PLS
COMS W4115

Bennett Rummel
bwr2116@columbia.edu

August 22, 2014

Contents

Chapter 1

Introduction to PLS

1.1 Description

PLS (PipeLine Script) is a small, imperative, dynamically typed, statically scoped, programming language designed to process streams of data. It allows the user to construct pipelines for filtering or otherwise transforming sequences of data.

1.2 Background

PLS is designed around the concept of *Kahn processes*[1]. To construct a PLS program, the user defines a set of *kernels* (a discrete unit of processing) and a set of single-writer, single-reader FIFO channels through which the kernels communicate. By connecting the kernels via channels, the user can build up a complex network of discrete processing units to perform complex computiations. PLS programs are deterministic — the same output will always be produced by the same input for a given program.

1.3 Motivation

Many programming languages and tools provide simple mechanisms for serially processing data in a small amount of code. However, few tools provide easy-to-use methods of constructing parallel programs for processing certain data sets. PLS is designed to be such a tool and allow users to write small programs for performing parallel computations across data sets.

1.4 Design Goals

PLS is designed to be a simple yet powerful standalone language for parallel programming that looks and feels familiar to many programmers.

A parallel program in PLS consists of a set of kernels and *channels* through which they communicate. Parallel execution is handled automatically by the runtime, freeing the programmer of the need to worry about such issues at runtime.

Writing a kernel is easy and is very similar to writing a function. The user needs only to learn about two special functions, `read` and `write`, in order to start writing useful programs.

PLS syntax is similar to that of C so it should feel familiar to many programmers. The behavior, however, is adapted to fit a scripting language environment and provides features not in C such as dynamic typing. Like C, PLS is statically scoped, allowing programmers to reason about sections of code in isolation. PLS provides many common operators and expressions found in C-like languages. It also provides many common data types and the ability to construct heterogenously typed lists of values.

PLS programs are compiled to a custom bytecode which can be executed with the provided runtime program. By default, PLS programs read from *stdin* and write to *stdout*, making them suitable for use inside shell scripts or other similar applications.

Chapter 2

Language Tutorial

2.1 Running These Examples

To run these examples, simply write the code to a file and call the PLS executable with the file name as the argument.

2.2 Hello World

The basic "hello world" program is very simple in PLS. We simply call the `print` function with the string `"hello world!"` as an argument:

Listing 2.1: hello_world.pls

```
print("hello world!");
```

2.3 Basic Pipelines

A "pipeline" is a sequence of `kernel`s connected by single-reader, single-writer FIFO queues. In the following example, we define a simple pipeline consisting of a single `kernel` called `plus_one`. The `plus_one` kernel has a single input channel, `input`, and a single output channel, `output`. The body of the kernel consists of an infinite loop during which it reads a single value from the input channel, adds one to the value, and writes the new value to the output channel. We then construct a simple pipeline by instantiating the `plus_one` kernel with `stdin` connected to `input`, and `stdout` connected to `output`.

Listing 2.2: plus_one.pls

```
kernel plus_one(in input, out output)
{
    while (true)
    {
        token = read(input);  // Read a value from the input channel
        token = token + 1;    // Add one to the value
        write(output, token); // Write the new value to the output channel
    }
}
```

```
plus_one(stdin, stdout);
```

Notice that there are no explicit variable declarations in the body of the kernel; the variable `local` is declared and defined when it is first assigned to. The program will terminate when there is no more input that can be processed. In this case, that happens when end-of-file is detected on `stdin`.

2.4 Basic Functions

We can define functions with arbitrary numbers of arguments. In the following example, we factor out the addition from the `plus_one` kernel in to a function called `add`, a function taking two arguments, `x` and `y` and returning their sum.

Listing 2.3: plus_one_func.pls

```
function add(x, y)
{
    return x + y;
}

kernel plus_one(in input, out output)
{
    while (true)
    {
        token = read(input);    // Read a value from the input channel
        token = add(token, 1);  // Add one to the value using the add function
        write(output, token);   // Write the new value to the output channel
    }
}

plus_one(stdin, stdout);
```

2.5 More Complicated Pipelines

A program can consist of multiple kernel instances with user-defined channels. In the following expample, we define two kernels in an over-elaborate scheme to compute the sum of each pair of values in two lists. The first kernel is `write_list` which takes a list of values as an argument and writes all of the elements to an output channel in order. The second is `sum_pair` which takes two input channels and writes the sum of the values from each to an output channel. We instantiate two separate instances of `write_list` to feed the values to the `sum_pair` instance. We also declare two channels, `a` and `b`, to pass data from the `write_list` instances.

Listing 2.4: sum_pairs.pls

```
kernel write_list(lst, out output)
{
    foreach(item : lst)
    {
        write(output, item);
    }
}

kernel sum_pairs(in first, in second, out output)
{
    while(true)
    {
        first_value = read(first);   // Read a value from the first input
        second_value = read(second); // Read a value from the second input
```

```
        // Write the sums to the output
        write(output, first_value + second_value);
    }
}

channel a, b;

write_list([1, 2, 3, 4, 5], a);
write_list([6, 7, 8, 9, 10], b);
sum_pairs(a, b, stdout);
```

This example illustrates several additional concepts. First, a kernel can have inputs without outputs and vice-versa. Second, the program does not have to take input if all of the necessary data is hard coded. In this case, the program terminates when no more processing is possible because there is no more input.

2.6 Feedback

It is possible to feed the output of a kernel back to one of its inputs (directly or indirectly). In the following example, we define a single kernel `fibonacci` to calculate a sequence of Fibonacci numbers. The kernel takes as parameters the length of the sequence to calculate (`n`), the first two values in the sequence `seed_1` and `seed_2`, an input channel populated with the previously computed values (`source`), an output channel to feed the previous and current values back to the kernel (`feedback`), and another output channel to which to write the sequence.

Listing 2.5: fib.pls

```
channel loop_channel;

kernel fibonacci(n, seed_1, seed_2, in source, out feedback, out output)
{
    // seed the input with the values
    write(feedback, seed_1);
    write(feedback, seed_2);

    // write the first two values as output
    write(output, seed_1);
    write(output, seed_2);

    for (counter = 2; counter <= n; counter = counter + 1)
    {
        // read the two values
        f_1 = read(source);
        f_2 = read(source);

        // calculate the next number
        fib = f_1 + f_2;

        // write the number to the output
        write(output, fib);

        // write the next two numbers back in to the feedback loop
        write(feedback, f_2);
        write(feedback, fib);
    }
}

// print the first 10 fibonacci numbers (starting with 0 and 1) to stdout
fibonacci(10, 0, 1, loop_channel, loop_channel, stdout);
```

Chapter 3

Language Reference Manual

3.1 Introduction

PLS (PipeLine Script) is intended to be a small, imperative, dynamically typed, statically scoped, programming language designed to process streams of data. It allows users to construct pipelines for processing data based on the concept of Kahn processes [1]. A pipeline is made up of a sequence of discrete processes called `kernel`s. `kernel`s communicate through single direction FIFO pipes. A program is made of a sequence of one or more `kernel`s.

3.2 Syntax Notation

In the following portion of this manual depicting syntax, non-terminal symbols are displayed in *italics* while terminal symbols are displayed in `fixed width` or inside 'single quotes'.

3.3 Lexical Conventions

3.3.1 Comments

Comments begin with the characters "\\" and continue until the end of the current line.

3.3.2 Identifiers

An identifier is a sequence of letters, digits, and underscores ("_") beginning with a letter or an underscore. Identifiers in PLS are case-sensitive and can be of any length greater than or equal to one character.

3.3.3 Reserved Identifiers

The following identifiers are reserved and may not be specified as identifiers otherwise:

```
if      elseif    else
for     while     foreach
true    false     null
return  kernal    main
print   len       append
read    write     in
out     function
```

3.3.4 Literals

Integers

An integer constant is represented by one or more decimal (*[0 – 9]*) digits.

Floating Point Numbers

A floating point number constant follows the same format as the C programming language in [2]:

> A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

Boolean Values

Boolean constants are either `true` or `false`.

Strings

Strings are represented by any sequence of characters surrounded by double quotes ("`"`"). A double-quote character in a string must be escaped by a preceding backslash ("\"). Strings may not contain new line characters instead the character sequence "\`n`" should be used.

Lists

A list is declared using the follow syntax:

list	::=	'`[`' *exprlist* '`]`'
exprlist	::=	*exprlist* '`,`' *expr*
	|	*expr*
	|	*empty*

The list will contain copies of the values resulting from the expressions in *exprlist*.

3.4 Data Types and Conversions

3.4.1 Integers

Integers are represented by the platform's signed 64-bit integer types. Integers may be converted to a floating point number without loss of precision.

3.4.2 Floating Point Numbers

Floating point values (hereafter "floats") are represented by the platform's double precision IEEE-754 type (e.g. `double`). Converting a floating point value to an integer will cause it be truncated towards 0. If a floating point value is outside the range of values able to be represented by an integer, the result of the conversion is undefined.

3.4.3 Strings

Strings are represented by a sequence of characters of an arbitrary length. Strings may not be converted to other data types.

3.4.4 Boolean Values

Simple true/false type. Integers and floats will be evaluated as `true` in a boolean context if they are non-zero, and `false` otherwise. `null` is always evaluated to `false`. All other types are always converted to boolean `true`.

3.4.5 Lists

A list is a composite type representing an ordered sequence of 0 or more elements of heterogenous types. Elements in a list are referred to by integer indices starting at zero. Items can be added to a list but not changed or removed. Lists can also be nested to allow for multi-dimensional lists.

3.4.6 `null`

`null` represents the abscence of a value.

3.5 Variables and Objects

Variables in PLS are dynamically typed and are declared by assigning them a value (see below). A variable's value can be changed (including the data type) by assigning a new value. Functions and kernels can not be assigned to a variable unless otherwise indicated.

3.6 Expressions

3.6.1 Primary Expressions

Primary expressions are the basic "building blocks" of other expressions. That is, only a primary expression can result in a primary expression. Primary expressions are defined as:

primary-expr ::= *identifier*
| *literal*
| '`(`' *expr* '`)`'
| *primary-expr* '`[`' *expr* '`]`'
| *identifier* '`(`' *expr-list* '`)`'

expr-list ::= *expr-list* '`,`' *expr*
| *expr*
| *empty*

identifier

An identifier can be used as an expression if it exists in a scope visible from the current scope (see below). The expression results in the type and value of *identifier*.

literal

A constant can be used as an expression and its type is the most appropriate given how the exact string specified in the source code is parsed (see above).

`(` *expr* `)`

Simply yields *expression*.

primary-expr `[` *expr* `]`

A primary expression followed by an integer expression surrounded by square brackets is used to refer to a specific element in a list. The result of *expr* is used to retrieve the n'element of *primary-expr*, counting from zero. The result of the expression is a reference to the element, meaning the list can be changed by assigning a new value to the result of the expression (see below). It is an error if *expression* is not an integer, is less than 0, or is greater than or equal to the length of the list.

identifier `(` *expr-list* `)`

An identifier followed by a pair of parentheses containing an optional comma-separated list of expressions is used to call a function named *identifier* with the results of the specified *expr-list*'s expressions as arguments. The result of the expression is the return value of the function or `null` if the function does not return a value. See below for more information.

3.6.2 Unary Operators

-expression

If *expression* is an integer or a float, the expression computes the negative of *expression*. It is an error for *expression* to be of any other type.

!expression

If *expression* is a boolean, the expression computes the logical negation of *expression*.

