String.make d ']' )
292   | 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 ']' )
293   | 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
294   | Semast.SOverloads(fl) -> "overloads["
295     ^ ( String.concat ", " ( List.map string_of_s_type_name
      fl ) )
296     ^ "]"
297   | Semast.SAlias(target, source) -> "using " ^
      string_of_qualified_id target ^ " -> " ^
      string_of_qualified_id source
298
299 let string_of_s_binding = function
300   | (n, t) -> n ^ " : " ^ string_of_s_type_name t
301
302 let string_of_s_locals = function
303   | Semast.SLocals(bl) -> if ( List.length bl < 1 ) then ""
      else ( String.concat ";\n" ( List.map
      string_of_s_binding bl ) ) ^ ";"
304
305 let string_of_s_literal = function
306   | Semast.SBoolLit(b) -> string_of_bool b
307   | Semast.SIntLit(i, _) -> Int64.to_string i
308   | Semast.SFloatLit(f, _) -> string_of_float f
309   | Semast.SStringLit(s) -> "\"" ^ s ^ "\""
310
311 let rec string_of_s_expression = function
312   | Semast.SObjectInitializer(el, tn) ->
313     string_of_s_type_name tn ^ "{ "
314     ^ String.concat ", " ( List.map string_of_s_expression
      el )
```

```
315       ^ " }"
316     | Semast.SArrayInitializer(el, tn) ->
317       string_of_s_type_name tn ^ "[ "
318       ^ String.concat ", " ( List.map string_of_s_expression
          el )
319       ^ " ]"
320     | Semast.SLiteral(l) -> string_of_s_literal l
321     | Semast.SQualifiedId(qid, tn) -> "[[ " ^
          string_of_s_type_name tn ^ " ]] " ^
          string_of_qualified_id qid
322     | Semast.SMember(e, qid, tn) -> "[[ " ^
          string_of_s_type_name tn ^ " ]] " ^
          string_of_s_expression e ^ "." ^ string_of_qualified_id
          qid
323     | 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 ) ) ^ "
           )"
324     | 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 ) ) ^ " ]"
325     | 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
326     | Semast.SPrefixUnaryOp(op, r, tn) -> "[[ " ^
          string_of_s_type_name tn ^ " ]] " ^ string_of_unary_op
          op ^ string_of_s_expression r
327     | Semast.SAssignment(l, r, tn) -> "[[ " ^
          string_of_s_type_name tn ^ " ]] " ^
          string_of_s_expression l ^ " = " ^
          string_of_s_expression r
328     | Semast.SNoop -> "(noop)"
329
330   let string_of_s_capture = function
331     | Semast.SParallelCapture(bl) -> let capturecount = List.
          length bl in
332       if capturecount == 0 then "[[no captures]]\n" else
333       "[[captures]] { " ^ ( String.concat ", " ( List.map
          string_of_s_binding bl ) )
334       ^ " }\n"
335
336   let string_of_s_variable_definition = function
337     | Semast.SVarBinding(b, e) -> "var " ^
```

```
    string_of_s_binding b ^ " = " ^ string_of_s_expression e
338
339 let rec string_of_s_general_statement = function
340   | Semast.SGeneralBlock(locals, gsl) -> "{\n" ^
      string_of_s_locals locals ^ "\n" ^ ( String.concat ";\n"
       (List.map string_of_s_general_statement gsl) ) ^ "\n}"
341   | Semast.SExpressionStatement(sexpr) ->
      string_of_s_expression sexpr
342   | Semast.SVariableStatement(v) ->
      string_of_s_variable_definition v
343
344 let string_of_s_general_statement_list gsl =
345   String.concat ";\n" (List.map
      string_of_s_general_statement gsl)
346
347 let string_of_s_parallel_expression = function
348   | Semast.SInvocations(e) -> string_of_s_expression e
349   | Semast.SThreadCount(e) -> string_of_s_expression e
350
351 let rec string_of_s_statement s =
352   let initializer_begin = function
353     | Semast.SGeneralBlock(locals, gsl) ->
354       let precount = List.length gsl in
355       if precount > 1 then
356         "{ "
357         ^ string_of_s_locals locals
358         ^ "\n" ^ string_of_s_general_statement_list gsl
359         ^ "\n"
360       else
361         ""
362     | _ -> ""
363   in
364   let initializer_end = function
365     | Semast.SGeneralBlock(locals, gsl) ->
366       let precount = List.length gsl in
367       if precount > 1 then
368         "}\n"
369       else
370         ""
371     | _ -> ""
372   in match s with
373   | Semast.SBlock(locals, sl) -> "{\n" ^ string_of_s_locals
      locals ^ "\n" ^ ( String.concat "\n" (List.map
     string_of_s_statement sl) ) ^ "\n}\n"
374   | Semast.SGeneral(g) ->  ( string_of_s_general_statement
```

```
      g ) ^ ";"
375   | Semast.SReturn(sexpr) -> "return " ^
      string_of_s_expression sexpr ^ ";"
376   | Semast.SBreak(n) -> if n < 2 then "break;" else "break
      " ^ string_of_int n ^ ";"
377   | Semast.SContinue -> "continue;"
378   | Semast.SAtomicBlock(s) -> "atomic {"
379     ^ string_of_s_statement s
380     ^ "}\n"
381   | Semast.SParallelBlock( pel, captures, s) ->
382     "parallel(" ^ (String.concat ", " (List.map
      string_of_s_parallel_expression pel)) ^ " ) {"
383     ^ "\n" ^ string_of_s_capture captures
384     ^ "\n" ^ string_of_s_statement s
385     ^ "}\n"
386   | Semast.SIfBlock(Semast.SControlInitializer(inits, cond)
      , s) ->
387     initializer_begin inits
388     ^ "if (" ^ string_of_s_expression cond ^ ") {"
389     ^ "\n" ^ string_of_s_statement s
390     ^ "}\n"
391     ^ initializer_end inits
392   | Semast.SIfElseBlock(Semast.SControlInitializer(inits,
      cond), is, es) ->
393     initializer_begin inits
394     ^ "if (" ^ string_of_s_expression cond ^ ") {"
395     ^ "\n" ^ string_of_s_statement is
396     ^ "}\n"
397     ^ "else {"
398     ^ "\n" ^ string_of_s_statement es
399     ^ "}\n"
400     ^ initializer_end inits
401   | Semast.SWhileBlock(Semast.SControlInitializer(inits,
      cond), s) ->
402     initializer_begin inits
403     ^ "while (" ^ string_of_s_expression cond ^ ") {"
404     ^ "\n" ^ string_of_s_statement s
405     ^ "}\n"
406     ^ initializer_end inits
407   | Semast.SForBlock(Semast.SControlInitializer(inits, cond
      ), increxprl, s) ->
408     let incrl = String.concat ", " ( List.map
      string_of_s_expression increxprl ) in
409     initializer_begin inits
410     ^ "for (;" ^ string_of_s_expression cond ^ "; " ^ incrl
```

```
      ^ ") {"
411   ^ "\n" ^ string_of_s_statement s
412   ^ "}\n"
413   ^ initializer_end inits
414
415 let string_of_s_statement_list sl =
416   ( String.concat "\n" ( List.map string_of_s_statement sl
      ) ) ^ "\n"
417
418 let string_of_s_block = function
419   | (locals, sl) -> "{\n" ^ string_of_s_locals locals
420     ^ "\n" ^ string_of_s_statement_list sl
421     ^ "\n}\n"
422
423 let string_of_s_parameters = function
424   | Semast.SParameters(parameters) -> String.concat ", " (
      List.map string_of_s_binding parameters)
425
426 let string_of_s_function_definition f =
427   "fun " ^ string_of_qualified_id f.Semast.func_name
428   ^ "(" ^ string_of_s_parameters f.Semast.func_parameters ^
      ") : "
429   ^ string_of_s_type_name f.Semast.func_return_type  ^ " {\
      n"
430   ^ string_of_s_statement_list f.Semast.func_body
431   ^ "}\n"
432
433 let string_of_s_basic_definition = function
434   | Semast.SVariableDefinition(v) ->
      string_of_s_variable_definition v
435   | Semast.SFunctionDefinition(f) ->
      string_of_s_function_definition f
436
437 let string_of_s_builtin_library = function
438   | Semast.Lib -> "lib"
439
440 let string_of_s_module = function
441   | Semast.SCode(s) -> "import [[code]] " ^ s
442   | Semast.SDynamic(s) -> "import [[dynamic]] " ^ s
443   | Semast.SBuiltin(bltin) -> "import [[builtin]] " ^
      string_of_s_builtin_library bltin
444
445 let rec string_of_s_definition = function
446   | Semast.SBasic(b) -> string_of_s_basic_definition b ^ "\
      n"
```

```
447
448 let string_of_s_program = function
449   | Semast.SProgram( attr, env, sdl ) ->
450   let symbolacc k tn l =
451     let entry = ( string_of_s_type_name tn ) ^ " | " ^ k in
452     entry :: l
453   and importacc m =
454     string_of_s_module m
455   in
456   let implist = List.map importacc env.Semast.env_imports
457   and symbollist = StringMap.fold symbolacc env.Semast.
      env_symbols []
458   and defsymbollist = StringMap.fold symbolacc env.Semast.
      env_definitions []
459   in
460   let i = "imports:\n\t" ^ ( String.concat "\n\t" implist )
461   and s = "symbols:\n\t" ^ ( String.concat "\n\t"
      symbollist )
462   and d = "code symbols:\n\t" ^ ( String.concat "\n\t"
      defsymbollist )
463   and a = "strings: " ^ string_of_bool attr.Semast.
      attr_strings
464     ^ "\narrays: " ^ string_of_int attr.Semast.attr_arrays
465     ^ "\nparallelism: " ^ string_of_bool attr.Semast.
      attr_parallelism
466   in
467   let p = String.concat "" (List.map string_of_s_definition
      sdl) in
468     Base.brace_tabulate ( a ^ "\n\n" ^ i ^ "\n\n" ^ s ^ "\n
    \n" ^ d ^ "\n\n" ^ p ) 0
```

../source/representation.ml

```
1 (* LePiX - LePiX Language Compiler Implementation
2 Copyright (c) 2016- ThePhD
3
4 Permission is hereby granted, free of charge, to any person
      obtaining a copy of this
5 software and associated documentation files (the "Software"
    ), to deal in the Software
6 without restriction, including without limitation the
    rights to use, copy, modify,
7 merge, publish, distribute, sublicense, and/or sell copies
    of the Software, and to
8 permit persons to whom the Software is furnished to do so,
    subject to the following
9 conditions:
```

```
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
     KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
     MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
     THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
     , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
     CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Ocamllex Scanner for LePiX Preprocessor *)
22
23 let whitespace = [' ' '\t' '\r']
24 let newline = ['\n']
25
26 rule token = parse
27 | newline as c { Lexing.new_line lexbuf; let b = Buffer.
       create 1024 in Buffer.add_char b c; sub_token b lexbuf }
28 | '#'          { let t = Preparser.HASH in t :: hash_token
       lexbuf  }
29 | _ as c       { let b = Buffer.create 1024 in Buffer.
       add_char b c; sub_token b lexbuf }
30 | eof          { [Preparser.EOF] }
31
32 and sub_token text_buf = parse
33 | newline as c  { Lexing.new_line lexbuf; Buffer.add_char
       text_buf c; sub_token text_buf lexbuf }
34 | '#'           { let s = Buffer.contents text_buf in let t
       = Preparser.TEXT(s) in let l = Preparser.HASH :: (
       hash_token lexbuf ) in t :: l  }
35 | _ as c        { Buffer.add_char text_buf c; sub_token
       text_buf lexbuf }
36 | eof           { let s = Buffer.contents text_buf in let t
       = Preparser.TEXT(s) in [t; Preparser.EOF] }
37
38 and hash_token = parse
39 | newline    { Lexing.new_line lexbuf; token lexbuf }
40 | whitespace { hash_token lexbuf }
```