3.6.3 Binary Operators

In the following operations, an expression in which one operand is an integer and the other is a float will invoke an implicit conversion of the integer to a float (see above) before the operation is performed unless otherwise indicated. Additionally, if any expression would result in a value outside the range able to be represented by its type, the result in undefined. Finally, the following operators are left-associative unless otherwise indicated.

expression* + *expression

If both expressions are numeric types (i.e. integers or floats), the expression yields the sum of the two values. If the first expression is a string, the expression yields the concatenation of the first string with the string representation of the second expression.

expression* - *expression

Computes the difference of the two operands if both are numeric types. Otherwise, it is an error.

expression* * *expression

Computes the product of the two operands if both are numeric types. Otherwise, it is an error.

expression* / *expression

Computes the quotient of the two operands. If both operands are integers, the fractional part of the result may be truncated towards 0. It is an error if the types of any operand is not numeric.

expression* % *expression

Computes the remainder of dividing the first expression by the second (i.e. the value of the first expression modulo the value of the second). It is an error unless both operands are numeric types.

identifier* = *expression

Assign to the variable as the left operand the result of the right operand *expression*. This operator is right-associative. If the left operand is an identifier that is not bound, a new one is allocated and set to the value of *expression*. If the identifier is bound in the current scope, its value is replaced. The identifier is only bound inside the scope in which it is first assigned to. See below for more information. The result of the expression is that of *expression*.

expression* == *expression

Comparison operator yields `true` if both operands represent the same value (after necessary type conversions), and `false` otherwise.

expression* != *expression

Yields `true` of the operands do not represent the same value, `false` otherwise.

expression* < *expression

Yields `true` if both operands are numeric types and the value of the first is strictly less than the value of the second. It is an error if either type is not numeric.

expression* > *expression

Yields `true` if both operands are numeric types and the value of the first is strictly greater than the value of the second. It is an error if either type is not numeric.

expression* <= *expression

Yields `true` if both operands are numeric types and the value of the first is less than or equal to the value of the second. It is an error if either type is not numeric.

expression* >= *expression

Yields `true` if both operands are numeric types and the value of the first is greater than or equal to the value of the second. It is an error if either type is not numeric.

expression* && *expression

Computes the logical “and” of both expressions. It is an error if either expression is not convertable to a boolean. Note: in the current implementation the operator does not “short circuit”.

expression* || *expression

Computes the logical “or” of both expressions. It is an error if either expression is not convertable to a boolean. Note: in the current implementation the operator does not “short circuit”.

3.7 Statements

Statements come in several different forms:

stmt	::=	*expression* '`;`'
	|	*block*
	|	*if-stmt*
	|	*while-stmt*
	|	*for-stmt*
	|	*foreach-stmt*
	|	*return-stmt*

3.7.1 Expressions

A statement may simply consist of an *expression* followed by a semicolon.

3.7.2 Blocks

Statements may also take the form:

block	::=	*stmt-list*
stmt-list	::=	*empty*
	|	*stmt-list stmt*

Such a statement may be useful for improving code readability or limiting the scope of a variable (see below). Blocks are also required as part of other statements.

3.7.3 `if` Statement

A `if` statement has the form:

if-stmt	::=	'`if`' '`(`' *expr* '`)`' *block else*
else	::=	*empty*
	|	*elseif*
	|	'`else`' *block*
elseif	::=	'`elseif`' '`(`' *expr* '`)`' *block else*

If the *expr* in the *if-stmt* evaluates to `true`, then the first *block* is executed. Otherwise it falls through to the next group in the statement. This continues for each (optional) `elseif` and its *expr*, and *block* group. If none of the *expr*s evaluate to `true`, then the *block* following the (also optional) `else` is executed. It is important to note that only the *expr*s up to the first one that results in `true` will be evaluated.

3.7.4 `while` Loop Statement

The `while` statement has the form:

while-stmt	::=	'`while`' '`(`' *expr* '`)`' *block*

If the *expr* evaluates to true, then *block* is executed once. Then the cycle repeats until *expr* evaluates to false.

3.7.5 `for` Loop Statement

The `for` statement has the form:

for-stmt ::= '`for`' '`(`' *expr* '`;`' *expr* '`;`' *expr* '`)`' *block*

In this statement, the first *expr* is executed once before anything else. Any new variables are local to the statement (see below). The second *expr* is the termination test — if it evaluates to `true`, then the *block* is executed once. The third *expr* specifies an operation to be performed after each execution of *block*. After *block* and the third *expr* are executed and evaluated once, the test is checked again and the cycle repeats. Any of the *expr*s may be the empty expression. If the first *expr* declares a new variable, the scope of that variable is the body of the loop.

3.7.6 `foreach` Loop Statement

The `foreach` statement has the form:

foreach-stmt ::= '`foreach`' '`(`' *identifier* '`:`' *expr* '`)`' *block*

In this statement, *expr* must produce a list. *identifier* is bound to a copy of each element of the list starting with index 0 and proceeding in order to the final element. *block* is executed once for element in the list. The scope of *identifier* is the body of the loop.

3.7.7 `return` Statement

The `return` statement has the form:

return-stmt ::= '`return`' *expr* '`;`'

It is valid only in a function definition (see below). When a `return` statement is executed, the value of *expr* is used as the result of the function call expression (see above) and control is returned to the caller.

3.8 Functions

3.8.1 Function Definitions

A function definition has the form:

function-decl ::= '`function`' *identifier* '`(`' *arg-list* '`)`' *block*

arg-list ::= *empty*
| *id-list*

id-list ::= *identifier*
| *id-list* '`,`' *identifier*

where *arg-list* is the optional comma-separated list of identifiers representing the parameters to the function. The function name *identifier* is bound to the function. Functions cannot carry state from one invocation to the next. Functions can optionally contain one or more `return` statements to return a value to the caller (see above). If control reaches a `return` statement, then control is returned to the caller. If control reaches the end of the body of the function, `null` is returned (i.e. there is an implicit "`return null;`" at the end of each function definition). Function arguments are passed by value, meaning a function can not directly affect the value of its parameters when called (except if the function is passed a global variable, see below).

Functions cannot be nested (they must be defined in global scope), preventing constructs such as closures. Additionally, they are not first-class types and can therefore not be assigned or passed to other functions.

3.8.2 Special Functions

There are several builtin functions defined.

`print(x)`

`print` is simply prints an ASCII representation of its single argument to `stdout` and returns `null`.

`len(l)`

`len` returns an integer containing the number of elements in `l` if `l` is a list, or `null` otherwise.

`append(l, x)`

Append `x` to the end of list `l`. Returns `l`.

`read(c)`

`read` is a special function used to pop a single token from a `kernel`'s input pipe `c`. If no tokens are available, the call will block until one is available. Calling `read` from anywhere but a `kernel` definition is an error.

`write(c, x)`

`write` is a special function used to push a single token `x` to a `kernel`'s output pipe `c`. `write` will not block. Calling `write` from any context but a `kernel` definition is an error.

3.9 Kernels

A `kernel` is a discrete processing unit. It communicates with other `kernel`s via single direction FIFO queues. `kernel`s appear similar but have different semantics than functions.

3.9.1 Kernel Definition

A `kernel` definition has the form:

kernel-decl ::= '`kernel`' *identifier* '`(`' *kernel-arg-list* '`)`' *block*

kernel-arg-list ::= *kernel-arg*
| *kernel-arg-list* '`,`' *kernel-arg*

kernel-arg-list ::= *identifier*
| '`in`' *identifier*
| '`out`' *identifier*

Where *identifier* is the name of the `kernel`. A kernel argument is a normal argument (like a function argument) or a channel argument. A channel argument specifies an input or output channel and consists of a direction (`in` or `out` for input and output channels, respectively) and an identifier used to reference the channel in the body *block* of the kernel.

3.9.2 Kernel Input and Output

There are two special functions available for use inside of a `kernel` body: `read` and `write` (see above). It is important to note that `kernel` inputs and outputs need not be symmetric, e.g. a `kernel` could read two tokens for every one that it outputs. Additionally, the communication pipes are unbounded. Finally, there is no way for a `kernel` to check if a pipe has any tokens or is empty.

3.10 Scope

Variables are lexically scoped, meaning they are valid within the scope in which they are first declared, including nested "child" scopes. Variables can also be declared in the global scope and are visible in every subsequent scope. Function parameters have the scope of the function and will hide any other variable or function with the same name. Using an identifier before it has been declared and made active via assignment will result in an error.

3.11 Channels

Channels are named FIFO-style queues used to pass data between kernels. There are two builtin channels that can be used as kernel arguments: `stdin` and `stdout`. `stdin` is populated with tokens taken from *stdin* when the program is executed and can only be used as an input channel. `stdout` is an output channel that simply prints its contents to *stdout* when the program.

Other channels must be explicitly declared in the global scope before they can be used using a channel declaration:

channel-decl ::= '`channel`' *channel-list* '`;`'

channel-list ::= *identifier*
| *channel-list* '`,`' *identifier*

Like other identifiers, channel names must be unique.

3.12 Pipelines

A pipeline is constructed via a series of kernel instantiations. A kernel instantiation looks exactly like a function call but is only valid in the global scope:

kernel-instantiation ::= *identifier* '`(`' *instance-args* '`)`'

instance-args ::= *identifier*
| *instance-args* '`,`' *identifier*

Where the *identifier* is the name of the kernel being instantiated and *instance-args* is a comma-separated list of arguments (both normal and channel arguments) matching the kernel definition. The rules are the same as calling a function except that valid channel names must be passed to the channel arguments. An error will be reported if more than one kernel instance uses the same channel as an input or an output channel argument (i.e. each end of the channel can only be used once). Feedback between one or more channels and kernels is allowed, permitting a type of recursion.

3.13 Programs

A program is simply a set of zero or more kernel instantiations and global statements. The global statements are executed before the kernel bodies are run. An unspecified algorithm is used to pick the next kernel to execute. If, during the execution of a kernel body, a `read` call is encountered for which the associated channel is empty, execution of that kernel instance is temporarily suspended to allow other kernels to run. The exception to this rule is when reading from the `stdin` channel; in this case, the entire program is suspended until a token or EOF is read from *stdin*. The program terminates when every kernel instance has reached the end of its body and all of the non-finished kernels are waiting on a `read` call.

Because PLS is designed around the concept of Kahn processes, programs are deterministic, meaning a program will always produce the same sequence of output values given the same input. However, the exact ordering or operations across kernels within a program is unspecified to allow for parallel computation.

Chapter 4

Project Plan

4.1 Processes

Working as the only person on this project allowed me to approach the project in a way that was much more comfortable for me as opposed to a way that would benefit a group. When I sat down to work, I tended to pick whatever portion of the project that sounded interesting at the moment and needed to be done rather than following a stringent plan for the day. That being said, there were some common themes to the way in which I developed the project and the code.

When designing, planning, and specifying my language, I started thinking about the kind of language (in terms of syntax, etc.) I would be comfortable using. From there I thought about the kind of features I would want to have in such a language while attempting to keep in mind the time constraints involved with the development. If in doubt, I tried to stay reasonably similar to C in appearance and behavior (exceptions noted in the the LRM in Chapter 3). After the initial specifications and design were complete, it was a matter of going back and refining the design where appropriate. I was also not afraid to change parts of the design during development if my initial assumptions about, e.g. ease of implementation, were wrong.

When actually developing my compiler, I tended to pick a feature, implement it as completely as possible, and write a basic test to prove that it worked before moving on to the next. Since I had a relatively good idea of what I wanted to implement before I started coding, this meant I rarely had to go back and fix old bugs; if I encountered a bug or a test case failing, it was more often than not due to something I was working on at the moment rather than something I had already finished.

After my compiler was feature complete, I went back to flesh out existing test cases and add more as an added assurance that everything was working correctly. My tests are described more in Chapter 6.

4.2 Style Guide

This project comprised my first OCaml program. As such, I did not have a good sense of what "good" style is for OCaml. Thus, my coding style is a combination of styles developed while working on other projects in other languages, styles observed in various OCaml programs around the web, and styles developed on the spot that seemed reasonable at the time. Below is a list of guidelines in no particular order:

Line length Lines should be no more than 80 characters in length

Indentation Indentation should not use tab characters and should be four spaces. The exception is when using type matching: the type being matched should be indented four spaces, but the "|" character should be indented two.

Naming Conventions Variables, functions, and types should use all lower case with underscores separating words (e.g. `do_something`, `interesting_variable`). Constructors should capitalize each word without underscores (e.g. `ExprStmt`, `Literal`)

Comments Multiline comments should occur within a single `(* *)` pair but the beginning of each line after the first should start with a * aligned with the * of the comment's opening. Functions and key parts of algorithms should be commented. Other comments are discretionary.

Mutable Values Mutable values (arrays, refs, etc.) should be avoided where possible unless the alternative is too ugly or difficult. This is mostly a judgement call.

Nested and Global Functions Functions should be declared in the most-nested scope in which they are needed. In other words, don't make every function global; only use globals for the public interface to a module or standalone utility functions.

Library Functions Library functions should be used if possible and reasonable.

Compiler Warnings The code should not cause the compiler to issue any warnings during builds (i.e. pretend GCC's `-Werror` is in use).

Loops Avoid explicit for and while loops in OCaml code.

Tail Recursion Tail recursion should be preferred to obtain reasonable performance.

Alignment Align code structures where possible, e.g. the `->` symbols in a `match`. If some construct can not fit on one line, its individual parts should be aligned and indented. For example:

```
%[language=ml]
        let env = {
            x        = 1;
            fortytwo = 42;
            (* snip *)
        }
```

Error Messages Error messages and diagnostics should give the user at least a basic idea of what went wrong.

Consistency Try to be consistent even in violation of the above rules where appropriate.

4.3 Project Timeline

Below is a list of planned implementation dates for various parts of the project.

Date	Goal
06/11/2014	Basic language design and proposal
07/02/2014	LRM; language features outlined; initial scanner, parser, and AST implementations
07/20/2014	Basic compiler, bytecode, and VM support with tests for implemented features
07/27/2014	Kernel and pipeline support
08/03/2014	Code feature complete, test suite expansion
08/17/2014	Code cleanup, bugfixes, and documentation
08/22/2014	Project complete

4.4 Development Environment

Development was performed on an x86 system running Arch Linux[1]. All of the compiler code is written in OCaml[2], version 4.01.0. Bash[3] was used to script the test driver. GNU Make 4.0[4] was used to coordinate builds. Git version 2.0.0[5] was used as the version control system.

4.5 Project Log

Below is a project log. Multiple entries on the same date have been combined in to one row for clarity. Additionally, entries have been cleaned up to make more sense to the reader.

Date	Comment
06/29/2014	Initial scanner, parser, AST, and "main" implementations
07/11/2014	Improved parser and AST implementations for statements, expressions, and function declarations
07/12/2014	More parser and AST work; AST pretty-printing functions
07/13/2014	Initial tests for basic statements; Compiler and bytecode initial implementations with basic code generation
07/16/2014	Variable lookup
07/18/2014	Lexical scoping implementation
07/19/2014	Function call support with tests; kernel implementation with tests; VM skeleton with partial implementation
07/20/2014	VM functional
07/26/2014	Rewrite variable lookup to better support global scoping
07/27/2014	Miscellaneous bug fixes and cleanup
07/28/2014	Initial "parallel" kernel support
07/29/2014	Improved kernel/pipeline support
07/30/2014	Miscellaneous bug fixes, cleanups and tweaks; cleaner kernel/pipeline support; add command line options
08/02/2014	Improved kernel/pipeline support
08/03/2014	Final pipeline implementation; lots more tests and test driver; stdin scanner for more interesting programs; comment compiler
08/04/2014	Immutable list support; `elseif` support
08/05/2014	`foreach` loop support; fix string handling to support embedded quotes in literals; more tests; improve Makefile a bit;
08/10/2014	Minor parser/AST cleanup to better support list assignment syntax
08/16/2014	Minor bug fixes; implemented forgotten mod operator `%`; improve test suite
08/18/2014	Very minor code cleanups and documentation fixes

[1] https://www.archlinux.org
[2] http://ocaml.org
[3] http://www.gnu.org/software/bash/
[4] http://www.gnu.org/software/make/
[5] http://git-scm.com/

Chapter 5

Architectural Design

The PLS compiler is implemented as a basic compiler that generates a custom bytecode as output. It consists of fairly standard components as shown in Figure 5.1. Each of the components is documented in more detail below. It also contains a virtual machine implementation to execute the compiled program.

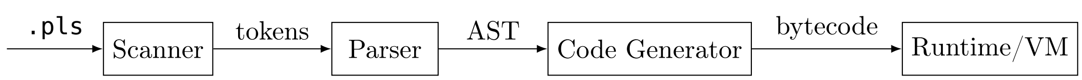

Figure 5.1: PLS architecture diagram

5.1 Scanner

The scanner is implemented as a standard `ocamllex` in `scanner.mll` (see Listing A.1). It matches basic operators (e.g. `==`, `+`, etc.), parentheses, brackets, syntactically significant punctuation, keywords (`if`, `else`, `while`, etc.), and literals. It contains special rules for comments and string literals in order to support escaped sequences such as quotes (e.g. `"hello \"world\""`). It outputs a sequence tokens to the parser.

5.2 Parser

The parser is implemented as a standard `ocamlyacc` parser in `parser.mly` (see Listing A.2). The parser outputs an abstract syntax tree to the code generator in the form of an `Ast.program` (see Listing A.3). The parser does not do anything beyond constructing the AST and ensuring that the input is accepted by the grammar.

5.3 Code Generator and Bytecode

The code generator is implemented in `compiler.ml` (see Listing A.5). The front end calls the `Compiler.compile` function with the AST from the parser and is returned the compiled program in bytecode form along with some metadata needed by the VM.

5.3.1 Bytecode

A custom bytecode language was implemented for this project. It is defined in `bytecode.ml` (see Listing A.4 for source code and more information).

Bytecodes operate on a stack in PLS. Each entry in the stack is of type `Bytecode.value`, which can represent any type expressible in a PLS program. The bytecodes are type agnostic, i.e. there are not different bytecodes for different types (except for those that operate on lists) and types are normalized at runtime. Bytecodes usually take zero or one arguments and almost always operate at the top of the stack.

Most of the bytecodes are relatively straightforward and/or self-explanatory. `Push` and `Pop` manipulate the stack pointer. `Add`, `Sub`, `Mul`, `Div`, and `Mod` mode perform arithmetic while others (such as `Ceq` and `Cgt`) perform comparison operations. Global and local variable store and load operations are implemented seperately: `Strl` and `Ldl` store and load respectively relative to the frame pointer while `Strg` and `Ldg` operate on global variables. Only relative branching operations are implemented (`Bz`, `Bnz`, and `Ba`). List operations get their own bytecodes (`Ilst`, `Alst`, and `Mlst`) as well do kernel IO primitives (`Read` and `Write`). Function-specific operations (`Call`, `Ent` and `Ret`) and their kernel equivalents (`Par`, `Kent`, and `Term`) are separate from each other. Finally, a special bytecode, `Run`, starts the "parallel" execution of kernel instances.

Some of the bytecodes may be redundant (e.g. `Ent` and `Kent`) but were implemented separately in order to ease development. They may have been consolidated if time permitted.

5.3.2 Statement and Expression Compilation

At a sufficiently low level, a PLS program is made up of a sequence of statements and expressions. Since these are the common building blocks of a PLS program, functions are provided to compile an individual statement or expression to bytecode. These functions are `compile_stmt` and `compile_expr`. Both share a similar interface: they take the current symbol table and the statement or expression and return a tuple containing the bytecode and updated symbol table.

The symbol table is returned for two reasons. First, since there are no explicit variable declarations (ignoring channel declarations), a given statement could potentially add an identifier that would need to be visible in subsequent statements if an unused identifier occurs on the left-hand side of an assignment expression (i.e. variables are declared when first defined). An updated symbol table is returned containing the newly defined identifier.

Second, this allows lexical scoping to be implemented fairly naturally. Blocks in PLS are defined to have their own sub-scope (i.e. variables declared in a block are not visible outside that block) and are implemented in the AST as a list of statements. To compile a block, we simply concatenate the bytecodes resulting from the compilation of each statement in the list and return that code along with the original symbol table (i.e. the one without the newly-declared variables).

The code generation for a specific expression or statement is relatively straightforward. In a couple of cases, a given construct is simply syntactic sugar (e.g. the `foreach` loop) or can easily be expressed in terms of other AST types. It is important to note that some of the code generated may not be optimal in terms of efficiency. More work is needed to optimize the generated code in general.

The symbol table also contains a value specifying the current compilation context in order to distinguish slightly different cases. For example, in the case where an assignment to a new variable is being compiled, the `Strg` bytecode is generated when a global statement is being compiled, but the `Strl` bytecode is generated inside a function or kernel declaration.

5.3.3 Code Generation Steps

The `Compiler.compile` function is fairly complicated and is divided in to several general steps.

First, the `Ast.program` is crawled to assign a unique index to each function and kernel declaration in the source program. This is used as a temporary entry point address to be patched later with the actual entry point of each function.

Then, these indices are used to populate the initial state of the symbol table. The symbol table is used to classify identifiers that have been encountered and/or are valid in the current scope in the program as well as store some metadata used throughout the compilation process.

Next, the individual top-level parts of the program are compiled. These consist of the function declarations, kernel declarations, channel declarations, and global statements. Each of these parts has its own helper function to handle specifics called `compile_func`, `compile_kernel`, and inline helper, and `compile_global_stmt`, respectively. For each of the top-level AST parts, the appropriate compilation function is called with the AST subtree and current symbol table. Each function returns a (possibly empty as in the case of channel declarations) list of bytecodes and a possibly updated symbol table to be used in subsequent compilations steps. At this stage, the code for function and kernel declarations is kept separate from that of global-scoped statements. They will later be combined with the global statements appearing at the end of the generated bytecode. The `compile_global_stmt` function handles global variable allocation and kernel instantiation (with a fair amount of error checking.). See the comments in Listing A.5 for more details.

Then, the offsets of each kernel and function declaration for the final output are calculated. Each `Call` and `Par` bytecode's offset in the concatenated declaration and global-scope code is patched with the actual offset based on the unique index from the first step. The maximum number of variables in the global scope is also calculated.

Finally, the generated program is outputted with an unconditional branch to the first global statement and the `Run` bytecode is appended so the program starts the kernel instance processing last.

5.4 Runtime/VM

The VM is implemented in `vm.ml`. The VM module implements a single function, `Vm.run_program`, to execute a compiled PLS program. It has two parameters: the program returned from the `Compiler.compile` function and a boolean that, if true, will cause debugging information to be printed as the program runs.

In the `Vm.run_program` function, a common array of `Bytecode.value`s called `globals` is used to hold global variables. The recursive `exec` function does most of the work of actually executing a program. As parameters it takes an array of `value`s to use as a stack, a frame pointer, a stack pointer, and a program counter. The latter three are integers that each refer to an index in the stack, stack, and program text. The `exec` function processes a single bytecode, manipulates the stack accordingly, and calls itself recursively with updated values of the parameters.

An additional function, `Vm.handle_binop` is used to handle binary operations including arithmetic and comparisons. It performs the appropriate type conversions or raises an exception if unable.

There is a global array of `value`s used as a global stack when executing global statements. Each kernel instance has its own stack to allow them to be executed parallelly; each time a `Par` bytecode is processed, the kernel arguments are copied off of the global stack onto the beginning of a new array of `value`s to be used as the stack for that kernel instance. Each kernel instance has its own `state` type to hold the current values of the frame pointer, stack pointer, etc. for when the kernel's execution is suspended to allow others to be executed.

An array of OCaml `Queue`s is used to represent the channels in a PLS program.

Builtin functions in PLS such as `print` have their own special cases in the `exec` function. In a similar fashion, the `Read` and `Write` bytecodes have special implementations for the `stdin` and `stdout` pipes. Writing a value to `stdout` is the same as calling `print` on the value. Reading from `stdin` uses a special scanner (implemented in `stdin_scanner.ml`) to read the text input from stdin and convert it to the appropriate `value` type.

The kernel instances are executed using a simple round-robin scheduling scheme. Each kernel is executed until it tries to read from a pipe which has no tokens or it reaches the end of its body. At that point, its state is saved and the execution of the next kernel instance resumes. The program terminates once all of the kernel instances are stuck reading from a pipe (and EOF has been read from `stdin` if it's used) or have reached the end of their bodies.