```
41 | '"'        { let b = ( Buffer.create 128 ) in
       string_literal b lexbuf }
42 | "import"    { Preparser.IMPORT :: hash_token lexbuf }
43 | "string"    { Preparser.STRING :: hash_token lexbuf }
44 | eof         { [Preparser.EOF] }
45 | _ as c      { raise (Errors.UnknownCharacter(String.make 1
       c, ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p
       lexbuf ) )) }
46
47 and string_literal string_buf = parse
48 | newline as c { Lexing.new_line lexbuf; Buffer.add_char
       string_buf c; string_literal string_buf lexbuf }
49 | '"'          { let sl = Preparser.STRINGLITERAL( Buffer.
       contents string_buf ) in [sl] }
50 | "\\\"" as s  { Buffer.add_string string_buf s;
       string_literal string_buf lexbuf }
51 | _ as c       { Buffer.add_char string_buf c;
       string_literal string_buf lexbuf }
```

../source/prescanner.mll

```
1  %{
2  (* LePiX - LePiX Language Compiler Implementation
3  Copyright (c) 2016- ThePhD
4
5  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
6  software and associated documentation files (the "Software"
       ), to deal in the Software
7  without restriction, including without limitation the
       rights to use, copy, modify,
8  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
9  permit persons to whom the Software is furnished to do so,
       subject to the following
10 conditions:
11
12 The above copyright notice and this permission notice shall
       be included in all copies
13 or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
16 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
17 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
```

```
18 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
19 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
20 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
21
22 (* Parser for the LePiX preprocessor: compatible with both
       ocamlyacc and
23 menhir, as we have developed against both for testing
       purposes. *)
24
25 %}
26
27 %token HASH
28 %token IMPORT STRING
29 %token <string> TEXT
30 %token <string> STRINGLITERAL
31 %token EOF
32
33 %start source
34 %type<Preast.pre_source> source
35 %%
36
37 blob:
38 | HASH IMPORT STRINGLITERAL { Preast.ImportSource($3) }
39 | HASH IMPORT STRING STRINGLITERAL { Preast.ImportString($4
       ) }
40 | TEXT { Preast.Text($1) }
41
42 blob_list: { [] }
43 | blob_list blob { $2 :: $1 }
44
45 source:
46 | blob_list EOF { List.rev $1 }
```

../source/preparser.mly

```
1 (* LePiX - LePiX Language Compiler Implementation
2 Copyright (c) 2016- ThePhD
3
4 Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5 software and associated documentation files (the "Software"
       ), to deal in the Software
6 without restriction, including without limitation the
       rights to use, copy, modify,
7 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 }
```

```
44 | '{'          { LBRACE }
45 | '}'          { RBRACE }
46 | '['        { LSQUARE }
47 | ']'        { RSQUARE }
48 | ';'          { SEMI }
49 | ':'        { COLON }
50 | ','          { COMMA }
51 | '+'          { PLUS }
52 | '-'          { MINUS }
53 | '*'          { TIMES }
54 | '/'          { DIVIDE }
55 | "+="         { PLUSASSIGN }
56 | "-="         { MINUSASSIGN }
57 | "*="         { TIMESASSIGN }
58 | "/="         { DIVIDEASSIGN }
59 | "%="         { MODULOASSIGN }
60 | "++"         { PLUSPLUS }
61 | "--"         { MINUSMINUS }
62 | "%"          { MODULO }
63 | '='          { ASSIGN }
64 | '&'          { AMP }
65 | "=="         { EQ }
66 | "!="         { NEQ }
67 | '<'          { LT }
68 | "<="         { LEQ }
69 | ">"          { GT }
70 | ">="         { GEQ }
71 | "&&"         { AND }
72 | '.'          { DOT }
73 | '&'          { AMP }
74 | "||"         { OR }
75 | "!"          { NOT }
76 | "if"         { IF }
77 | "else"       { ELSE }
78 | "for"        { FOR }
79 | "while"      { WHILE }
80 | "by"         { BY }
81 | "to"         { TO }
82 | "return"     { RETURN }
83 | "auto"       { AUTO }
84 | "int" ((decimal_digit+)? as s) { let bits = if s = ""
     then Base.default_integral_bit_width else (
     int_of_string s ) in INT(bits) }
85 | "float" ((decimal_digit+)? as s) { let bits = if s = ""
     then Base.default_floating_bit_width else (
```

```
      int_of_string s ) in FLOAT(bits) }
86  | "bool"    { BOOL }
87  | "string"  { STRING }
88  | "void"    { VOID }
89  | "memory"  { MEMORY }
90  | "true"    { TRUE }
91  | "false"   { FALSE }
92  | "var"     { VAR }
93  | "let"     { LET }
94  | "const"   { CONST }
95  | "fun"      { FUN }
96  | "parallel" { PARALLEL }
97  | "break" { BREAK }
98  | "continue" { CONTINUE }
99  | "invocations" { INVOCATIONS }
100 | "thread_count" { THREADCOUNT }
101 | "atomic" { ATOMIC }
102 | "namespace" { NAMESPACE }
103 | "import" { IMPORT }
104 | '"'      { string_literal ( Buffer.create 128 ) lexbuf }
105 | decimal_digit+ as lxm { INTLITERAL(Num.num_of_string lxm)
      }
106 | "0c" | "0C" { octal_int_literal lexbuf }
107 | "0x" | "0X" { hex_int_literal lexbuf }
108 | "0b" | "0B" { binary_int_literal lexbuf }
109 | ( "0n" | "0N" ) ( decimal_digit+ as b ) ("n" | "N") {
      n_int_literal (int_of_string b) lexbuf }
110 | '.' ['0'-'9']+ ('e' ('+'|'-')? ['0'-'9']+)? as s {
      FLOATLITERAL(Polyfill.num_of_string s ) }
111 | ['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 ) )) }
112 | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as s { ID
      (s) }
113 | eof { EOF }
114 | _ as c { raise (Errors.UnknownCharacter(String.make 1 c,
      ( Lexing.lexeme_start_p lexbuf, Lexing.lexeme_end_p
      lexbuf ) )) }
115
116
117 and octal_int_literal = parse
118 | 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 }
```

```
143
144 and multi_comment level = parse
145 | newline { Lexing.new_line lexbuf; multi_comment level
      lexbuf }
146 | "*/" { if level = 0 then token lexbuf else multi_comment
      (level-1) lexbuf }
147 | "/*" { multi_comment (level+1) lexbuf }
148 | _    { multi_comment level lexbuf }
149
150
151 and single_comment = parse
152 | newline { Lexing.new_line lexbuf; token lexbuf }
153 | _    { single_comment lexbuf }
```

../source/scanner.mll

```
1  %{
2  (* LePiX - LePiX Language Compiler Implementation
3  Copyright (c) 2016- ThePhD
4
5  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
6  software and associated documentation files (the "Software"
       ), to deal in the Software
7  without restriction, including without limitation the
       rights to use, copy, modify,
8  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
9  permit persons to whom the Software is furnished to do so,
       subject to the following
10 conditions:
11
12 The above copyright notice and this permission notice shall
       be included in all copies
13 or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
16 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
17 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
18 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
19 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
20 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
```

```
21
22 (* Parser for the LePiX language: compatible with both
       ocamlyacc and
23 menhir, as we have developed against both for testing
       purposes. *)
24
25 %}
26
27 %token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
28 %token LSQUARE RSQUARE COLON
29 %token DOT
30 %token PARALLEL INVOCATIONS ATOMIC THREADCOUNT
31 %token PLUSPLUS MINUSMINUS
32 %token PLUS MINUS TIMES DIVIDE ASSIGN
33 %token MODULO
34 %token PLUSASSIGN MINUSASSIGN TIMESASSIGN DIVIDEASSIGN
35 %token MODULOASSIGN
36 %token NOT AND OR EQ NEQ LT LEQ GT GEQ
37 %token TRUE FALSE
38 %token VAR LET
39 %token FUN TO BY
40 %token RETURN CONTINUE BREAK IF ELSE FOR WHILE
41 %token AMP CONST
42 %token <int> INT
43 %token <int> FLOAT
44 %token BOOL VOID STRING MEMORY
45 %token AUTO
46 %token NAMESPACE
47 %token IMPORT
48 %token <string> ID
49 %token <string> STRINGLITERAL
50 %token <Num.num> INTLITERAL
51 %token <Num.num> FLOATLITERAL
52 %token EOF
53
54 %right ASSIGN
55 %right PLUSASSIGN MINUSASSIGN
56 %right TIMESASSIGN DIVIDEASSIGN MODULOASSIGN
57 %left OR
58 %left AND
59 %left EQ NEQ
60 %left LT GT LEQ GEQ
61 %left PLUS MINUS
62 %left TIMES DIVIDE MODULO
63 %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:
```

```
107 | type_category builtin_type                { Ast.BuiltinType(
      $2, $1) }
108 | type_category builtin_type array_spec   { Ast.Array(Ast.
      BuiltinType($2, Ast.no_qualifiers), $3, $1) }
109 | type_category builtin_type sized_array_spec  { let (d, el
      ) = $3 in
110   if d <> ( List.length el ) then
111     raise(Parsing.Parse_error)
112   else
113     Ast.SizedArray(Ast.BuiltinType($2, Ast.no_qualifiers),
      d, el, $1)
114 }
115
116 sub_type_name_list_builder: { [] }
117 | sub_type_name { [$1] }
118 | sub_type_name_list_builder COMMA sub_type_name { $3 :: $1
      }
119
120 sub_type_name_list:
121 | sub_type_name_list_builder { List.rev $1 }
122
123 type_name:
124 | sub_type_name { $1 }
125 | type_category LPAREN sub_type_name_list RPAREN
      sub_type_name { Ast.Function($5, $3, $1) }
126
127 expression_comma_list:                      { [] }
128 | expression                                { [$1] }
129 | expression COMMA expression_comma_list { $1 :: $3 }
130
131 op_expression:
132 | expression TIMES expression { Ast.BinaryOp($1, Ast.Mult,
      $3) }
133 | expression DIVIDE expression { Ast.BinaryOp($1, Ast.Div,
      $3)  }
134 | expression PLUS expression { Ast.BinaryOp($1, Ast.Add, $3
      ) }
135 | expression MINUS expression { Ast.BinaryOp($1, Ast.Sub,
      $3) }
136 | expression MODULO expression { Ast.BinaryOp($1, Ast.
      Modulo, $3) }
137 | expression TIMESASSIGN expression { Ast.BinaryOp($1, Ast.
      MultAssign, $3) }
138 | expression DIVIDEASSIGN expression { Ast.BinaryOp($1, Ast
      .DivAssign, $3)  }
```