Chapter 6

Test Plan

6.1 Sample Test Programs

Below are two sample test programs along with their generated target program and expected output and input (if any). I have inserted comments (beginning with a semicolon) with the corresponding line numbers from the source. The first column in the generated program is the address of the instruction. The target program was generated by passing the `-b` option to the PLS executable.

6.1.1 fibonacci.pls

Listing 6.1: fibonacci.pls

```
channel loop_channel;

kernel fibonacci(n, seed_1, seed_2, in source, out feedback, out output)
{
    // seed the input with the values
    write(feedback, seed_1);
    write(feedback, seed_2);

    // write the first two values as output
    write(output, seed_1);
    write(output, seed_2);

    for (counter = 2; counter <= n; counter = counter + 1)
    {
        // read the two values
        f_1 = read(source);
        f_2 = read(source);

        // calculate the next number
        fib = f_1 + f_2;

        // write the number to the output
        write(output, fib);

        // write the next two numbers back in to the feedback loop
        write(feedback, f_2);
        write(feedback, fib);
    }

    print("All done!");
```

```
}

// print the first 10 fibonacci numbers (starting with 0 and 1) to stdout
fibonacci(10, 0, 1, loop_channel, loop_channel, stdout);
```

Listing 6.2: fibonacci.pls Bytecode

```
0   : Ba 60
1   : Kent 10 ; 4
2   : Ldl 4   ; 6
3   : Ldl 1
4   : Write
5   : Pop
6   : Ldl 4   ; 7
7   : Ldl 2
8   : Write
9   : Pop
10  : Ldl 5   ; 10
11  : Ldl 1
12  : Write
13  : Pop
14  : Ldl 5   ; 11
15  : Ldl 2
16  : Write
17  : Pop
18  : Push 2  ; 13
19  : Strl 7
20  : Pop
21  : Ldl 7
22  : Ldl 0
23  : Cle
24  : Bz 32
25  : Ldl 3   ; 16
26  : Read
27  : Strl 8
28  : Pop
29  : Ldl 3   ; 17
30  : Read
31  : Strl 9
32  : Pop
33  : Ldl 8   ; 20
34  : Ldl 9
35  : Add
36  : Strl 10
37  : Pop
38  : Ldl 5   ; 23
39  : Ldl 10
40  : Write
41  : Pop
42  : Ldl 4   ; 26
43  : Ldl 9
44  : Write
45  : Pop
46  : Ldl 4   ; 27
47  : Ldl 10
48  : Write
49  : Pop
50  : Ldl 7   ; 13 (for-loop 3rd statement)
51  : Push 1
52  : Add
53  : Strl 7
54  : Pop
55  : Ba -34
56  : Push All done! ; 30
57  : Call -1
58  : Pop
59  : Term    ; 31
```

```
60  : Push 10 ; 34
61  : Push 0
62  : Push 1
63  : Push [Channel 2]
64  : Push [Channel 2]
65  : Push [Channel 1]
66  : Par 1 6
67  : Run
```

Listing 6.3: fibonacci.pls Output

```
0
1
1
2
3
5
8
13
21
34
55
All done!
```

6.1.2 splitter.pls

Listing 6.4: splitter.pls

```
// On an input of the integers 1 to 10, this program will output 1-8. This is
// because each "buffer" kernel does two reads. Since 10%2 == 2, these reads
// will hang forever. Therefore we just terminate the program.

kernel src(in data, out even, out odd)
{
    while(true)
    {
        token = read(data);

        is_even = token % 2 == 0;

        if (is_even)
        {
          write(even, token);
        }
        else
        {
          write(odd, token);
        }
    }
}

kernel buffer(in data_in, out data_out)
{
    while (true)
    {
        first = read(data_in);
        second = read(data_in);

        write(data_out, first);
        write(data_out, second);
    }
}

```

```
kernel sink(in even_data, in odd_data, out output)
{
    while (true)
    {
        write(output, read(odd_data));
        write(output, read(even_data));
    }
}

channel src_to_even, src_to_odd, even_to_sink, odd_to_sink;

src(stdin, src_to_even, src_to_odd);
buffer(src_to_even, even_to_sink);
buffer(src_to_odd, odd_to_sink);
sink(even_to_sink, odd_to_sink, stdout);
```

Listing 6.5: splitter.pls Bytecode

```
0   : Ba 64
1   : Kent 5    ; 6
2   : Push true ; 7
3   : Bz 24
4   : Ldl 0     ; 9
5   : Read
6   : Strl 4
7   : Pop
8   : Ldl 4     ; 11
9   : Push 2
10  : Mod
11  : Push 0
12  : Ceq
13  : Strl 5
14  : Pop
15  : Ldl 5     ; 13
16  : Bz 6
17  : Ldl 1     ; 15
18  : Ldl 4
19  : Write
20  : Pop
21  : Ba 5
22  : Ldl 2     ; 19
23  : Ldl 4
24  : Write
25  : Pop
26  : Ba -24    ; Loop back to line 7
27  : Term      ; 22
28  : Kent 4    ; 25
29  : Push true ; 26
30  : Bz 18
31  : Ldl 0     ; 28
32  : Read
33  : Strl 3
34  : Pop
35  : Ldl 0     ; 29
36  : Read
37  : Strl 4
38  : Pop
39  : Ldl 1     ; 31
40  : Ldl 3
41  : Write
42  : Pop
43  : Ldl 1     ; 32
44  : Ldl 4
45  : Write
46  : Pop
47  : Ba -18    ; Loop back to line 26
48  : Term      ; 34
```

```
49  : Kent 0    ; 37
50  : Push true ; 38
51  : Bz 12
52  : Ldl 2     ; 40
53  : Ldl 1
54  : Read
55  : Write
56  : Pop
57  : Ldl 2     ; 41
58  : Ldl 0
59  : Read
60  : Write
61  : Pop
62  : Ba -12    ; Loop back to line 38
63  : Term      ; 43
64  : Push [Channel 0]  ; 47
65  : Push [Channel 5]
66  : Push [Channel 4]
67  : Par 1 3
68  : Push [Channel 5]  ; 48
69  : Push [Channel 3]
70  : Par 28 2
71  : Push [Channel 4]  ; 49
72  : Push [Channel 2]
73  : Par 28 2
74  : Push [Channel 3]  ; 50
75  : Push [Channel 2]
76  : Push [Channel 1]
77  : Par 49 3
78  : Run
```

Listing 6.6: splitter.pls Input

```
1
2
3
4
5
6
7
8
9
10
```

Listing 6.7: splitter.pls Output

```
1
2
3
4
5
6
7
8
```

6.2 Test Suite Code

The source code, input, and expected output for each test case as well as the test automation script are included in Section A.2.

6.3 Test Case Selection

Each test case was chosen for one of several reasons. The three beginning with the word "basic" were developed early and changed often to quickly check new features in the code. They aren't as much exhaustive as quick sanity checks. Some of the others, such as the "control_structures" and "expr" tests, were chosen to test different features in more depth. Finally others, such as the "fibonacci" and "splitter" tests were meant to be complete (if useless) programs.

6.4 Test Automation

Each test program has an associated ".in" and ".out" file with the same name. The test driver, "run_tests.sh", searches for the three files and if found, executes the test with the ".in" file redirected to `stdin` and compares the output with that in the ".out" file with the `diff` utility. If the output matches the expected output, the test passes. Otherwise the test fails. All of the output is logged. The test driver runs by default whenever the "all" target in the make file is build to ensure the tests stay up to date.

Chapter 7

Lessons Learned

I learned many lessons over the course of this project and will attempt to document some of them in no particular order.

First, there is never enough time. It's very difficult to estimate how much time a particular task will take, and this applies doubly when developing something like a compiler in an unfamiliar language. I left myself a substantial buffer (about two weeks, a third of my planned coding time) for solving any issues that may crop up. There were, of course, plenty and I recommend other groups do the same.

Second, OCaml is your friend. A lot of functionality is baked right in almost everything I wanted to do in my compiler was expressible in what usually amounted to only a few lines of code with the standard library functions. My initial instinct was to translate from something resembling C in my head directly to OCaml but this was usually a mistake. Taking some time to really understand features such as `List.fold_left` helped a lot. As corollary, if you find yourself writing lots of code to express an idea, there's probably an easier way to go about it.

Third, test early and test often. I found it very helpful to right small tests right along side my code, or even to write tests before I had implemented features. This allowed me to get immediate feedback as I developed my compiler. I also found it very helpful to automatically run the entire test suite as part of my build so I could always be sure I hadn't broken anything.

Finally, use your available tools. I tend to be rather lazy as I develop and I want my tools to do as much work for me as possible. I spent some time before I wrote a single line of code for this project to configure my editor to help with OCaml development. For example, I configured my editor to compile as I typed to provide instant feedback with regards to errors and warnings. I also was able to send snippets to an OCaml REPL to test certain lines or functions. Such configuration can greatly aid development.

Appendix A

Source Code

A.1 Compiler

Listing A.1: scanner.mll

```
{ open Parser }

let number = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let integer = number+
let exponent =  'e' ['+' '-']? integer
let fraction = '.' integer

rule token = parse
  [' ' '\t' '\r' '\n']  { token lexbuf }
| "//"                  { comment lexbuf }
| '('                   { LPAREN }
| ')'                   { RPAREN }
| '{'                   { LBRACE }
| '}'                   { RBRACE }
| '['                   { LBRACKET }
| ']'                   { RBRACKET }
| ';'                   { SEMICOLON }
| ','                   { COMMA }
| '='                   { ASSIGN }
| "=="                  { EQ }
| "!="                  { NE }
| '<'                   { LT }
| "<="                  { LTE }
| '>'                   { GT }
| ">="                  { GTE }
| '+'                   { PLUS }
| '-'                   { MINUS }
| '*'                   { TIMES }
| '/'                   { DIVIDE }
| '!'                   { NOT }
| "&&"                  { AND }
| "||"                  { OR }
| '%'                   { MODULO }
| ':'                   { COLON }
| "if"                  { IF }
| "elseif"              { ELSEIF }
| "else"                { ELSE }
| "for"                 { FOR }
| "while"               { WHILE }
```

```
| "foreach"             { FOREACH }
| "return"              { RETURN }
| "function"            { FUNCTION }
| "kernel"              { KERNEL }
| "channel"             { CHANNEL }
| "in"                  { IN }
| "out"                 { OUT }
| "true" as s           { BOOL(Pervasives.bool_of_string s) }
| "false" as s          { BOOL(Pervasives.bool_of_string s) }
| "null"                { NULL }
| (integer? fraction exponent?) | (integer '.'? exponent) | (integer '.')
    as num              { FLOAT(float_of_string num) }
| number+ as s          { INT(int_of_string s) }
| (letter | '_') (letter | number | '_')* as s    { ID(s) }
| '"'                   { STRING(string_literal "" lexbuf) }
| eof                   { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

(* Ignore everything from // to end of line *)
and comment = parse
  ['\r' '\n' ]  { token lexbuf }
| _             { comment lexbuf }

(* Match string literals.
 * TODO: other escape sequences?
 * TODO: this is maybe super-inefficient. *)
and string_literal str = parse
  ([^ '"' '\\']*) as s  { string_literal (str ^ s) lexbuf }
| '\\' '"'              { string_literal (str ^ "\"") lexbuf }
| '"'                   { str }
```