```
139 | expression PLUSASSIGN expression { Ast.BinaryOp($1, Ast.
      AddAssign, $3) }
140 | expression MINUSASSIGN expression { Ast.BinaryOp($1, Ast.
      SubAssign, $3) }
141 | expression MODULOASSIGN expression { Ast.BinaryOp($1, Ast
      .ModuloAssign, $3) }
142 | expression LT expression { Ast.BinaryOp($1, Ast.Less, $3)
      }
143 | expression GT expression { Ast.BinaryOp($1, Ast.Greater,
      $3) }
144 | expression LEQ expression { Ast.BinaryOp($1, Ast.Leq, $3)
      }
145 | expression GEQ expression { Ast.BinaryOp($1, Ast.Geq, $3)
      }
146 | expression NEQ expression { Ast.BinaryOp($1, Ast.Neq, $3)
      }
147 | expression EQ expression { Ast.BinaryOp($1, Ast.Equal, $3
      ) }
148 | expression AND expression { Ast.BinaryOp($1, Ast.And, $3)
      }
149 | expression OR expression { Ast.BinaryOp($1, Ast.Or, $3) }
150 | expression ASSIGN expression { Ast.Assignment($1, $3) }
151 | MINUS expression %prec NEG { Ast.PrefixUnaryOp(Ast.Neg,
      $2) }
152 | NOT expression { Ast.PrefixUnaryOp(Ast.Not, $2) }
153 | PLUSPLUS expression { Ast.PrefixUnaryOp(Ast.PreIncrement,
      $2) }
154 | MINUSMINUS expression { Ast.PrefixUnaryOp(Ast.
      PreDecrement, $2) }
155
156 value_expression:
157 | INTLITERAL { let v = match $1 with
158   | Num.Int(i) -> Ast.IntLit( (Int64.of_int i ), Base.
      default_integral_bit_width)
159   | Num.Big_int(bi) ->
160     begin try
161       Ast.IntLit( Int64.of_int( Big_int.int_of_big_int bi )
      , 32 )
162     with
163       _ -> Ast.IntLit( Big_int.int64_of_big_int bi, 64 )
164     end
165   | n -> Ast.FloatLit( Num.float_of_num n, Base.
      default_floating_bit_width )
166   in
167   Ast.Literal(v)
```

```
168 }
169 | FLOATLITERAL { Ast.Literal(Ast.FloatLit( Num.float_of_num
      ( $1 ), Base.default_floating_bit_width )) }
170 | STRINGLITERAL { Ast.Literal(Ast.StringLit($1)) }
171 | TRUE { Ast.Literal(Ast.BoolLit(true)) }
172 | FALSE { Ast.Literal(Ast.BoolLit(false)) }
173 | LSQUARE expression_comma_list RSQUARE { Ast.
      ArrayInitializer( $2 ) }
174 | LBRACE expression_comma_list RBRACE { Ast.
      ObjectInitializer( $2 ) }
175
176 postfix_expression:
177 | expression LSQUARE expression_comma_list RSQUARE { Ast.
      Index($1, $3) }
178 | expression LPAREN expression_comma_list RPAREN { Ast.Call
      ($1, $3) }
179
180 expression:
181 | qualified_id { Ast.QualifiedId($1) }
182 | value_expression { $1 }
183 | value_expression DOT qualified_id { Ast.Member($1, $3) }
184 | op_expression { $1 }
185 | postfix_expression { $1 }
186 | postfix_expression DOT qualified_id { Ast.Member($1, $3)
      }
187 | LPAREN expression RPAREN { $2 }
188 | LPAREN expression RPAREN DOT qualified_id { Ast.Member($2
      , $5) }
189
190 type_spec:
191 | COLON type_name { $2 }
192
193 maybe_type_spec: { Ast.BuiltinType(Ast.Auto, Ast.
      no_qualifiers) }
194 | COLON type_name { $2 }
195
196 binding:
197 | ID type_spec { ($1, $2) }
198
199 binding_list: { [] }
200 | binding { [$1] }
201 | binding COMMA binding_list { $1 :: $3 }
202
203 var_binding:
204 | ID type_spec { ($1, $2) }
```

```
205 | ID { ($1, Ast.BuiltinType(Ast.Auto, Ast.no_qualifiers)) }
206
207 variable_definition:
208 | VAR var_binding ASSIGN expression { Ast.VarBinding($2, $4
      ) }
209 | LET var_binding ASSIGN expression { Ast.VarBinding(Ast.
      add_const($2), $4) }
210 | VAR var_binding { Ast.VarBinding($2, Ast.Noop) }
211 | LET var_binding { Ast.VarBinding(Ast.add_const($2), Ast.
      Noop) }
212
213 statement_list_builder: { [] }
214 | statement_list_builder statement { $2 :: $1 }
215
216 statement_list :
217 | statement_list_builder { List.rev $1 }
218
219 parallel_binding:
220 | INVOCATIONS ASSIGN expression { Ast.Invocations($3) }
221 | THREADCOUNT ASSIGN expression { Ast.ThreadCount($3) }
222
223 parallel_binding_list_builder: { [] }
224 | parallel_binding { [$1] }
225 | parallel_binding_list_builder COMMA parallel_binding { $3
      :: $1 }
226
227 parallel_binding_list:
228 | parallel_binding_list_builder { List.rev $1 }
229
230 sub_general_statement:
231 | expression { Ast.ExpressionStatement($1) }
232 | variable_definition { Ast.VariableStatement($1) }
233
234 general_statement:
235 | sub_general_statement SEMI { Ast.General($1) }
236
237 control_initializer_builder:
238 | sub_general_statement { ( [$1], 1 ) }
239 | control_initializer_builder SEMI sub_general_statement {
      let (l, c) = $1 in ( $3 :: l, 1 + c ) }
240
241 control_initializer:
242 | control_initializer_builder { let ( il, c ) = $1 in if c
      < 2 then ([], List.hd il) else (List.rev ( List.tl il ),
       List.hd il) }
```

```
243
244 sub_general_statement_list_builder: { [] }
245 | sub_general_statement { [$1] }
246 | sub_general_statement_list_builder COMMA
      sub_general_statement { $3 :: $1 }
247
248 sub_general_statement_list:
249 | sub_general_statement_list_builder { List.rev $1 }
250
251 statement:
252 | general_statement { $1 }
253 | IF LPAREN control_initializer RPAREN LBRACE
      statement_list RBRACE { Ast.IfBlock($3,$6) }
254 | IF LPAREN control_initializer RPAREN LBRACE
      statement_list RBRACE ELSE LBRACE statement_list RBRACE
      { Ast.IfElseBlock($3,$6,$10)  }
255 | WHILE LPAREN control_initializer RPAREN LBRACE
      statement_list RBRACE { Ast.WhileBlock($3, $6) }
256 | FOR LPAREN sub_general_statement_list SEMI expression
      SEMI expression_comma_list RPAREN LBRACE statement_list
      RBRACE { Ast.ForBlock($3, $5, $7, $10) }
257 | FOR LPAREN expression TO expression BY expression RPAREN
      LBRACE statement_list RBRACE { Ast.ForByToBlock($3, $5,
      $7, $10) }
258 | RETURN expression SEMI { Ast.Return($2) }
259 | RETURN SEMI { Ast.Return(Ast.Noop) }
260 | BREAK SEMI { Ast.Break(1) }
261 | BREAK INTLITERAL SEMI { Ast.Break( Num.int_of_num $2 ) }
262 | CONTINUE SEMI { Ast.Continue }
263 | PARALLEL LPAREN parallel_binding_list RPAREN LBRACE
      statement_list RBRACE  { Ast.ParallelBlock($3, $6) }
264 | 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) }
265 | ATOMIC LBRACE statement_list RBRACE { Ast.AtomicBlock($3)
      }
266
267 function_definition:
268 | FUN ID LPAREN binding_list RPAREN maybe_type_spec LBRACE
      statement_list RBRACE { ([$2], $4, $6, $8) }
269
270 import_definition:
271 | IMPORT qualified_id { $2 }
```

```
272
273 definition_list : { [] }
274 | definition_list import_definition { Ast.Import(Ast.
      LibraryImport($2)) :: $1 }
275 | definition_list function_definition { Ast.Basic(Ast.
      FunctionDefinition($2)) :: $1 }
276 | definition_list variable_definition SEMI { Ast.Basic(Ast.
      VariableDefinition($2)) :: $1 }
277 | definition_list NAMESPACE qualified_id LBRACE
      definition_list RBRACE { Ast.Namespace($3, List.rev $5)
      :: $1 }
278
279 program:
280 | definition_list EOF { Ast.Program(List.rev $1) }
```

../source/parser.mly

```
1  (* LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
```

```
20
21 (* Drives the typical lexing and parsing algorithm
22 while adding pertinent source, line and character
      information. *)
23
24 type context = {
25       mutable source_name : string;
26       mutable source_code : string;
27   mutable original_source_code : string;
28   mutable token_count : int;
29       mutable token : Parser.token * Base.token_source;
30 }
31
32 let lex sourcename lexbuf =
33   let rec acc lexbuf tokennumber tokens =
34     let next_token = Scanner.token lexbuf
35     and startp = Lexing.lexeme_start_p lexbuf
36     and endp = Lexing.lexeme_end_p lexbuf
37     in
38     let line = startp.Lexing.pos_lnum
39     and relpos = (1 + startp.Lexing.pos_cnum - startp.
      Lexing.pos_bol)
40     and endrelpos = (1 + endp.Lexing.pos_cnum - endp.Lexing
      .pos_bol)
41     and abspos = startp.Lexing.pos_cnum
42     and endabspos = endp.Lexing.pos_cnum
43     in
44     let create_token token =
45       let t = ( token, { Base.token_source_name =
      sourcename; Base.token_number = tokennumber;
46           Base.token_line_number = line; Base.
      token_line_start = startp.Lexing.pos_bol;
47           Base.token_column_range = (relpos, endrelpos); Base
      .token_character_range = (abspos, endabspos) }
48       ) in
49       t
50     in
51     match next_token with
52     | Parser.EOF as token -> ( create_token token ) ::
      tokens
53     | token -> ( create_token token ) :: ( acc lexbuf ( 1 +
       tokennumber ) tokens )
54   in
55   acc lexbuf 0 []
56
```

```
57 let parse context token_list =
58   (* Keep a reference to the original token list
59   And use that to dereference rather than whatever crap we
       get from
60   the channel *)
61   let tokenlist = ref(token_list) in
62   let tokenizer _ = match !tokenlist with
63   (* Break each token down into pieces, info and all*)
64   | (token, info) :: rest ->
65     context.source_name <- info.Base.token_source_name;
66     context.token_count <- 1 + context.token_count;
67     context.token <- ( token, info );
68     (* Shift the list down by one by referencing
69     the beginning of the rest of the list *)
70     tokenlist := rest;
71     (* return token we care about *)
72     token
73   (* The parser stops calling the tokenizer when
74   it hits EOF: if it reaches the empty list, WE SCREWED UP
       *)
75   | [] -> raise (Errors.MissingEoF)
76   in
77   (* Pass in an empty channel built off a cheap string
78   and then ignore the fuck out of it in our 'tokenizer'
79   internal function *)
80   let program = Parser.program tokenizer (Lexing.
       from_string "") in
81   program
82
83 let analyze program =
84   (* TODO: other important checks and semantic analysis
       here
85   that will create a proper checked program type*)
86   let sem = Semant.check program in
87   sem
```

../source/driver.ml