Listing A.2: parser.mly

```
%{ open Ast %}

%token ASSIGN PLUS MINUS TIMES DIVIDE MODULO
%token AND OR NOT
%token EOF
%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token SEMICOLON COMMA COLON
%token EQ NE LT LTE GT GTE
%token IF ELSEIF ELSE FOR WHILE FOREACH RETURN
%token KERNEL FUNCTION
%token IN OUT CHANNEL
%token <int> INT
%token <float> FLOAT
%token <string> STRING
%token <string> ID
%token <bool> BOOL
%token NULL

%right ASSIGN
%left OR
%left AND
%left EQ NE
%left LT GT LTE GTE
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%right NOT
%left LBRACKET

%start program
%type <Ast.program> program

%%

program:
```

```
    /* nothing */           { [] }
  | program func_decl       { FuncDecl($2) :: $1 }
  | program kernel_decl     { KernelDecl($2) :: $1 }
  | program stmt            { Stmt($2) :: $1 }
  | program channel_decl    { Channels($2) :: $1 }

channel_decl:
    CHANNEL id_list SEMICOLON   { $2 }

id_list:
    ID                  { [$1] }
  | id_list COMMA ID    { $3 :: $1 }

func_decl:
    FUNCTION ID LPAREN func_param_list_opt RPAREN block
        { { name = $2; args = $4; body = $6} }

func_param_list_opt:
    /* empty args */    { [] }
  | id_list             { List.rev $1 }

kernel_decl:
    KERNEL ID LPAREN kernel_arg_list RPAREN block
        { { kname = $2; kargs = List.rev $4; kbody = $6 } }

kernel_arg_list:
    kernel_arg                          { [$1] }
  | kernel_arg_list COMMA kernel_arg    { $3 :: $1 }

kernel_arg:
    IN ID   { Input($2) }
  | OUT ID  { Output($2) }
  | ID      { BasicArg($1) }

stmt:
    expr SEMICOLON                              { ExprStmt($1) }
  | RETURN expr SEMICOLON                       { Return($2) }
  | block                                       { Block($1) }
  | IF LPAREN expr RPAREN block else_opt        { If($3, Block($5), $6) }
  | FOR LPAREN expr_opt SEMICOLON expr_opt SEMICOLON expr_opt RPAREN block
        { For(ExprStmt($3), $5, ExprStmt($7), Block($9)) }
  | WHILE LPAREN expr RPAREN block              { While($3, Block($5)) }
  | FOREACH LPAREN ID COLON expr RPAREN block   { ForEach($3, $5, Block($7)) }

else_opt:
    /* nothing */   { Block([]) }
  | ELSE block      { Block($2) }
  | elseif          { $1 }

elseif:
    ELSEIF LPAREN expr RPAREN block else_opt { If($3, Block($5), $6) }

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

block:
    LBRACE stmt_list RBRACE { List.rev $2 }

expr_opt:
    /* nothing */   { EmptyExpr }
  | expr            { $1 }

assignable:
    expr LBRACKET expr RBRACKET       { ListIndex($1, $3) }
  | ID                                { Id($1) }
```

```
primary_expr:
    assignable                         { Assignable($1) }
  | literal                            { Literal($1) }
  | dlist                              { $1 }
  | LPAREN expr RPAREN                 { $2 }
  | ID LPAREN expr_list_opt RPAREN     { Call($1, $3) }

expr:
    primary_expr                       { $1 }
  | ID ASSIGN expr                     { Assign(Id($1), $3) }
  | expr PLUS expr                     { Binop($1, Add, $3) }
  | expr MINUS expr                    { Binop($1, Subtract, $3) }
  | expr TIMES expr                    { Binop($1, Multiply, $3) }
  | expr DIVIDE expr                   { Binop($1, Divide, $3) }
  | expr MODULO expr                   { Binop($1, Modulo, $3) }
  | expr EQ expr                       { Binop($1, Equal, $3) }
  | expr NE expr                       { Binop($1, Neq, $3) }
  | expr LT expr                       { Binop($1, Less, $3) }
  | expr LTE expr                      { Binop($1, Leq, $3) }
  | expr GT expr                       { Binop($1, Greater, $3) }
  | expr GTE expr                      { Binop($1, Geq, $3) }
  | MINUS expr                         { Binop(Literal(Int(0)), Subtract, $2) }
  | NOT expr                           { Binop(Literal(Bool(true)), Neq, $2) }
  | expr AND expr                      { Binop($1, And, $3) }
  | expr OR expr                       { Binop($1, Or, $3) }

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

expr_list_opt:
    /* nothing */   { [] }
  | expr_list       { $1 }

literal:
    INT     { Int($1) }
  | FLOAT   { Float($1) }
  | STRING  { String($1) }
  | BOOL    { Bool($1) }
  | NULL    { Null }

dlist:
    LBRACKET expr_list_opt RBRACKET { DList(List.rev $2) }
```

Listing A.3: ast.ml

```
type op =
    Add
  | Subtract
  | Multiply
  | Divide
  | Modulo
  | Equal
  | Neq
  | Less
  | Leq
  | Greater
  | Geq
  | And
  | Or

type literal =
    Int of int
  | Float of float
  | String of string
  | Bool of bool
  | Null
```

```
type assignable =
    Id of string
  | ListIndex of expr * expr
and expr =
    Literal of literal
  | Assignable of assignable
  | Binop of expr * op * expr
  | Assign of assignable * expr
  | Call of string * expr list
  | DList of expr list
  | EmptyExpr

type stmt =
    ExprStmt of expr
  | Return of expr
  | Block of stmt list
  | If of expr * stmt * stmt
  | For of stmt * expr * stmt * stmt
  | While of expr * stmt
  | ForEach of string * expr * stmt

type func_decl = {
    name: string;
    args: string list;
    body: stmt list;
}

type kernel_arg =
    Input of string
  | Output of string
  | BasicArg of string

type kernel_decl = {
    kname: string;
    kargs: kernel_arg list;
    kbody: stmt list;
}

type program_part =
    FuncDecl of func_decl
  | KernelDecl of kernel_decl
  | Stmt of stmt
  | Channels of string list

type program = program_part list

(* Pretty printing functions *)
let string_of_op = function
    Add         -> "+"
  | Subtract    -> "-"
  | Multiply    -> "*"
  | Divide      -> "/"
  | Modulo      -> "%"
  | Equal       -> "=="
  | Neq         -> "!="
  | Less        -> "<"
  | Leq         -> "<="
  | Greater     -> ">"
  | Geq         -> ">="
  | And         -> "&&"
  | Or          -> "||"

let rec string_of_literal = function
    Int(i)      -> string_of_int i
  | Float(f)    -> string_of_float f
  | String(s)   -> "\"" ^ s ^ "\""
```

```
  | Bool(b)     -> string_of_bool b
  | Null        -> "null"

let rec string_of_expr = function
    Literal(l)          -> string_of_literal l
  | Assignable(a)       -> string_of_assignable a
  | Binop(lhs, op, rhs) ->
      string_of_expr lhs ^ " " ^ string_of_op op ^ " " ^ string_of_expr rhs
  | Assign(lhs, rhs)    ->
      string_of_assignable lhs ^ " = " ^ string_of_expr rhs
  | Call(f, args)       ->
      f ^ "(" ^
      String.concat ", " (List.map string_of_expr (List.rev args)) ^
      ")"
  | DList(exprs)        ->
      "[" ^ String.concat ", " (List.map string_of_expr exprs) ^ "]"
  | EmptyExpr           -> ""
and string_of_assignable = function
    Id(id)              -> id
  | ListIndex(lst, idx) -> string_of_expr lst ^ "[" ^ string_of_expr idx ^ "]"

let rec string_of_stmt = function
    ExprStmt(expr)                  -> string_of_expr expr ^ ";\n"
  | Return(expr)                    -> "return " ^ string_of_expr expr ^ ";\n"
  | Block(stmts)                    ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | If(cond, then_stmt, else_stmt)  ->
      "if(" ^ string_of_expr cond ^ ")" ^ string_of_stmt then_stmt ^
      "else\n" ^ string_of_stmt else_stmt ^ "\n"
  | For(start, term, inc, body)     ->
      "for(" ^ string_of_stmt start ^ "; " ^
      string_of_expr(term) ^ "; " ^
      string_of_stmt(inc) ^ ")\n" ^
      string_of_stmt body ^ "\n"
  | While(cond, body)               ->
        "while(" ^ string_of_expr cond ^ ")\n" ^ string_of_stmt body ^ "\n"
  | ForEach(id, lst, body)          ->
        "foreach(" ^ id ^ " : " ^ string_of_expr lst ^ ")\n" ^
        string_of_stmt body ^ "\n"
and string_of_elseif (expr, stmt) =
    "elseif(" ^ string_of_expr expr ^ ")\n{" ^ string_of_stmt stmt ^ "}\n"

let string_of_func_decl f =
    "function " ^ f.name ^ "(" ^ String.concat ", " f.args ^ ")\n{\n" ^
    String.concat "" (List.map string_of_stmt f.body) ^ "}\n\n"

let string_of_kernel_decl k =
    let string_of_kernel_arg = function
        Input(a)    -> "in " ^ a
      | Output(a)   -> "out " ^ a
      | BasicArg(a) -> a
    in
    "kernel " ^ k.kname ^ "(" ^
    String.concat ", " (List.map string_of_kernel_arg k.kargs) ^ ")\n{\n" ^
    String.concat "" (List.map string_of_stmt k.kbody) ^ "}\n\n"

let string_of_program p =
    let string_of_program_part = function
        FuncDecl(f)     -> string_of_func_decl f
      | KernelDecl(k)   -> string_of_kernel_decl k
      | Stmt(s)         -> string_of_stmt s
      | Channels(c)     -> "channel " ^ (String.concat ", " c) ^ ";\n"
    in
    String.concat "" (List.map string_of_program_part p)
```