```
1 (* LePiX Language Compiler Implementation
2 Copyright (c) 2016- ThePhD
3
4 Permission is hereby granted, free of charge, to any person
      obtaining a copy of this
5 software and associated documentation files (the "Software"
    ), to deal in the Software
6 without restriction, including without limitation the
    rights to use, copy, modify,
```

```
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
       CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Semantic checking for the Lepix compiler that will
       produce a new
22 SemanticProgram type with things like locals group into a
       single type
23 and type promotions / conversions organized for operators.
       *)
24
25 module StringMap = Map.Make(String)
26
27 let extract_binding = function
28   | Ast.VarBinding(b, _) -> b
29
30 let extract_s_binding = function
31   | Semast.SVarBinding(b, _) -> b
32
33 let extract_s_binding_name = function
34   | (n, _) -> n
35
36 let extract_s_binding_type = function
37   | (_, tn) -> tn
38
39 let create_s_attributes () = {
40   Semast.attr_parallelism = false;
```

```
41    Semast.attr_arrays = 0;
42    Semast.attr_strings = false;
43  }
44
45  let create_s_environment () = {
46    Semast.env_usings = [];
47    Semast.env_symbols = StringMap.empty;
48    Semast.env_definitions = StringMap.empty;
49    Semast.env_imports = [];
50    Semast.env_loops = [];
51  }
52
53  let enter_block envl locals parameters =
54    let acc_symbols m l =
55      let ( n, tn ) = (extract_s_binding_name l,
       extract_s_binding_type l) in
56      StringMap.add n tn m
57    in
58    let symbols = List.fold_left acc_symbols StringMap.empty
       parameters in
59    let symbols = List.fold_left acc_symbols symbols locals
       in
60    let env = {
61      Semast.env_usings = [];
62      Semast.env_symbols = symbols;
63      Semast.env_definitions = StringMap.empty;
64      Semast.env_imports = [];
65      Semast.env_loops = [];
66    } in
67    env :: envl
68
69  let enter_parameter_block envl parameters =
70    let acc_symbols m l =
71      let ( n, tn ) = (extract_s_binding_name l,
       extract_s_binding_type l) in
72      StringMap.add n tn m
73    in
74    let m k v1 v2 = match (v1, v2) with
75      | Some(x), Some(_) -> Some(x)
76      | Some(x) as s, None -> s
77      | None, ( Some(y) as s ) -> s
78      | _ -> None
79    in
80    let symbols = List.fold_left acc_symbols StringMap.empty
       parameters in
```

```
81    let env = List.hd envl in
82    let env = { env with
83      Semast.env_symbols = ( StringMap.merge m symbols env.
       Semast.env_symbols );
84    } in
85    env :: envl
86
87  let lookup_id name mapl =
88    let rec find = function
89      | [] -> None
90      | h :: tl -> try Some ( StringMap.find name h )
91        with | _ -> find tl
92    in
93    find mapl
94
95  let env_lookup_id name envl =
96    let mapl = ( List.map ( fun env -> env.Semast.env_symbols
       ) envl ) in
97    lookup_id name mapl
98
99  let accumulate_string_type_bindings syms (n, qt) =
100   let rec list_cmp v1 v2 = match v1, v2 with
101     | hl::tll, hr::tlr -> hl = hr && list_cmp tll tlr
102     | [], [] -> true
103     | _, [] -> false
104     | [], _ -> false
105   in
106   try
107     let check_f l =
108       let qt_argst = Semast.args_type_name qt in
109       let pred t =
110         let argst = Semast.args_type_name t in
111         ( List.length argst ) = ( List.length qt_argst )
112         && ( list_cmp argst qt_argst )
113       in
114       begin try
115         let _ = List.find pred l in
116         raise(Errors.FunctionAlreadyExists("an id with name
       " ^ n ^ " and type " ^ ( Representation.
      string_of_s_type_name qt ) ^ " is already present"))
117       with
118         Not_found -> Semast.SOverloads( qt :: l )
119       end
120     in
121     let v = StringMap.find n syms in
```

```
122       let vt = match v with
123         | Semast.SOverloads(tl) -> check_f tl
124         | Semast.SFunction(_,_,_) as t -> check_f [t]
125         | _ -> raise(Not_found)
126       in
127       StringMap.add n vt syms
128     with Not_found ->
129       StringMap.add n qt syms
130
131 let import_builtin_module symbols = function
132   | Semast.Lib -> begin
133       let c_bindings = [
134         ("lib.print", Semast.SFunction( Semast.void_t, [
      Semast.int32_t], Semast.no_qualifiers ) );
135         ("lib.print", Semast.SFunction( Semast.void_t, [
      Semast.string_t], Semast.no_qualifiers ) );
136         ("lib.print", Semast.SFunction( Semast.void_t, [
      Semast.float64_t], Semast.no_qualifiers ) );
137         ("lib.print_n", Semast.SFunction( Semast.void_t, [
      Semast.int32_t], Semast.no_qualifiers ) );
138         ("lib.print_n", Semast.SFunction( Semast.void_t, [
      Semast.string_t], Semast.no_qualifiers ) );
139         ("lib.print_n", Semast.SFunction( Semast.void_t, [
      Semast.float64_t], Semast.no_qualifiers ) );
140       ]
141       in
142       print_endline "Here!";
143       let symbols = List.fold_left
      accumulate_string_type_bindings symbols c_bindings in
144       (symbols)
145     end
146
147 let rec type_name_of_ast_type_name = function
148     | Ast.BuiltinType(bt, tq) -> Semast.SBuiltinType( bt, tq
      )
149     | Ast.Array(tn, d, tq) -> Semast.SArray( (
      type_name_of_ast_type_name tn ), d, tq )
150     | Ast.SizedArray(tn, d, dims, tq) -> Semast.SSizedArray(
      ( type_name_of_ast_type_name tn ), d, dims, tq )
151     | Ast.Function( tn, pl, tq) -> Semast.SFunction( (
      type_name_of_ast_type_name tn ), ( List.map
      type_name_of_ast_type_name pl ), tq )
152
153 let type_name_of_ast_literal attrs envl astlit =
154   let t = match astlit with
```

```
155       | Ast.BoolLit(_) -> Ast.BuiltinType( Ast.Bool, Ast.
      no_qualifiers )
156       | Ast.IntLit(_, b) -> Ast.BuiltinType( Ast.Int(b), Ast.
      no_qualifiers )
157       | Ast.FloatLit(_, b) -> Ast.BuiltinType( Ast.Float(b),
      Ast.no_qualifiers )
158       | Ast.StringLit(_) -> Ast.BuiltinType( Ast.String, Ast.
      no_qualifiers )
159     in
160     type_name_of_ast_type_name t
161
162   let check_binary_op_common_type lt bop rt = match (lt, rt)
      with
163     | (Semast.SBuiltinType(Ast.Int(n), ltq), Semast.
      SBuiltinType(Ast.Int(m), rtq)) -> Semast.SBuiltinType(
      Ast.Int(max n m), Semast.no_qualifiers)
164     | (Semast.SBuiltinType(Ast.Float(n), ltq), Semast.
      SBuiltinType(Ast.Int(m), rtq)) -> Semast.SBuiltinType(
      Ast.Float(max n m), Semast.no_qualifiers)
165     | (Semast.SBuiltinType(Ast.Int(n), ltq), Semast.
      SBuiltinType(Ast.Float(m), rtq)) -> Semast.SBuiltinType(
      Ast.Float(max n m), Semast.no_qualifiers)
166     | (Semast.SBuiltinType(Ast.Float(n), ltq), Semast.
      SBuiltinType(Ast.Float(m), rtq)) -> Semast.SBuiltinType(
      Ast.Float(max n m), Semast.no_qualifiers)
167     | (Semast.SBuiltinType(Ast.Bool, ltq), Semast.
      SBuiltinType(Ast.Bool, rtq)) -> Semast.SBuiltinType(Ast.
      Bool, Semast.no_qualifiers)
168     | _ -> raise(Errors.InvalidBinaryOperation("cannot
      perform a binary operation on two non-numeric types"))
169
170   let check_unary_op uop sr = match sr with
171     | Semast.SBuiltinType(Ast.Int(n), ltq) -> Semast.
      SBuiltinType(Ast.Int(n), Semast.no_qualifiers)
172     | Semast.SBuiltinType(Ast.Float(n), ltq) -> Semast.
      SBuiltinType(Ast.Float(n), Semast.no_qualifiers)
173     | _ -> raise(Errors.InvalidUnaryOperation("cannot perform
       a unary operation on this type"))
174
175   let overload_resolution args = function
176     | Semast.SFunction(__, tnl, _) -> let argslen = ( List.
      length args ) in
177       let overloadlen = ( List.length tnl ) in
178       argslen = overloadlen &&
179       if ( argslen > 0 ) then ( List.exists ( fun a -> List.
```

```
      mem a tnl ) args )
180       else true
181     | _ -> raise(Errors.TypeMismatch("cannot resolve an
        overload that includes a non-function in its type
        listing"))
182
183   let check_function_overloads args overloadlist =
184     begin try
185       let ft = List.find ( overload_resolution args )
        overloadlist
186       in
187       ( ft, Semast.return_type_name ft )
188     with _ ->
189       let argslist = "( " ^ ( String.concat "," ( List.map
        Representation.string_of_s_type_name args ) ) ^ " )" in
190       raise(Errors.BadFunctionCall("could not resolve the
        specific overload for this set of " ^ ( Representation.
        string_of_s_type_name ( Semast.SOverloads(overloadlist)
        ) ) ^ " using " ^ argslist ))
191     end
192
193   let rec type_name_of_ast_expression attrs envl astexpr =
194     let t = match astexpr with
195       | Ast.Literal(lit) -> type_name_of_ast_literal attrs
        envl lit
196       | Ast.QualifiedId(qid)-> let qualname = Semast.
        string_of_qualified_id qid in
197         begin match ( env_lookup_id qualname envl ) with
198           | None -> raise(Errors.IdentifierNotFound("
        identifier '" ^ qualname ^ "' not found"))
199           | Some(stn) -> stn
200         end
201       | Ast.Call(e, args) -> let ft =
        type_name_of_ast_expression attrs envl e in
202         (* TODO: check arguments to make sure it matches *)
203         begin match ft with
204           | Semast.SFunction(rt, pl, tq) ->
205             rt
206           | Semast.SOverloads(fl) ->
207             let sargs = List.map (
        type_name_of_ast_expression attrs envl ) args in
208             let (ft, r) = check_function_overloads sargs fl
        in
209             r
210           | _ -> raise(Errors.TypeMismatch("expected a
```

```
      function type, but received something else."))
211       end
212     | Ast.Noop -> Semast.void_t
213     | Ast.BinaryOp(l, bop, r) ->
214       let sl = type_name_of_ast_expression attrs envl l
215       and sr = type_name_of_ast_expression attrs envl r
216       in
217       check_binary_op_common_type sl bop sr
218     | Ast.PrefixUnaryOp(uop, r) ->
219       let sr = type_name_of_ast_expression attrs envl r in
220       check_unary_op uop sr
221     | Ast.Assignment(lhs, rhs) -> let lhst =
      type_name_of_ast_expression attrs envl lhs in
222       lhst
223     | Ast.Member(_, _) -> raise(Errors.Unsupported("member
      access is not supported"))
224     | _ -> raise(Errors.Unsupported("expression conversion
      currently unsupported"))
225   in
226   (* TODO: some type checks to make sure weird things like
      void& aren't put in place... *)
227   t
228
229 let process_ast_import prefix symbols defs imports =
      function
230   | Ast.LibraryImport(qid) ->
231     let qualname = Semast.string_of_qualified_id qid in
232     let (v, impsymbols) = match List.filter ( fun (n, _) ->
      n = qualname ) Semast.builtin_library_names with
233       | (_, bltin) :: [] -> let b = Semast.SBuiltin(bltin)
      in
234         let ( bltinsymbols ) = import_builtin_module
      symbols bltin in
235         ( b, bltinsymbols )
236       | _ -> ( Semast.SDynamic(qualname), symbols )
237     in
238     ( prefix, impsymbols, defs, v :: imports )
239
240
241 let generate_global_env = function
242 | Ast.Program(ast_definitions) ->
243   let rec acc ( prefix, symbols, defs, imports ) def =
244     match def with
245       | Ast.Import(imp) -> process_ast_import prefix
      symbols defs imports imp
```

```
246         | Ast.Basic(Ast.FunctionDefinition((qid, args, rt, _)
    )) ->
247           let argst = List.map Ast.binding_type args in
248           let qualname = prefix ^ Semast.
    string_of_qualified_id qid in
249           let qt = Ast.Function(rt, argst, Ast.no_qualifiers)
     in
250           let sqt = ( type_name_of_ast_type_name qt ) in
251           print_endline "here";
252           let nsymbols = accumulate_string_type_bindings
    symbols (qualname, sqt)
253           and ndefs = accumulate_string_type_bindings defs (
    qualname, sqt)
254           in
255           ( prefix, nsymbols, ndefs, imports )
256         | Ast.Basic(Ast.VariableDefinition(v)) ->
257           let (name, qt) = extract_binding v in
258           let qualname = prefix ^ name in
259           if StringMap.mem prefix symbols then raise (Errors.
    VariableAlreadyExists(qualname)) else
260           let nsymbols = ( StringMap.add qualname (
    type_name_of_ast_type_name qt ) symbols )
261           and ndefs = ( StringMap.add qualname (
    type_name_of_ast_type_name qt ) defs )
262           in
263           ( prefix, nsymbols, ndefs, imports )
264         | Ast.Namespace(n, dl) ->
265           let qualname = prefix ^ Semast.
    string_of_qualified_id n in
266           let (_, innersymbols, innerdefs, innerimports ) =
    List.fold_left acc ( qualname ^ ".", symbols, defs,
    imports ) dl in
267           ( prefix, innersymbols, innerdefs, innerimports )
268       in
269       let (_, symbols, defs, imports) = List.fold_left acc ("",
          StringMap.empty, StringMap.empty, []) ast_definitions
         in
270       let attrs = create_s_attributes () in
271       let env = {
272         Semast.env_usings = [];
273         Semast.env_symbols = symbols;
274         Semast.env_definitions = defs;
275         Semast.env_imports = imports;
276         Semast.env_loops = [];
277       }
```

```
278   in
279   ( attrs, env )
280
281 let check_qualified_identifier attrs envl sl t =
282   (attrs, envl, Semast.SQualifiedId(sl, t))
283
284 let check_function_call attrs envl target args =
285   let (t, rt) = match Semast.type_name_of_s_expression
       target with
286     | Semast.SFunction(tn, tnl, tq) as f -> f, tn
287     | Semast.SOverloads(fl) ->
288       let args_t = ( List.map Semast.
       type_name_of_s_expression args ) in
289       check_function_overloads args_t fl
290     | _ -> raise(Errors.BadFunctionCall("cannot invoke an
       expression which does not result in a function type of
       some sort"))
291   in
292   (attrs, envl, Semast.SCall(( Semast.
       coerce_type_name_of_s_expression t target ), args, rt ))
293
294 let generate_s_binding prefix attrs envl = function
295   | (name, tn) -> (attrs, envl, (Semast.
       string_of_qualified_id ( prefix @ [name] ),
       type_name_of_ast_type_name tn))
296
297 let gather_ast_locals attrs envl sl pl =
298   let acc locals = function
299     | Ast.General(Ast.VariableStatement(v)) ->
300       let (_, _, sb) = generate_s_binding [] attrs envl (
       extract_binding v ) in
301       sb :: locals
302     | _ -> locals
303   in
304   let l = List.rev( List.fold_left acc [] sl ) in
305   if ( List.length l ) > 0 then begin
306     let envl = ( enter_block envl l pl ) in
307     (true, attrs, envl, l)
308   end else
309     let envl = ( enter_parameter_block envl pl ) in
310     (false, attrs, envl, l)
311
312 let generate_s_literal attrs envl = function
313   | Ast.BoolLit(b) -> (attrs, envl, Semast.SBoolLit(b))
314   | Ast.IntLit(i, b) -> (attrs, envl, Semast.SIntLit(i, b))
```

```
315     | Ast.FloatLit(f, b) -> (attrs, envl, Semast.SFloatLit(f,
          b))
316     | Ast.StringLit(s) -> (attrs, envl, Semast.SStringLit(s))
317
318   let rec generate_s_expression attrs envl astexpr =
319     let acc_s_expression (attrs, envl, sel) e =
320       let (attrs, envl, se) = generate_s_expression attrs
        envl e in
321       (attrs, envl, se :: sel)
322     in
323     let ( attrs, envl, se ) = match astexpr with
324       | Ast.Literal(lit) ->
325         let ( attrs, envl, slit ) = generate_s_literal attrs
        envl lit in
326         ( attrs, envl, Semast.SLiteral( slit ) )
327       | Ast.QualifiedId(sl) ->
328         let t = type_name_of_ast_expression attrs envl
        astexpr in
329         check_qualified_identifier attrs envl sl t
330       | Ast.Call(e, el) ->
331         let ( attrs, envl, target ) = ( generate_s_expression
         attrs envl e ) in
332         let ( attrs, envl, args ) = ( List.fold_left
        acc_s_expression (attrs, envl, []) el ) in
333         let args = List.rev args in
334         check_function_call attrs envl target args
335       | Ast.BinaryOp(lhs, bop, rhs) ->
336         let ( attrs, envl, slhs ) = ( generate_s_expression
        attrs envl lhs ) in
337         let ( attrs, envl, srhs ) = ( generate_s_expression
        attrs envl rhs ) in
338         let rhst = ( Semast.type_name_of_s_expression srhs )
        in
339         let lhst = ( Semast.type_name_of_s_expression slhs )
        in
340         ( attrs, envl, Semast.SBinaryOp( slhs, bop, srhs, (
        check_binary_op_common_type lhst bop rhst ) ) )
341       | Ast.PrefixUnaryOp(uop, rhs) ->
342         let ( attrs, envl, srhs ) = ( generate_s_expression
        attrs envl rhs ) in
343         let rhst = ( Semast.type_name_of_s_expression srhs )
        in
344         ( attrs, envl, Semast.SPrefixUnaryOp( uop, srhs, (
        check_unary_op uop rhst ) ) )
345       | Ast.Assignment(lhs, rhs) ->
```

```
346       let ( attrs, envl, slhs ) = ( generate_s_expression
      attrs envl lhs ) in
347       let ( attrs, envl, srhs ) = ( generate_s_expression
      attrs envl rhs ) in
348       let lhst = ( Semast.type_name_of_s_expression slhs )
      in
349       ( attrs, envl, Semast.SAssignment( slhs, srhs, lhst )
       )
350     | Ast.Noop -> ( attrs, envl, Semast.SNoop )
351     | _ -> raise(Errors.Unsupported("expression generation
      for this type is current unsupported"))
352   in
353   let t = Semast.type_name_of_s_expression se in
354   let attrs = match t with
355     | Semast.SArray(_,d,_)
356     | Semast.SSizedArray(_,d,_,_) -> {
357         Semast.attr_strings = attrs.Semast.attr_strings;
358         Semast.attr_arrays = max d attrs.Semast.attr_arrays
      ;
359         Semast.attr_parallelism = attrs.Semast.
      attr_parallelism;
360       }
361     | Semast.SBuiltinType( Ast.String, _ ) -> {
362         Semast.attr_strings = true;
363         Semast.attr_arrays = attrs.Semast.attr_arrays;
364         Semast.attr_parallelism = attrs.Semast.
      attr_parallelism;
365       }
366     | _ -> attrs
367   in
368   ( attrs, envl, se )
369
370 let generate_s_variable_definition prefix attrs envl =
      function
371   | Ast.VarBinding(b, e) ->
372     let (attrs, envl, sb) = generate_s_binding prefix attrs
       envl b in
373     let (attrs, envl, se) = generate_s_expression attrs
      envl e in
374     (attrs, envl, Semast.SVarBinding(sb, se))
375
376 let generate_s_general_statement attrs envl = function
377   | Ast.ExpressionStatement(e) ->
378     let (attrs, envl, se ) = generate_s_expression attrs
      envl e in
```

```
379       ( attrs, envl, Semast.SExpressionStatement( se ) )
380     | Ast.VariableStatement(v) ->
381       let (attrs, envl, sv ) = generate_s_variable_definition
          [] attrs envl v in
382       ( attrs, envl, Semast.SVariableStatement( sv ) )
383
384   let generate_s_statement attrs envl = function
385     | Ast.General(g) ->
386       let ( attrs, envl, sgs ) = generate_s_general_statement
          attrs envl g in
387       ( attrs, envl, Semast.SGeneral( sgs ) )
388     | Ast.Return(e) ->
389       let ( attrs, envl, se ) = generate_s_expression attrs
         envl e in
390       ( attrs, envl, Semast.SReturn( se ) )
391     | _ -> raise(Errors.Unsupported("statement type not
         supported"))
392
393   let check_returns name ssl rt =
394     (* Todo: recursively inspect all inner blocks for return
         types as well *)
395     let acc rl = function
396       | Semast.SReturn(se) -> ( Semast.
         type_name_of_s_expression se ) :: rl
397       | _ -> rl
398     in
399     let returns = List.fold_left acc [] ssl in
400     let returnlength = List.length returns in
401     if name = "main" then begin
402       let sret0 = Semast.SReturn(Semast.SLiteral(Semast.
         SIntLit(Int64.zero, 32))) in
403       let mainpred = function
404         | Semast.SBuiltinType(Ast.Auto, _) ->
405           ()
406         | Semast.SBuiltinType(Ast.Int(32), (_, r)) ->
407           if r then raise(Errors.InvalidMainSignature("Cannot
          return a reference from main"));
408           ()
409         | _ ->
410           raise(Errors.InvalidMainSignature("You can only
         return an int from main"))
411       in
412       let ssl = if returnlength < 1 then begin ssl @ [sret0]
         end else ssl
413       in
```