Listing A.4: bytecode.ml

```
(* This type represents any value that can be operated on at runtime *)
type value =
    Int of int
  | Float of float
  | String of string
  | Bool of bool
  | VList of value list
  | Channel of int
  | Null

(*
 *  Bytecode | Description                                     | SP   | PC
 * ----------+-------------------------------------------------+------+--------
 *   Halt    | End program                                     | --   | --
 *   Push v  | Push 'v' on to stack                            | +1   | +1
 *   Pop     | Remove value from top of stack                  | +1   | +1
 *   Add     | Pop two values, push sum                        | -1   | +1
 *   Sub     | Pop two values, push difference                 | -1   | +1
 *   Mul     | Pop two values, push product                    | -1   | +1
 *   Div     | Pop two values, push quotient                   | -1   | +1
 *   Mod     | Pop two values, push remainder (sp-2) % (sp-1)  | -1   | +1
 *   Ceq     | Pop two values, push bool (sp-2) == (sp-1)      | -1   | +1
 *   Cne     | Pop two values, push bool (sp-2) != (sp-1)      | -1   | +1
 *   Clt     | Pop two values, push bool (sp-2) < (sp-1)       | -1   | +1
 *   Cle     | Pop two values, push bool (sp-2) <= (sp-1)      | -1   | +1
 *   Cgt     | Pop two values, push bool (sp-2) > (sp-1)       | -1   | +1
 *   Cge     | Pop two values, push bool (sp-2) >= (sp-1)      | -1   | +1
 *   And     | Pop two values, push (sp-2) && (sp-1)           | -1   | +1
 *   Or      | Pop two values, push (sp-2) || (sp-1)           | -1   | +1
 *   Strg n  | Store top of stack to global index              | --   | +1
 *   Ldg n   | Load global from arg index and push on stack    | +1   | +1
 *   Strl n  | Store top of stack to local index relative FP   | --   | +1
 *   Ldl n   | Load relative FP and push on stack              | +1   | +1
 *   Call n  | Call function at absolute offset 'n'          $ | +1   | =n
 *   Ret n   | Return from function of arity 'n'               |FP-n  | *
 *   Bz n    | Branch relative if top of stack is 0            | -1   | +1 or n
 *   Bnz n   | Branch relative if top of stack is not 0        | -1   | +1 or n
 *   Ba n    | Unconditional branch relative                   | --   | +n
 *   Mlst n  | Pop 'n' values, construct list, push list       | +1-n | +1
 *   Ilst    | Pop list, pop index, push value from list index | -1   | +1
 *   Alst    | Pop list at (sp-3), set idx (sp-2) to (sp-1)    | -2   | +1
 *   Read    | Pop channel index, push value from chan head #  | +1   | +1
 *   Write   | Pop val, pop chan index, write val to chan      | -1   | +1
 *   Ent n   | Push current FP, advance SP by 'n'              | +n+1 | +1
 *   Par n m | Create kernel instance at PC 'n' with arity 'm' | -n   | +1
 *   Kent n  | Advance SP by 'n'                               | +n   | +1
 *   Term    | Terminate kernel instance                       | --   | --
 *   Run     | Start kernel instances                          | --   | --
 *
 *  * = Retrieve old values from stack
 *  # = Will yield execution if channel is empty
 *  $ = Builtin functions are "inlined" and just consume arguments and push
 *      results without branching.
 *)

type bcode =
    Halt (* End program *)
  | Push of value (* Push a literal on the stack *)
  | Pop (* Pop a value off the stack *)
  | Add (* add *)
  | Sub (* subtract *)
  | Mul (* multiply *)
  | Div (* divide *)
  | Mod (* modulus *)
  | Ceq (* compare equal *)
```

```
  | Cne (* compare not equal *)
  | Clt (* compare less than *)
  | Cle (* compare less than or equal to *)
  | Cgt (* compare greater than or *)
  | Cge (* compare greater than or equal to *)
  | And (* logical and *)
  | Or (* logical or *)
  | Strg of int (* Store global var *)
  | Ldg of int (* Load var *)
  | Strl of int (* Store local var *)
  | Ldl of int (* Load local var *)
  | Call of int (* call func with specified index *)
  | Ret of int (* Return with number of args to the function *)
  | Bz of int (* Branch relative if top if stack zero *)
  | Bnz of int (* Branch relative if top of stack is not zero *)
  | Ba of int (* Branch relative always *)
  | Mlst of int (* Make a list with the top n values on the stack *)
  | Ilst (* Index a list *)
  | Alst (* Assign a value to a list *)
  | Read (* Push a value from the channel on the top of the stack *)
  | Write (* Write a value to a channel *)
  | Ent of int (* allocate room for locals, shift sp and fp for call *)
  | Par of int * int (* Spawn of kernel instance with num locals *)
  | Kent of int (* Allocate space for kernel locals *)
  | Term (* Terminate a kernel instance *)
  | Run (* Start the parallel processing *)

type program = {
    num_globals          : int;
    text                 : bcode array;
    num_channels         : int;
    num_kernel_instances : int;
}

(* Convert an Ast literal to a value for the stack *)
let rec value_of_literal = function
    Ast.Int(i)      -> Int(i)
  | Ast.Float(f)    -> Float(f)
  | Ast.String(s)   -> String(s)
  | Ast.Bool(b)     -> Bool(b)
  | Ast.Null        -> Null

let bool_of_value = function
    Int(i)      -> i != 0
  | Float(f)    -> f <> 0.0
  | String(_)   -> true
  | Bool(b)     -> b
  | VList(_)    -> true
  | Channel(_)  -> true
  | Null        -> false

(* Pretty Printing Functions *)
let rec string_of_value = function
    Int(i)      -> string_of_int i
  | Float(f)    -> string_of_float f
  | String(s)   -> s
  | Bool(b)     -> string_of_bool b
  | VList(l)    -> "[" ^ String.concat ", " (List.map string_of_value l) ^ "]"
  | Channel(c)  -> "[Channel " ^ string_of_int c ^ "]"
  | Null        -> "Null"

let string_of_bcode = function
    Halt        -> "Halt"
  | Push(value) -> "Push " ^ string_of_value value
  | Pop         -> "Pop"
  | Add         -> "Add"
  | Sub         -> "Sub"
```

```
  | Mul         -> "Mul"
  | Div         -> "Div"
  | Mod         -> "Mod"
  | Ceq         -> "Ceq"
  | Cne         -> "Cne"
  | Clt         -> "Clt"
  | Cle         -> "Cle"
  | Cgt         -> "Cgt"
  | Cge         -> "Cge"
  | And         -> "And"
  | Or          -> "Or"
  | Strg(idx)   -> "Strg " ^ string_of_int idx
  | Ldg(idx)    -> "Ldg " ^ string_of_int idx
  | Strl(idx)   -> "Strl " ^ string_of_int idx
  | Ldl(idx)    -> "Ldl " ^ string_of_int idx
  | Call(idx)   -> "Call " ^ string_of_int idx
  | Ret(nargs)  -> "Ret " ^ string_of_int nargs
  | Bz(idx)     -> "Bz " ^ string_of_int idx
  | Bnz(idx)    -> "Bnz " ^ string_of_int idx
  | Ba(idx)     -> "Ba " ^ string_of_int idx
  | Mlst(n)     -> "Mlst " ^ string_of_int n
  | Ilst        -> "Ilst"
  | Alst        -> "Alst"
  | Read        -> "Read"
  | Write       -> "Write"
  | Ent(num)    -> "Ent " ^ string_of_int num
  | Par(idx, nargs) -> "Par " ^ string_of_int idx ^ " " ^ string_of_int nargs
  | Kent(n)     -> "Kent " ^ string_of_int n
  | Term        -> "Term"
  | Run         -> "Run"

let string_of_program prog =
    "num_globals: " ^ string_of_int prog.num_globals ^ "\n" ^
    "num_channels: " ^ string_of_int prog.num_channels ^ "\n" ^
    "num_kernel_instances: " ^ string_of_int prog.num_kernel_instances ^ "\n" ^
    (String.concat "\n"
        (Array.to_list (Array.mapi
            (fun idx bcode ->
                Printf.sprintf "%-4d: %s" idx (string_of_bcode bcode))
            prog.text)))
```

Listing A.5: compiler.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

type context =
    Global
  | Function
  | Kernel

type channel_type =
    In
  | Out

type kernel_instance_arg =
    ChannelArg of int
  | ExprArg of expr

type kernel_instance = {
    index           : int;
    instance_args   : kernel_instance_arg list;
}

(* Symbol table *)
```

```
type symbol_table = {
    (* Map unique index and number of arguments for each functions *)
    function_index   : (int * int) StringMap.t;
    (* Map unique index and list of arguments for each kernel decl *)
    kernel_index     : (int * kernel_arg list) StringMap.t;
    (* Map unique index to each global variable *)
    global_index     : int StringMap.t;
    (* Map unique index to each local variable (kernel or function) *)
    local_index      : int StringMap.t;
    (* When compiling a kernel, map each channel name to index in stack and
     * direction (in or out) *)
    channel_index    : (int * channel_type) StringMap.t;
    (* Helper to keep track of the maximum number of local variables declared
     * in a function or kernel decl for 'Ent' and 'Kent' opcodes *)
    max_locals       : int ref;
    (* Offset local variable indices by this amount *)
    local_offset     : int;
    (* Enum specifying current compilation context (global, kernel, function *)
    context          : context;
    (* Unique index for all defined channels in the program *)
    channels         : int StringMap.t;
    (* Helper map for verifying channel connections - (bool input * bool
     * output) where value is true if that end of the channel is connected *)
    channel_usage    : (bool * bool) StringMap.t;
    (* Number of kernel instances in the program *)
    kernel_instances : int;
}

(* Return a list of pairs, int * 'a, starting at 'n' and increasing by 'stride'
 * for each element of the list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n + stride) tl

let string_map_pairs map pairs =
    List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

let string_map_append map item =
    if StringMap.mem item map then
        (raise (Failure ("Item " ^ item ^ " already in map")))
    else
        StringMap.add item (StringMap.cardinal map) map

(* Compile the program 'prog' in Ast form to bytecode *)
let compile prog =
    (* These functions are always visible *)
    let builtin_funcs =
        string_map_pairs StringMap.empty [
            ((-1, 1), "print");
            ((-4, 2), "append");
            ((-5, 1), "length");
    ]
    in
    (* These functions are only visible when compiling a kernel body *)
    let kernel_funcs =
        string_map_pairs StringMap.empty [((-2, 1), "read"); ((-3, 2), "write")]
    in
    (* These channels are always available *)
    let builtin_channels =
        string_map_pairs StringMap.empty [(0, "stdin"); (1, "stdout")]
    in
    (* Builtin channels already have one end connected *)
    let builtin_channel_usage =
        string_map_pairs StringMap.empty [
            ((true, false), "stdin");
            ((false, true), "stdout")
        ]
```

```
in

(* Compile an AST expression. Returns a pair, (bytecode list,
 * symbol_table), since an expression (assignment) could change the
 * symbol table. *)
let rec compile_expr env = function
    (* Literals are easy, we just push a value on the stack *)
    Literal(l) -> ([Push(value_of_literal l)], env)
    (* Binops compile the left hand side, then the right hand side, and
     * finally output the appropriate opcode *)
  | Binop(lhs, op, rhs) ->
        (* TODO: do we need the symbol_table here in case lhs/rhs is an
         * assignment? *)
        let (lhs_code, _) = compile_expr env lhs
        and (rhs_code, _) = compile_expr env rhs
        in
        (lhs_code @ rhs_code @
         [match op with
            Add -> Add
          | Subtract -> Sub
          | Multiply -> Mul
          | Divide -> Div
          | Modulo -> Mod
          | Equal -> Ceq
          | Neq -> Cne
          | Less -> Clt
          | Leq -> Cle
          | Greater -> Cgt
          | Geq -> Cge
          | And -> And
          | Or -> Or], env)
  | Assignable(a) -> compile_assignable env a
    (* Assignments are tricky. First, like the Ids, we need to determine if
     * the variable being assigned to is a local, a global, or new. In the
     * case of the first two, just store a value at that index. Otherwise,
     * it's a new declaration. If we are in a kernel or function context,
     * it's a local variable and should be added to the local map and the
     * symbol table's max_locals might need to be incremented. In a global
     * context, we need to allocate a new global variable. *)
  | Assign(lhs, rhs) -> compile_assignment env lhs rhs
    (* For function calls, a couple of things could happen. If we are
     * actually calling a function, we check that the function is in the
     * function_index of the symbol_table, check that the number of args
     * matches, compile the code for the args in declaration order, and
     * output the 'Call' bytecode to jump to the beginning of the function
     * body. Since we are still compiling the program at this point, we
     * don't actually know the absolute address of the function, so the
     * 'Call' bytecode gets the index of the function in the symbol table
     * to be patched up later.
     *
     * Kernel functions (read, write) get handled a bit differently in the
     * 'compile_kernel_io_func' function since they have some slight
     * differences. *)
  | Call(f, args) ->
        (* Look up the function in the symbol table *)
        let (func_index, nargs) =
            (try (StringMap.find f env.function_index)
            with Not_found ->
                raise (Failure ("Undefined function '" ^ f ^ "'")))
        in
        (* Verify the number of args matches. *)
        if nargs = List.length args then
            (* Kernel read/write calls get handled differently *)
            if env.context = Kernel && (StringMap.mem f kernel_funcs) then
                (compile_kernel_io_func env f (List.rev args), env)
            else
                (* Compile the arguments in order. *)
```

```
                (* TODO: preserve env? Allow assignments in calls? *)
                ((List.concat
                    (List.map fst
                        (List.map (compile_expr env) args))) @
                [Call(func_index)], env)
        else
            raise (Failure (Printf.sprintf
            "Arity mismatch when calling '%s': got %d args, expected %d"
            f (List.length args) nargs))
  | EmptyExpr -> ([], env) (* Do nothing *)
  | DList(exprs) ->
        let exprs_code = List.map (fun e -> fst (compile_expr env e)) exprs
        in
        ((List.concat exprs_code) @ [Mlst(List.length exprs)], env)
and compile_assignable env = function
    (* Names get looked up in the local variables, then failing that, the
     * globals. Once found, we load the value corresponding index from the
     * frame pointer on to the top of the stack *)
    Id(id) ->
        let load = (try [Ldl (StringMap.find id env.local_index)]
        with Not_found -> try [Ldg (StringMap.find id env.global_index)]
        with Not_found -> raise (Failure ("unknown variable '" ^ id ^ "'")))
        in
        (load, env)
  | ListIndex(list_expr, idx_expr) ->
        let (list_expr_code, env) = compile_expr env list_expr in
        let (idx_code, env) = compile_expr env idx_expr in
        (list_expr_code @ idx_code @ [Ilst], env)
and compile_assignment env lhs rhs =
    let (rhs_code, env) = compile_expr env rhs in
    match lhs with
    Id(id) ->
        let (store_code, env) =
            let location = (StringMap.mem id env.local_index,
                            StringMap.mem id env.global_index) in
            match location with
                (true, _) -> (* existing local - get index *)
                    ([Strl(StringMap.find id env.local_index)], env)
              | (false, true) -> (* global - get global index *)
                    ([Strg(StringMap.find id env.global_index)], env)
              | (false, false) -> (* new var, update env *)
                    let (map, offset) = match env.context with
                        Global -> (env.global_index, 1) (* 0-index *)
                      | _      -> (env.local_index, env.local_offset)
                    in
                    (* The index of the new variable gets the next highest
                     * integer. Since variable allocation is FIFO, we don't
                     * need to worry about holes in the allocation. *)
                    let new_idx = ((StringMap.cardinal map) + 1 - offset)
                    in
                    env.max_locals := max !(env.max_locals) new_idx;
                    (* Save the new index back to the appropriate slot in
                     * the symbol table. *)
                    let new_map = StringMap.add id new_idx map in
                    match env.context with
                        Global -> ([Strg(new_idx)],
                                   { env with global_index = new_map })
                      | _      -> ([Strl(new_idx)],
                                   { env with local_index = new_map })
        in
        (* Put the RHS on the top of the stack, and store that value. *)
        (rhs_code @ store_code, env)
  | ListIndex(lst, idx) ->
          let (lst_code, _) = compile_expr env lst
          and (idx_code, env) = compile_expr env idx
          in
          ([Halt] @ lst_code @ [Halt] @ idx_code @ rhs_code @ [Alst], env)
```