```
414       let ssl = match rt with
415         | Semast.SBuiltinType(Ast.Int(32), (_, r)) ->
416           if r then raise(Errors.InvalidMainSignature("Cannot
        return a reference to and integer"));
417           let _ = List.iter mainpred returns in
418           ssl
419         | _ -> let _ = List.iter mainpred returns in
420           ssl
421       in
422       (ssl, Semast.int32_t)
423     end
424     else begin
425       let generalpred rt r = match rt with
426         | Semast.SBuiltinType(Ast.Auto, _) -> r
427         | _ -> let r = Semast.unqualify r in
428           let urt = Semast.unqualify rt in
429           if r <> urt then raise(Errors.
        InvalidFunctionSignature("return types do not match
        across all returns", name))
430           else rt
431       in
432       let rt = List.fold_left generalpred rt returns in
433       let (ssl, rt) = match rt with
434         | Semast.SBuiltinType(Ast.Auto, _) ->
435           if returnlength < 1 then
436             ( ssl @ [Semast.SReturn(Semast.SNoop)], Semast.
        void_t )
437           else
438             (ssl, rt)
439         | Semast.SBuiltinType(Ast.Void, _) ->
440           if returnlength < 1 then
441             ( ssl @ [Semast.SReturn(Semast.SNoop)], Semast.
        void_t )
442           else
443             (ssl, rt)
444         | _ ->
445           if returnlength < 1 then
446             raise(Errors.InvalidFunctionSignature("function
        was expected to return a value: returned no value", name
        ))
447           else
448             (ssl, rt)
449       in
450       (ssl, rt)
451     end
```

```
452
453 let generate_s_function_definition prefix attrs envl
      astfdef =
454   let acc_ast_statements (attrs, envl, ssl) s =
455     let ( attrs, envl, ss ) = ( generate_s_statement attrs
      envl s ) in
456     ( attrs, envl, ss :: ssl )
457   in
458   let acc_ast_parameters (attrs, envl, pl) p =
459     let (attrs, envl, sp) = generate_s_binding [] attrs
      envl p in
460     (attrs, envl, sp :: pl)
461   in
462   let (qid, astparameters, astrt, body) = astfdef in
463   let fqid = prefix @ qid in
464   let fqn = Semast.string_of_qualified_id fqid in
465   let (attrs, envl, parameters) = List.fold_left
      acc_ast_parameters (attrs, envl, []) astparameters in
466   let rt = type_name_of_ast_type_name astrt in
467   let (has_locals, attrs, envl, bl) = gather_ast_locals
      attrs envl body parameters in
468   let (attrs, envl, ssl) = List.fold_left
      acc_ast_statements (attrs, envl, []) body in
469   let ssl = List.rev ssl in
470   let (ssl, rt) = check_returns fqn ssl rt in
471   let sfuncdef = if has_locals then
472   {
473     Semast.func_name = fqid;
474     Semast.func_parameters = Semast.SParameters(parameters)
      ;
475     Semast.func_return_type = rt;
476     Semast.func_body = [Semast.SBlock(Semast.SLocals(bl),
      ssl)];
477   }
478   else
479   {
480     Semast.func_name = fqid;
481     Semast.func_parameters = Semast.SParameters(parameters)
      ;
482     Semast.func_return_type = rt;
483     Semast.func_body = ssl;
484   }
485   in
486   ( attrs, envl, Semast.SFunctionDefinition( sfuncdef ) )
487
```

```
488
489 let generate_s_basic_definition prefix attrs envl =
      function
490   | Ast.FunctionDefinition(fdef) ->
491     let (attrs, envl, sfdef) =
      generate_s_function_definition prefix attrs envl fdef in
492     (attrs, envl, Semast.SBasic(sfdef))
493   | Ast.VariableDefinition(vdef) ->
494     let (attrs, envl, svdef) =
      generate_s_variable_definition prefix attrs envl vdef in
495     (attrs, envl, Semast.SBasic(Semast.SVariableDefinition(
      svdef)))
496
497
498 let define_libraries attrs env =
499   let fi = Semast.SFunction(Semast.void_t, [Semast.int32_t
      ], Semast.no_qualifiers) in
500   let ff = Semast.SFunction(Semast.void_t, [Semast.
      float64_t], Semast.no_qualifiers) in
501   let fs = Semast.SFunction(Semast.void_t, [Semast.string_t
      ], Semast.no_qualifiers) in
502   let fo = Semast.SOverloads([fi;ff;fs]) in
503   let lib_printn_defint = {
504     Semast.func_name = ["lib"; "print_n"];
505     Semast.func_parameters = Semast.SParameters([("i",
      Semast.int32_t)]);
506     Semast.func_return_type = Semast.void_t;
507     Semast.func_body = [
508       Semast.SGeneral(Semast.SExpressionStatement(
509         Semast.SCall(Semast.SQualifiedId(["lib"; "print"],
      fo), [Semast.SQualifiedId(["i"], Semast.int32_t)],
      Semast.void_t)
510       ));
511       Semast.SGeneral(Semast.SExpressionStatement(
512         Semast.SCall(Semast.SQualifiedId(["lib"; "print"],
      fo), [Semast.SLiteral(Semast.SStringLit("\n"))], Semast.
      void_t)
513       ));
514       Semast.SReturn(Semast.SNoop);
515     ];
516   }
517   and lib_printn_deffloat = {
518     Semast.func_name = ["lib"; "print_n"];
519     Semast.func_parameters = Semast.SParameters([("i",
      Semast.float64_t)]);
```

```
520       Semast.func_return_type = Semast.void_t;
521       Semast.func_body = [
522         Semast.SGeneral(Semast.SExpressionStatement(
523           Semast.SCall(Semast.SQualifiedId(["lib"; "print"],
        fo), [Semast.SQualifiedId(["i"], Semast.float64_t)],
        Semast.void_t)
524         ));
525         Semast.SGeneral(Semast.SExpressionStatement(
526           Semast.SCall(Semast.SQualifiedId(["lib"; "print"],
        fo), [Semast.SLiteral(Semast.SStringLit("\n"))], Semast.
        void_t)
527         ));
528         Semast.SReturn(Semast.SNoop);
529       ];
530     }
531     and lib_printn_defstr = {
532       Semast.func_name = ["lib"; "print_n"];
533       Semast.func_parameters = Semast.SParameters([("i",
        Semast.string_t)]);
534       Semast.func_return_type = Semast.void_t;
535       Semast.func_body = [
536         Semast.SGeneral(Semast.SExpressionStatement(
537           Semast.SCall(Semast.SQualifiedId(["lib"; "print"],
        fo), [Semast.SQualifiedId(["i"], Semast.string_t)],
        Semast.void_t)
538         ));
539         Semast.SGeneral(Semast.SExpressionStatement(
540           Semast.SCall(Semast.SQualifiedId(["lib"; "print"],
        fo), [Semast.SLiteral(Semast.SStringLit("\n"))], Semast.
        void_t)
541         ));
542         Semast.SReturn(Semast.SNoop);
543       ];
544     }
545     in
546     let libdefs = [
547       Semast.SBasic(Semast.SFunctionDefinition(
        lib_printn_defstr));
548       Semast.SBasic(Semast.SFunctionDefinition(
        lib_printn_defint));
549       Semast.SBasic(Semast.SFunctionDefinition(
        lib_printn_deffloat));
550     ] in
551     let acc defs = function
552       | Semast.SBasic(Semast.SFunctionDefinition(fdef)) ->
```

```
553         let n = (Semast.string_of_qualified_id fdef.Semast.
    func_name )
554         and qt = ( Semast.type_name_of_s_function_definition
    fdef )
555         in
556         if ( StringMap.mem n defs ) then defs else
    accumulate_string_type_bindings defs (n, qt)
557       | Semast.SBasic(Semast.SVariableDefinition(Semast.
    SVarBinding((n, tn), _))) ->
558         ( StringMap.add n tn defs )
559     in
560     let ndefs = List.fold_left acc env.Semast.env_definitions
        libdefs in
561     ( { env with Semast.env_definitions = ndefs; }, libdefs)
562
563 let direct_code_inject attrs globalenv imp sdl =
564     let s = match imp with
565       | Ast.LibraryImport(qid) -> Semast.
    string_of_qualified_id qid
566     in
567     match s with
568       | "lib" -> let (globalenv, library_defs) =
    define_libraries attrs globalenv in
569         (globalenv, library_defs @ sdl)
570       | _ -> (globalenv, sdl)
571
572 let generate_semantic attrs globalenv = function
573 | Ast.Program(dl) ->
574     let envl = [globalenv] in
575     let rec acc_ast_definitions (prefix, attrs, envl, sdl) =
    function
576       | Ast.Import(imp) ->
577         let globalenv = List.hd ( List.rev envl ) in
578         let (globalenv, sdl) = direct_code_inject attrs
    globalenv imp sdl in
579         let tail = List.tl ( List.rev envl ) in
580         (prefix, attrs, ( globalenv :: tail ), sdl)
581       | Ast.Namespace(n, dl) -> let qualname = prefix @ n in
582         let (_, attrs, envl, nssdl) = List.fold_left
    acc_ast_definitions (qualname, attrs, envl, sdl) dl in
583         ( prefix, attrs, envl, nssdl )
584       | Ast.Basic(b) ->
585         let (attrs, envl, sb) = ( generate_s_basic_definition
     prefix attrs envl b ) in
586         ( prefix, attrs, envl, sb :: sdl )
```

```
587   in
588   let (_, attrs, envl, sdefs) = List.fold_left
       acc_ast_definitions ([], attrs, envl, []) dl in
589   let globalenv = List.hd ( List.rev envl ) in
590   Semast.SProgram( attrs, globalenv, List.rev sdefs )
591
592 let modify_symbols = function
593 | Semast.SProgram(attrs, env, sdls) ->
594   let rec acc ( symbols, defs ) def =
595     match def with
596       | Semast.SBasic(Semast.SFunctionDefinition(f)) ->
597         let qualname = Semast.string_of_qualified_id f.
      Semast.func_name in
598         let sqt = ( Semast.
      type_name_of_s_function_definition f ) in
599         let nsymbols = accumulate_string_type_bindings
      symbols (qualname, sqt)
600         and ndefs = accumulate_string_type_bindings defs (
      qualname, sqt)
601         in
602         ( nsymbols, ndefs )
603       | Semast.SBasic(Semast.SVariableDefinition(v)) ->
604         ( symbols, defs )
605   in
606   let (symbols, defs) = List.fold_left acc (StringMap.empty
       , StringMap.empty) sdls in
607   let m k l r = match (l, r) with
608     | Some(l), Some(_) -> Some(l)
609     | None, ( Some(r) as s ) -> s
610     | ( Some(l) as s ), None -> s
611     | _ -> None
612   in
613   let env = { env with
614     Semast.env_symbols = StringMap.merge m symbols env.
      Semast.env_symbols;
615     Semast.env_definitions = defs;
616   }
617   in
618   Semast.SProgram( attrs, env, sdls )
619
620 let check astprogram =
621   (* Pass 1: Gather globals inside of all the namespaces
622   so they can be referenced even before they're defined (
       just so long as
623   they're in the same lateral global scope, not necessarily
```

```
624     in vertical order) *)
625     let ( attrs, env ) = generate_global_env astprogram in
626     (* Pass 2: Generate the actual Semantic Tree based on
          what
627     is inside the AST program... *)
628     print_endline "Pass 2";
629     let sprog = generate_semantic attrs env astprogram in
630     (* Pass 3: Update any symbols that were resolved during
631     bottom-up type derivation... *)
632     print_endline "Pass 3";
633     let Semast.SProgram(attrs, env, _) = modify_symbols sprog
          in
634     (* Pass 4: Finalize everything with new information *)
635     print_endline "Pass 4";
636     let sprog = generate_semantic attrs env astprogram in
637     sprog
```

../source/semant.ml

```
1  (* LePiX Language Compiler Implementation
2  Copyright (c) 2016- ThePhD
3
4  Permission is hereby granted, free of charge, to any person
       obtaining a copy of this
5  software and associated documentation files (the "Software"
       ), to deal in the Software
6  without restriction, including without limitation the
       rights to use, copy, modify,
7  merge, publish, distribute, sublicense, and/or sell copies
       of the Software, and to
8  permit persons to whom the Software is furnished to do so,
       subject to the following
9  conditions:
10
11 The above copyright notice and this permission notice shall
       be included in all copies
12 or substantial portions of the Software.
13
14 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
       KIND, EXPRESS OR IMPLIED,
15 INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
       MERCHANTABILITY, FITNESS FOR A
16 PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
       THE AUTHORS OR COPYRIGHT
17 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY
       , WHETHER IN AN ACTION
18 OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
```

```
    CONNECTION WITH THE
19 SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. *)
20
21 (* Code generation: translate takes a semantically checked
    AST and produces
22 LLVM IR:
23 http://llvm.org/docs/tutorial/index.html
24 http://llvm.moe/ocaml/ *)
25
26 (* Linked code after the c bindings from the makefile
27 compiled the ll for the c bindings *)
28
29 module StringMap = Map.Make(String)
30
31 type li_universe = {
32   lu_attrs : Semast.s_attributes;
33   lu_env : Semast.s_environment;
34   lu_module : Llvm.llmodule;
35   lu_context : Llvm.llcontext;
36   lu_builder : Llvm.llbuilder;
37   lu_variables : Llvm.llvalue StringMap.t;
38   lu_functions : Llvm.llvalue StringMap.t;
39   lu_named_values : Llvm.llvalue StringMap.t;
40   lu_named_params : Llvm.llvalue StringMap.t;
41   lu_handlers : ( li_universe -> ( Semast.s_expression list
      ) -> ( li_universe * Llvm.llvalue list ) ) StringMap.t;
42 }
43
44 let create_li_universe = function | Semast.SProgram(attrs,
    env, _) ->
45   let context = Llvm.global_context() in
46   let builder = Llvm.builder context in
47   let m = Llvm.create_module context "lepix" in
48   {
49     lu_attrs = attrs;
50     lu_env = env;
51     lu_module = m;
52     lu_context = context;
53     lu_builder = builder;
54     lu_variables = StringMap.empty;
55     lu_functions = StringMap.empty;
56     lu_named_values = StringMap.empty;
57     lu_named_params = StringMap.empty;
58     lu_handlers = StringMap.empty;
59   }
```

```
60
61  let rec llvm_type_of_s_type_name lu st =
62    let f32_t   = Llvm.float_type   lu.lu_context
63    and f64_t   = Llvm.double_type  lu.lu_context
64    (* for 'char' type to printf -- even if they resolve to
       same type, we differentiate*)
65    and char_t  = Llvm.i8_type      lu.lu_context
66    and i16_t   = Llvm.i16_type     lu.lu_context
67    and i32_t   = Llvm.i32_type     lu.lu_context
68    and i64_t   = Llvm.i64_type        lu.lu_context
69    (* LLVM treats booleans as 1-bit integers, not distinct
       types with their own true / false *)
70    and bool_t  = Llvm.i1_type      lu.lu_context
71    and void_t  = Llvm.void_type    lu.lu_context
72    in
73    let p_char_t  = Llvm.pointer_type    char_t
74    in
75    match st with
76    (* TODO: handle reference-ness (e.g., make it behave like
         a pointer here) *)
77    | Semast.SBuiltinType( Ast.Bool, tq ) -> bool_t
78    | Semast.SBuiltinType( Ast.Int(n), tq ) -> begin match n
       with
79      | 64 -> i64_t
80      | 32 -> i32_t
81      | 16 -> i16_t
82      | _ -> Llvm.integer_type lu.lu_context n
83    end
84    | Semast.SBuiltinType( Ast.Float(n), tq ) -> begin match
       n with
85      | 64 -> f64_t
86      | 32 -> f32_t
87      | 16 -> (* LLVM actually has support for this, but
       shitty OCaml bindings *)
88        (* TODO: Proper Error *)
89        raise( Failure "Cannot have a Half Float because
       OCaml binding for LLVM is garbage" )
90      | _ -> (* TODO: Proper Error *)
91        raise( Failure "Unallowed Float Width" )
92    end
93    | Semast.SBuiltinType( Ast.String, tq ) -> p_char_t
94    | Semast.SBuiltinType( Ast.Void, tq ) -> void_t
95    | Semast.SArray(t, d, tq) -> Llvm.array_type (
       llvm_type_of_s_type_name lu t) d
96    | Semast.SSizedArray(t, d, szs, tq) -> Llvm.array_type (
```

```
      llvm_type_of_s_type_name lu t) d
97    | Semast.SFunction(rt, argst, tq) ->
98      let lrt = llvm_type_of_s_type_name lu rt
99      and largst = Array.map ( llvm_type_of_s_type_name lu )
        ( Array.of_list argst )
100     in
101     Llvm.function_type lrt largst
102   | _ -> (* TODO: Proper Error *)
103     raise(Errors.Unsupported("This type is not convertible
        to an LLVM type"))
104
105 let should_reference_pointer = function
106   | Semast.SBuiltinType(Ast.String, _) -> true
107   | Semast.SArray(_, _, _) -> true
108   | Semast.SSizedArray(_, _, _, _) -> true
109   | Semast.SFunction(_, _, _) -> true
110   | _ -> false
111
112 let find_argument_handler lu target =
113   let hn = Llvm.value_name target in
114   try Some( StringMap.find hn lu.lu_handlers )
115   with _ -> None
116
117 let llvm_lookup_function lu name t =
118   let mname = Semast.mangle_name [name] t in
119   match Llvm.lookup_function mname lu.lu_module with
120     | Some(v) -> v
121     | None -> raise( Errors.FunctionLookupFailure( name,
      mname ) )
122
123 let llvm_lookup_variable lu name t =
124   match Llvm.lookup_global name lu.lu_module with
125     | Some(v) -> v
126     | None -> raise( Errors.VariableLookupFailure( name,
      name ) )
127
128 let dump_s_qualified_id lu qid t =
129   let fqn = Semast.string_of_qualified_id qid in
130   let lookup n =
131     try
132       let v = StringMap.find n lu.lu_named_values in
133       Some(v)
134     with | Not_found ->
135       try
136         let v = StringMap.find n lu.lu_named_params in
```

```
137           Some(v)
138       with | Not_found -> try
139           let v = StringMap.find n lu.lu_variables in
140           Some(v)
141       with | Not_found -> None
142     in
143     let lookup_func n =
144       try
145         let v = StringMap.find n lu.lu_named_values in
146         Some(v)
147       with | Not_found ->
148         try
149           let v = StringMap.find n lu.lu_named_params in
150           Some(v)
151       with | Not_found -> try
152           let v = StringMap.find n lu.lu_functions in
153           Some(v)
154       with | Not_found -> None
155     in
156     let overload_lookup qid ft =
157       let mangled = Semast.mangle_name qid ft in
158       begin match lookup_func mangled with
159         | Some(v) as s -> s
160         | None -> begin match lookup_func fqn with
161           | Some(v) as s -> s
162           | None -> None
163           end
164       end
165     in
166     let overload_acc op ft =
167       match op with
168         | Some(v) as s -> s
169         | None -> overload_lookup qid ft
170     in
171     let idval = match t with
172       | Semast.SFunction(rt, tnl, tq) as ft ->
173         begin match overload_lookup qid ft with
174           | Some(v) -> v
175           | None -> raise (Errors.UnknownFunction(fqn))
176         end
177       | Semast.SOverloads(fl)->
178         begin match List.fold_left overload_acc None fl with
179           | Some(v) -> v
180           | None -> raise(Errors.UnknownFunction(fqn))
181         end
```

```
182       | _ -> match lookup fqn with
183           | Some(v) -> v
184           | None -> raise (Errors.UnknownVariable fqn)
185     in
186     (lu, idval)
187
188   let dump_s_literal lu lit =
189     let f64_t   = Llvm.double_type   lu.lu_context
190     and bool_t  = Llvm.i1_type       lu.lu_context
191     in
192     let v = match lit with
193       | Semast.SBoolLit(value) -> Llvm.const_int bool_t (if
        value then 1 else 0) (* bool_t is still an integer, must
         convert *)
194       | Semast.SIntLit(value, b) -> Llvm.const_of_int64 (Llvm
        .integer_type lu.lu_context b) value true (* bool for
        signedness *)
195       | Semast.SStringLit(value) ->
196         let str = Llvm.build_global_stringptr value "str_lit"
         lu.lu_builder in
197         str
198       | Semast.SFloatLit(value, b) -> Llvm.const_float f64_t
        value
199     in
200     (lu, v)
201
202   let dump_temporary_value lu ev v =
203     let v = match ( Llvm.classify_type ( Llvm.type_of v ) )
        with
204       | Llvm.TypeKind.Pointer ->
205         if ( should_reference_pointer ev ) then
206           v
207         else
208           Llvm.build_load v "tmp" lu.lu_builder
209       | _ -> v
210     in
211     v
212
213   let dump_s_expression_temporary_gen f lu e =
214     let (lu, v) = ( f lu e ) in
215     let v = dump_temporary_value lu ( Semast.
        type_name_of_s_expression e ) v in
216     (lu, v)
217
218   let dump_arguments_gen f lu el =
```

```
219     let acc_expr (lu, vl) e =
220       let (lu, v) = dump_s_expression_temporary_gen f lu e in
221       ( lu, v :: vl )
222     in
223     let (lu, args) = List.fold_left acc_expr (lu, []) el in
224     (lu, args)
225
226   let rec dump_s_expression lu e =
227     match e with
228     | Semast.SLiteral(lit) -> dump_s_literal lu lit
229     | Semast.SQualifiedId(qid, t) ->
230       let (lu, v) = dump_s_qualified_id lu qid t in
231       (lu, v)
232     | Semast.SCall(e, el, t) ->
233       let (lu, target) = dump_s_expression lu e in
234       let oparghandler = find_argument_handler lu target in
235       let ( lu, args ) = match oparghandler with
236         | None ->
237           let ( lu, args ) = ( dump_arguments_gen (
       dump_s_expression ) lu el ) in
238           (lu, List.rev args)
239         | Some(h) ->
240           let (lu, args) = ( h lu el ) in
241           ( lu, args )
242       in
243       let arr_args = Array.of_list args in
244       let v = match t with
245         | Semast.SBuiltinType(Ast.Void, _) -> Llvm.build_call
       target arr_args "" lu.lu_builder
246         | _ -> Llvm.build_call target arr_args "tmp.call" lu.
      lu_builder
247       in
248       (lu, v)
249     | Semast.SBinaryOp(l, bop, r, t) ->
250       let (lu, lv) = dump_s_expression lu l in
251       let (lu, rv) = dump_s_expression lu r in
252       let opf = match bop with
253         | Ast.Add -> Llvm.build_add
254         | Ast.Sub -> Llvm.build_sub
255         | Ast.Mult -> Llvm.build_mul
256         | Ast.Div -> Llvm.build_sdiv
257         | _ -> raise(Errors.Unsupported("This binary
      operation type is not supported for code generation"))
258       in
259       let v = opf lv rv "tmp.bop" lu.lu_builder in
```