```
(* Kernel IO functions are a lot like regular functions except that we
 * check to make sure that they are called on the correct type of channel.
 * E.g. it is an error to call "read" on a channel declared as "out" *)
and compile_kernel_io_func env name args =
    (* This function verifies that the first argument to the function (the
     * channel) is just an identifier matching a channel and not part of an
     * expression. It returns (int * channel_type) *)
    let find_channel env args =
        match (List.hd args) with
            Assignable(Id(id)) ->
                (try (StringMap.find id env.channel_index)
                with Not_found ->
                    raise (Failure ("Unknown channel '" ^ id ^ "'")))
          | _ -> raise (Failure ("Cannot use expression as channel name"))
    in
    match name with
        "read" -> let (idx, channel_type) = find_channel env args
            in
            if channel_type = In then
                [Ldl(idx); Read]
            else
                raise (Failure ("read() requires an input channel"))
      | "write" -> let (idx, channel_type) = find_channel env args
            in
            if channel_type = Out then
                [Ldl(idx)] @
                fst (compile_expr env (List.nth args 1)) @
                [Write]
            else
                raise (Failure ("write() requires an output channel"))
      | _ -> raise (Failure ("Unknown error compiling kernel IO"))
in

(* This function compiles a statement recursively. It returns a pair of
 * (bytecode list, symbol_table) since the symbol table might change. *)
let rec compile_stmt env = function
    (* Expressions use the output of the 'compile_expr' function above. We
     * also pop the value off the top of the stack since every expression
     * pushes one unused value. This is to support chaining of calls
     * without much hassle, e.g. 'f(x);' needs to have it's return value
     * popped since it's never used. *)
    ExprStmt(expr) ->
        let (expr_code, env) = compile_expr env expr in
          (expr_code @ [Pop], env)
    (* Returns are only valid in a function context. The 'Ret' value is set
     * to 0 to be set to the number of function arguments later (so we know
     * where to place the return value) *)
  | Return(expr) ->
        if env.context <> Function
        then raise (Failure "Can only return from a function")
        else let (expr_code, env) = compile_expr env expr in
            (expr_code @ [Ret(0)], env)
    (* A block is just a sequence of statements so compile them in order *)
  | Block(stmts) ->
        (fst (List.fold_left stmt_sequence_helper ([], env) stmts), env)
  | If(test, body, else_stmt) ->
        let (test_code, _) = compile_expr env test
        and (body_code, _) = compile_stmt env body
        and (else_body_code, _) = compile_stmt env else_stmt
        in
        (test_code @
        (* Branch to else if cond is false *)
        [Bz(List.length body_code + 2)] @
        body_code @
        (* Branch out of if statement *)
        [Ba(1 + List.length else_body_code)] @
        (* else *)
```

```
        else_body_code,
        env)
    (* TODO: rewrite this as while loop in Block()? *)
  | For(init, test, inc, body) ->
        let (init_code, for_env) = compile_stmt env init in
        let (test_code, _) = compile_expr for_env test
        and (inc_code, _) = compile_stmt for_env inc
        and (body_code, _) = compile_stmt for_env body in
        (init_code @
        test_code @
        (* if the test returns false, skip out of the loop *)
        [Bz(2 + List.length body_code + List.length inc_code)] @
        body_code @
        inc_code @
        [Ba(-
            (List.length inc_code + List.length body_code +
            List.length test_code + 1))], env)
  | While(test, body) ->
        let (test_code, _) = compile_expr env test
        and (body_code, _) = compile_stmt env body
        in
        (test_code @
        (* Skip the body if the test is false *)
        [Bz(2 + List.length body_code)] @
        body_code @
        (* Branch back to the test *)
        [Ba(- (List.length body_code + List.length test_code + 1))], env)
    (* A foreach loop can be compiled as a for loop with some
     * hack^Wtrickiness: foreach(id : lst) { body } is equivalent to
     * for(i = 0; i < length(lst); i = i + 1) { id = lst[i]; body }
     * TODO: Currently, the list index value name is hardcoded so we cannot
     * have nested foreach loops without "fun" occurring. *)
  | ForEach(id, lst, body) ->
        let idx = Id("__i") in (* TODO: nested loops needs unique name *)
        compile_stmt env (For(
            ExprStmt(Assign(idx, Literal(Int(0)))),
            Binop(Assignable(idx), Less, Call("length", [lst])),
            ExprStmt(Assign(idx,
                            Binop(Assignable(idx), Add, Literal(Int(1))))),
            Block([
                ExprStmt(Assign(Id(id), Assignable(ListIndex(lst, Assignable(idx)))));
                body])))
(* Small helper function for fold_left when compiling a sequence of
 * statements. Passes the resulting symbol table of each call to
 * compile_stmt to the next and accumulates the bytecodes of each. *)
and stmt_sequence_helper res stmt =
    let (code, env) = compile_stmt (snd res) stmt in
    (fst res @ code, env)
in

(* Compile the body of a function or kernel decl. Return
 * (bcode list * symbol_table) *)
let compile_body env body =
    List.fold_left stmt_sequence_helper ([], env) body
in

(* Compile a function declaration. *)
let compile_func env f =
    let num_args = List.length f.args in
    (* Construct the local symbol table. Arguments are numbered starting at
     * -2 and decreasing by one in declaration order. We offset the first
     * local variable by the number of arguments. *)
    let func_env = { env with
        local_index =
            string_map_pairs StringMap.empty (enum (-1) (-2) f.args);
        local_offset = num_args;
        context = Function }
```

```
    in
    (* Compile the body and get the updated symbol table *)
    let (code, env) = compile_body func_env f.body in
    (* Patch the 'Ret' opcodes with the number of arguments. *)
    let fixed_code = List.map
        (function
            Ret(n) when n = 0 -> Ret(num_args)
          | _ as instr -> instr)
        code
    in
    (* Functions start with an 'Ent' to make room for the locals, and
     * always end in a 'return null;', even if it's unreachable. *)
    [Ent !(env.max_locals)] @ fixed_code @ [Push(Null); Ret(num_args)]
in

(* Compile a kernel declaration. *)
let compile_kernel env k =
    (* Kernels behave a bit differently than functions - since each kernel
     * instance has its own stack, we start the arguments at 0 and go _up_
     * by one, in order. *)
    let arg_indices = enum 1 0 k.kargs in
    (* We need to separate the arguments a bit for some later verification.
     * This preserves their indices, though. *)
    let separate_args (args, channels) = function
        (idx, Input(id)) -> (args, (idx, In, id) :: channels)
      | (idx, Output(id)) -> (args, (idx, Out, id) :: channels)
      | (idx, BasicArg(id)) -> ((idx, id) :: args, channels)
    in
    let (args, channels) = List.fold_left separate_args ([], []) arg_indices
    in
    (* Set up the symbol table for the kernel body. *)
    let kernel_env = { env with
        (* Fold in the kernel IO functions *)
        function_index =
            StringMap.fold StringMap.add env.function_index kernel_funcs;
        (* Initial local variables are the non-channel arguments *)
        local_index =
            string_map_pairs StringMap.empty args;
        (* Register the channels, their stack locations, and their
         * directions *)
        channel_index =
            string_map_pairs
                StringMap.empty
                (List.map
                    (fun (idx, ctype, name) -> ((idx, ctype), name))
                    channels);
        (* Channels don't go in the local's map, so we need to offset all
         * local variables by the number of channels to make sure they
         * don't get overwritten *)
        local_offset = -(List.length channels);
        context = Kernel;
        max_locals = ref 0; }
    in
    let (code, env) = compile_body kernel_env k.kbody
    in
    (* Kernel bodies start with a 'Kent' to allocate room for the local
     * variables. Can't use an 'Ent' here because we don't need to store
     * the current fp, etc. *)
    [Kent(!(env.max_locals))] @ code @ [Term]
in

(* Compile a single statement in the global context. This includes kernel
 * instance creation. *)
let compile_global_stmt env stmt =
    (* This function does all the work for instantiating a kernel. *)
    let instantiate_kernel env id args =
        (* Verify a single kernel arg - channel arguments get names of
```

```
 * channels, non-channel arguments are normal expressions. *)
let construct_arg arg arg_def = match (arg, arg_def) with
    (* Channel arg - make sure the channel exists *)
    (Assignable(Id(name)), Input(_))
  | (Assignable(Id(name)), Output(_)) ->
        let channel = (try (StringMap.find name env.channels)
            with Not_found -> raise (Failure
                ("Unknown channel \"" ^ name ^ "\"")))
        in
        ChannelArg(channel)
    (* Channel argument, but got expr *)
  | (_, Input(_ as def)) | (_, Output(_ as def)) -> raise
        (Failure (Printf.sprintf
            "Cannot use expression \"%s\" as channel argument %s"
            (string_of_expr arg) def))
    (* Non-channel argument, just pass the expression. *)
  | (_ as expr, BasicArg(_)) -> ExprArg(expr)
in

(* This function makes sure that we haven't already connected this
 * end of the channel. If we haven't, store that it is now
 * connected. *)
let verify_channel_arg env arg arg_def = match (arg, arg_def) with
    (Assignable(Id(name)), Input(_))
  | (Assignable(Id(name)), Output(_)) ->
        (let usage = StringMap.find name env.channel_usage in
        match (usage, arg_def) with
            ((_, false), Input(_)) ->
                { env with channel_usage =
                    StringMap.add name ((fst usage), true)
                    env.channel_usage }
          | ((false, _), Output(_)) ->
                { env with channel_usage =
                    StringMap.add name (true, (snd usage))
                    env.channel_usage }
          | _ -> raise (Failure
                ("Channel " ^ name ^ " is already bound")))
  | _ -> env
in

(* Make sure that the kernel we are trying to instantiate has
 * already been compiled. If so, get its index and arguments from
 * the symbol table. *)
let (index, arg_defs) = (try (StringMap.find id env.kernel_index)
    with Not_found -> raise (Failure
        ("Unknown kernel \"" ^ id ^ "\"")))
in

(* First argument pass *)
let kernel_args = (try (List.map2 construct_arg args arg_defs)
    with Invalid_argument(_) -> raise
        (Failure "Kernel arity mismatch")) (* TODO: error msg *)
in

(* Verify the channel connections *)
let env = List.fold_left2 verify_channel_arg env args arg_defs
in

(* Generate code for the arguments *)
let gen_arg_code = function
    ChannelArg(chan) -> [Push(Channel(chan))]
  | ExprArg(expr)    -> fst (compile_expr env expr)
in

let arg_code = List.concat (List.map gen_arg_code kernel_args)
in
```

```
        (* 'Par' to spawn the kernel instance, preceded by arguments *)
        (arg_code @ [Par(index, List.length kernel_args)],
        { env with kernel_instances = env.kernel_instances + 1 })
    in

    (* Copy the environment, but set the context to global so all variables
     * are global variables, etc. *)
    let global_env = { env with context = Global; }
    in