```
260       (lu, v)
261     | _ -> raise(Errors.Unsupported("This expression is not
          supported for code generation"))
262
263   let dump_s_expression_temporary lu e =
264     let (lu, v) = ( dump_s_expression lu e ) in
265     let v = dump_temporary_value lu ( Semast.
          type_name_of_s_expression e ) v in
266     (lu, v)
267
268   let dump_s_locals lu locals =
269     let acc lu (n, tn) =
270       let lty = llvm_type_of_s_type_name lu tn in
271       let v = Llvm.build_alloca lty n lu.lu_builder in
272       { lu with lu_named_values = StringMap.add (n) v lu.
          lu_named_values }
273     in
274     let Semast.SLocals(bl) = locals in
275     let lu = List.fold_left acc lu bl in
276     lu
277
278   let dump_s_parameters lu llfunc parameters =
279     let paramarr = Llvm.params llfunc in
280     let paraml = Array.to_list paramarr in
281     let Semast.SParameters(bl) = parameters in
282     let nameparam i p =
283       let (n, _) = ( List.nth bl i ) in
284       ( Llvm.set_value_name n p )
285     in
286     let _ = Array.iteri nameparam paramarr in
287     let acc lu (n, tn) =
288       let v = List.find ( fun p -> ( ( Llvm.value_name p ) =
          n ) ) paraml in
289       { lu with lu_named_params = StringMap.add n v lu.
          lu_named_params }
290     in
291     let lu = List.fold_left acc lu bl in
292     lu
293
294   let dump_store lu lhs lhst rhs rhst =
295     let _ = Llvm.build_store rhs lhs lu.lu_builder in
296     lhs
297
298   let dump_assignment lu lhse rhse lhst  =
299     let rhst = Semast.type_name_of_s_expression rhse in
```

```
300     let (lu, rhs) = dump_s_expression_temporary lu rhse in
301     let (lu, lhs) = dump_s_expression lu lhse in
302     let v = dump_store lu lhs lhst rhs rhst in
303     ( lu, v )
304
305   let dump_s_variable_definition lu = function
306     | Semast.SVarBinding((n, tn), rhse) -> let lhse = Semast.
        SQualifiedId([n], tn) in
307       let lhst = tn in
308       let rhst = Semast.type_name_of_s_expression rhse in
309       let (lu, rhs) = dump_s_expression_temporary lu rhse in
310       let (lu, lhs) = dump_s_expression lu lhse in
311       let v = dump_store lu lhs lhst rhs rhst in
312       ( lu, v )
313
314   let rec dump_s_general_statement lu gs =
315     let acc lu bgs =
316       dump_s_general_statement lu bgs
317     in
318     match gs with
319     | Semast.SGeneralBlock( locals, gsl ) ->
320       let lu = dump_s_locals lu locals in
321       let lu = List.fold_left acc lu gsl in
322       lu
323     | Semast.SExpressionStatement(e) ->
324       let (lu, _) = dump_s_expression lu e in
325       lu
326     | Semast.SVariableStatement(vdef) ->
327       let (lu, _) = dump_s_variable_definition lu vdef in
328       lu
329
330   let rec dump_s_statement lu s =
331     let acc lu s =
332       dump_s_statement lu s
333     in
334     match s with
335     | Semast.SBlock( locals, sl ) ->
336       let lu = dump_s_locals lu locals in
337       let lu = List.fold_left acc lu sl in
338       lu
339     | Semast.SGeneral(gs) ->
340       dump_s_general_statement lu gs
341     | Semast.SReturn(e) ->
342       let lu = match e with
343         | Semast.SNoop ->
```

```
344           let _ = Llvm.build_ret_void lu.lu_builder in
345           lu
346         | e -> let (lu, v) = dump_s_expression_temporary lu e
      in
347           let _ = Llvm.build_ret v lu.lu_builder in
348           lu
349       in
350       lu
351     | _ -> raise(Errors.Unsupported("This statement type is
      unsupported"))
352
353 let dump_s_variable_definition_global lu = function
354   | Semast.SVarBinding((n, tn), e) ->
355     let (lu, rhs) = dump_s_expression lu e in
356     let v = Llvm.define_global n rhs lu.lu_module in
357     let lu = { lu with
358       lu_variables = StringMap.add n v lu.lu_variables
359     } in
360     (*
361     let lty = llvm_type_of_s_type_name lu v in
362     let v = Llvm.declare_global lty k lu.lu_module in
363     { lu with lu_variables = StringMap.add k v lu.
      lu_variables }
364     let v = llvm_lookup_variable lu n tn in
365     let _ = Llvm.set_initializer v rhs in
366     let lu = { lu with lu_variables = StringMap.add n v lu.
      lu_variables } in
367     *)
368     ( lu, v )
369
370 let dump_s_function_definition lu f =
371   let acc lu s =
372     dump_s_statement lu s
373   in
374   (* Generate the function with its signature *)
375   (* Which means we just look it up in the llvm module *)
376   let ft = Semast.type_name_of_s_function_definition f in
377   let n = Semast.string_of_qualified_id f.Semast.func_name
      in
378   let llfunc = llvm_lookup_function lu n ft in
379   (* generate the body *)
380   let entryblock = Llvm.append_block lu.lu_context "entry"
      llfunc in
381     Llvm.position_at_end entryblock lu.lu_builder;
382   let lu = dump_s_parameters lu llfunc f.Semast.
```

```
      func_parameters in
383   let lu = List.fold_left acc lu f.Semast.func_body in
384   let lu = { lu with
385     lu_named_params = StringMap.empty;
386   } in
387   lu
388
389 let dump_s_basic_definition lu = function
390   | Semast.SVariableDefinition(v) -> let (lu, _) =
      dump_s_variable_definition_global lu v in
391     lu
392   | Semast.SFunctionDefinition(f) ->
      dump_s_function_definition lu f
393
394 let dump_s_definition lu = function
395   | Semast.SBasic(b) -> dump_s_basic_definition lu b
396
397 let dump_array_prelude lu =
398   (* Unfortunately, unsupported... *)
399   lu
400
401 let dump_parallelism_prelude lu =
402   (* Unfortunately, unsupported... *)
403   lu
404
405 let dump_global_string lu n v =
406   let rhs = Llvm.const_stringz lu.lu_context v in
407   let v = Llvm.define_global n rhs lu.lu_module in
408   (lu, v)
409
410 let dump_builtin_lib lu =
411   let char_t  = Llvm.i8_type      lu.lu_context
412   and i32_t   = Llvm.i32_type     lu.lu_context
413   (* LLVM treats booleans as 1-bit integers, not distinct
      types with their own true / false *)
414   in
415   let p_char_t  = Llvm.pointer_type    char_t
416   and llzero = Llvm.const_int i32_t 0
417   in
418   let f_acc lu (n, lv) =
419     { lu with lu_functions = StringMap.add n lv lu.
      lu_functions }
420   in
421   let print_lib lu =
422     let printf_t = Llvm.var_arg_function_type i32_t [|
```

```
p_char_t |] in
423   let printf_func = Llvm.declare_function "printf"
      printf_t lu.lu_module in
424   let (_, int_format_str) = dump_global_string lu "__ifmt
      " "%d"
425   and (_, str_format_str) = dump_global_string lu "__sfmt
      " "%s"
426   and (_, float_format_str) = dump_global_string lu "
      __ffmt" "%f"
427   in
428   let handler_name = "printf" in
429   let handler lu el =
430     let ( lu, exprl ) = ( dump_arguments_gen (
      dump_s_expression ) lu el ) in
431     if (List.length el) < 1 then ( lu, exprl ) else
432     let hdt = Semast.type_name_of_s_expression ( List.hd
      el ) in
433     let insertion = match hdt with
434       | Semast.SBuiltinType(Ast.String, _) ->
      str_format_str
435       | Semast.SBuiltinType(Ast.Float(n), _) ->
      float_format_str
436       | Semast.SBuiltinType(Ast.Int(n), _) ->
      int_format_str
437       | _ -> raise(Errors.BadPrintfArgument)
438     in
439     let fptr = Llvm.build_gep insertion [| llzero; llzero
       |] "tmp.fmt" lu.lu_builder in
440     ( lu, fptr :: exprl )
441   in
442   let libprintfuncs = [
443     (( Semast.mangle_name ["lib"; "print"] ( Semast.
      SFunction(Semast.void_t, [Semast.string_t], Semast.
      no_qualifiers)) ), printf_func);
444     (( Semast.mangle_name ["lib"; "print"] ( Semast.
      SFunction(Semast.void_t, [Semast.float64_t], Semast.
      no_qualifiers)) ), printf_func);
445     (( Semast.mangle_name ["lib"; "print"] ( Semast.
      SFunction(Semast.void_t, [Semast.int32_t], Semast.
      no_qualifiers)) ), printf_func);
446   ] in
447   let lu = List.fold_left f_acc lu libprintfuncs in
448   let lu = { lu with
449     lu_handlers = ( StringMap.add handler_name ( handler
      ) lu.lu_handlers )
```

```
450       } in
451       lu
452     in
453     let math_lib lu =
454       lu
455     in
456     let lu = print_lib lu in
457     let lu = math_lib lu in
458     lu
459
460   let dump_builtin_module lu = function
461     | Semast.Lib -> dump_builtin_lib lu
462
463   let dump_module_import lu = function
464     | Semast.SBuiltin(lib) -> dump_builtin_module lu lib
465     | Semast.SCode(_) -> lu
466     | Semast.SDynamic(_) -> lu
467
468   let dump_declarations lu =
469     let rec declare k lu t = match t with
470       | Semast.SOverloads(fl) ->
471         ( List.fold_left (declare k) lu fl )
472       | Semast.SFunction(rt,args,tq) as ft -> let lty =
      llvm_type_of_s_type_name lu t in
473         let mk = Semast.mangle_name [k] ft in
474         let v = Llvm.declare_function mk lty lu.lu_module in
475         { lu with lu_functions = StringMap.add mk v lu.
      lu_functions }
476       | _ -> lu
477     in
478     let acc_def k t lu =
479       declare k lu t
480     in
481     let toplevel = lu.lu_env.Semast.env_definitions in
482     let lu = StringMap.fold acc_def toplevel lu in
483     lu
484
485   let dump_prelude lu sprog =
486     let lu = dump_array_prelude lu in
487     let lu = dump_parallelism_prelude lu in
488     let lu = List.fold_left dump_module_import lu lu.lu_env.
      Semast.env_imports in
489     let lu = dump_declarations lu in
490     lu
491
```

```
492 let generate sprog =
493   let acc_def lu d =
494     let lu = dump_s_definition lu d in
495     lu
496   in
497   let lu = create_li_universe sprog in
498   let lu = dump_prelude lu sprog in
499   let lu = match sprog with
500     | Semast.SProgram(_, _, defs) -> ( List.fold_left
      acc_def lu defs )
501   in
502   lu.lu_module
```

../source/codegen.ml