    (* Check if this is a kernel instantiation or a normal function call *)
    match stmt with
        (* If this is a 'Call' AST node, and the id of the call is a
         * function, we output a normal function call. Otherwise, if the id
         * is a kernel, try to instantiate that kernel.
         * NOTE: the consequence is that if we have a function and a kernel
         * with the same name, you can never instantiate that kernal!
         * TODO: above should never happen *)
        ExprStmt(Call(id, args)) -> (try (compile_stmt global_env stmt)
            with _ -> instantiate_kernel env id (List.rev args))
        (* It's some other statement - just compile it as usual. *)
      | _                        -> compile_stmt global_env stmt
in

(* This bit of code assigns a unique integer id to each kernel and function
 * decl, starting at 0 and increasing by one. This will be used later to
 * calculate entry points *)
let decl_indices =
    enum 1 0 (List.filter (fun (s, _, _) -> (String.length s) > 0)
        (List.map (fun part -> match part with
            FuncDecl(f) -> (f.name, List.length f.args, [])
          | KernelDecl(k) -> (k.kname, List.length k.kargs, k.kargs)
          | _ -> ("", 0, [])) prog))
in

(* Get the function indices by checking that the number of kernel args from
 * 'decl_indices' is 0 *)
let function_indices =
    StringMap.fold StringMap.add builtin_funcs
    (string_map_pairs StringMap.empty
        (List.map (fun (idx, (name, nargs, _)) -> ((idx, nargs), name))
            (List.filter (fun (_, (_, _, kargs)) -> List.length kargs = 0)
                decl_indices)))
in

(* Get the kernel indices by checking that the number of kernel args from
 * 'decl_indices' is greater than 0 *)
let kernel_indices =
    string_map_pairs StringMap.empty
        (List.map (fun (idx, (name, _, kargs)) -> ((idx, kargs), name))
            (List.filter (fun (_, (_, _, kargs)) -> List.length kargs > 0)
        decl_indices))
in

(* Our initial symbol table *)
let env = { function_index   = function_indices;
            kernel_index     = kernel_indices;
            global_index     = StringMap.empty;
            local_index      = StringMap.empty;
            channel_index    = StringMap.empty;
            max_locals       = ref 0;
            local_offset     = 0;
            context          = Global;
            channels         = builtin_channels;
            channel_usage    = builtin_channel_usage;
            kernel_instances = 0 }
in
```

```
(*StringMap.iter (fun name (idx, nargs) ->*)
(*    Printf.printf "%s -> (idx: %d, nargs: %d)\n" name idx nargs) env.function_index;*)

(* Compile each segment of the program in order. We store kernel and
 * function decl code seperately from global-scope code. *)
let compile_part ((decls, globals), env) = function
    FuncDecl(f)     -> (((compile_func env f) :: decls, globals), env)
  | KernelDecl(k)   -> (((compile_kernel env k) :: decls, globals), env)
  | Stmt(s)         -> let (body, new_env) = compile_global_stmt env s in
        ((decls, body :: globals), new_env)
  | Channels(c)     ->
        let new_channels =
            List.fold_left string_map_append env.channels c
        and new_channel_usage =
            List.fold_left
                (fun map name -> StringMap.add name (false, false) map)
                env.channel_usage c
        in
        ((decls, globals), { env with
            channels = new_channels;
            channel_usage = new_channel_usage
        })
in

(* Actually compile the program *)
let (bodies, env) = List.fold_left compile_part (([], []), env) prog
in

let decl_bodies = List.rev (fst bodies)
in

let global_stmts = List.rev (snd bodies)
in

(* Calculate the actual address of each decl *)
let decl_offsets = Array.of_list (List.rev (fst (List.fold_left
    (fun (offsets, idx) body ->
        (idx :: offsets, (idx + List.length body)))
    ([], 1) decl_bodies)))
in

(*Array.iter (fun o -> Printf.printf "%d\n" o) decl_offsets;*)

(* Get the total length of function and kernel code so we know where to
 * branch when the program starts. *)
let decl_bodies_length = List.length (List.concat decl_bodies)
in

(* Generate the program text segment.
 * 1) Go back and patch 'Call' and 'Par' bcodes with the actual addresses of
 * the kernels or functions.
 * 2) Concatenate the global code after all of the decls *)
let text = List.map
    (function
        Call(idx) when idx >= 0 -> Call(decl_offsets.(idx))
      | Par(idx, nargs) when idx >= 0 -> Par(decl_offsets.(idx), nargs)
      | _ as instr -> instr)
    (List.concat (decl_bodies @ global_stmts))
in

let num_globals = List.fold_left
    (fun num bcode -> match bcode with
        Strg(i) -> max num i
      | _ -> num)
    0 text
in
```

```
    (* Output the final program - the text is starts with a branch to the
     * beginning of the global code. The last bytecode after the global
     * statements is the special 'Run' to start the parallel processing. *)
    {
        text = Array.of_list ([Ba(decl_bodies_length + 1)] @ text @ [Run]);
        num_globals = num_globals + 1; (* 0-indexed! *)
        num_channels = StringMap.cardinal env.channels;
        num_kernel_instances = env.kernel_instances;
    }
```

Listing A.6: stdin_scanner.mll

```
{
open Bytecode

type token = EOF | Value of value
}

let number = ['0'-'9']
let integer = number+
let exponent =  'e' ['+' '-']? integer
let fraction = '.' integer
let whitespace = [' ' '\n' '\t' '\r']

rule token = parse
    eof { EOF }
  | (integer? fraction exponent?) | (integer '.'? exponent) | (integer '.')
        as num              { Value(Float(float_of_string num)) }
  | integer as num          { Value(Int(int_of_string num)) }
  | "true"                  { Value(Bool(true)) }
  | "false"                 { Value(Bool(false)) }
  | [^ ' ' '\n' '\t' '\r']+ as str  { Value(String(str)) }
  | whitespace              { token lexbuf }
```

Listing A.7: vm.ml

```
open Bytecode

type state = {
    stack: value array;
    fp       : int;
    sp       : int;
    pc       : int;
    finished : bool;
}

let stack_size = 1024

let run_program prog debug =
    let globals = Array.make prog.num_globals Null
    and text = prog.text
    and channels = Array.init prog.num_channels (fun _ -> Queue.create ())
    and kernel_states =
        Array.init prog.num_kernel_instances
            (fun _ -> {
                stack = Array.make stack_size Null;
                fp = 0; sp = 0; pc = 0; finished = false;
            })
    and next_state = ref 0
    and global_stack = Array.make stack_size Null
    and got_eof = ref false
    and stdin_lexbuf = Lexing.from_channel stdin
    in

    let read_token_from_stdin () =
```

```
    (*print_string "input> "; flush stdout;*)
    match Stdin_scanner.token stdin_lexbuf with
        Stdin_scanner.EOF -> raise End_of_file
      | Stdin_scanner.Value(_ as v) -> v
in

let ptr_of_value = function
    Int(i) -> i
  | _      -> raise (Failure ("Bad stack pointer"))
in

let dump_stack stack fp sp =
    print_endline "STACK";
    Array.iteri (fun i v -> if (i < sp) && (i >= fp) then
        Printf.printf "%3d: %s\n" i (string_of_value v)) stack;
    print_endline "END STACK"
in

let list_assign lst idx value =
    if idx >= List.length lst
    then raise
        (Failure (Printf.sprintf "List index %d is out of range" idx))
    else
        List.mapi (fun i v -> if i == idx then value else v) lst
in

let read_channel stack sp = match stack.(sp - 1) with
    Channel(0) -> (try (
        let token = read_token_from_stdin () in
        (*print_endline ("GOT TOKEN: \"" ^ string_of_value token ^ "\"");*)
        stack.(sp - 1) <- token; true)
        with End_of_file -> got_eof := true; false)
  | Channel(1) -> raise (Failure "Cannot read from stdout")
  | Channel(i) -> let channel = channels.(i) in
        (if Queue.is_empty channel then
            false
        else
            (stack.(sp - 1) <- Queue.pop channel; true))
  | _ as v -> raise (Failure (Printf.sprintf
        "Cannot read from non-channel \"%s\""
        (string_of_value v)))
in

let write_channel stack sp = match stack.(sp - 2) with
    Channel(0) -> raise (Failure "Cannot write to stdin")
  | Channel(1) -> print_endline (string_of_value stack.(sp - 1))
  | Channel(i) -> Queue.push stack.(sp - 1) channels.(i)
  | _          -> raise (Failure "Cannot write to non-channel")
in

(* Normalize types for arithmetic *)
let normalize_arith_types lhs rhs = match (lhs, rhs) with
    (Int(_), Int(_))        -> (lhs, rhs)
  | (Int(l), Float(_))      -> (Float(float_of_int l), rhs)
  | (Int(_), Bool(r))       -> (lhs, Int(if r then 1 else 0))
  | (Float(_), Float(_))    -> (lhs, rhs)
  | (Float(_), Int(r))      -> (lhs, Float(float_of_int r))
  | (Float(_), Bool(r))     -> (lhs, Float(if r then 1.0 else 0.0))
  | (Bool(l), Int(_))       -> (Int(if l then 1 else 0), rhs)
  | (Bool(l), Float(_))     -> (Float(if l then 1.0 else 0.0), rhs)
  | (Bool(_), Bool(_))      -> (lhs, rhs)
  | (String(_), _)          -> (lhs, String(string_of_value rhs))
  | (_, String(_))          -> (String(string_of_value lhs), rhs)
  | _                       -> raise (Failure
        (Printf.sprintf "Unknown conversion: lhs = %s, rhs = %s\n"
                        (string_of_value lhs) (string_of_value rhs)))
in
```

```
let handle_binop stack op lhs rhs =
    let raise_binop_failure op (lhs, rhs) = raise
        (Failure (Printf.sprintf "Invalid binop %s for %s, %s"
            (string_of_bcode op)
            (string_of_value lhs)
            (string_of_value rhs)))
    in
    let normalized_args = normalize_arith_types lhs rhs
    in
    match op with
        Add -> (match normalized_args with
            (Int(l), Int(r))       -> Int(l + r)
          | (Float(l), Float(r))   -> Float(l +. r)
          | (String(l), String(r)) -> String(l ^ r)
          | _                   -> raise_binop_failure op normalized_args)
      | Sub -> (match normalized_args with
            (Int(l), Int(r))     -> Int(l - r)
          | (Float(l), Float(r)) -> Float(l -. r)
          | _                    -> raise_binop_failure op normalized_args)
      | Mul -> (match normalized_args with
            (Int(l), Int(r))     -> Int(l * r)
          | (Float(l), Float(r)) -> Float(l *. r)
          | _                    -> raise_binop_failure op normalized_args)
      | Div -> (match normalized_args with
            (Int(l), Int(r))     -> Int(l / r)
          | (Float(l), Float(r)) -> Float(l /. r)
          | _                    -> raise_binop_failure op normalized_args)
      | Mod -> (match normalized_args with
            (Int(l), Int(r))     -> Int(l mod r)
          | (Float(l), Float(r)) ->
                Int((int_of_float l) mod (int_of_float r))
          | _                    -> raise_binop_failure op normalized_args)
      | Ceq -> Bool((fst normalized_args) = (snd normalized_args))
      | Cne -> Bool((fst normalized_args) <> (snd normalized_args))
      | Clt -> Bool(match normalized_args with
            (Int(l), Int(r))     -> l < r
          | (Float(l), Float(r)) -> l < r
          | _                    -> raise_binop_failure op normalized_args)
      | Cle -> Bool(match normalized_args with
            (Int(l), Int(r))     -> l <= r
          | (Float(l), Float(r)) -> l <= r
          | _                    -> raise_binop_failure op normalized_args)
      | Cgt -> Bool(match normalized_args with
            (Int(l), Int(r))     -> l > r
          | (Float(l), Float(r)) -> l > r
          | _                    -> raise_binop_failure op normalized_args)
      | Cge -> Bool(match normalized_args with
            (Int(l), Int(r))     -> l >= r
          | (Float(l), Float(r)) -> l >= r
          | _                    -> raise_binop_failure op normalized_args)
      | _ -> raise (Failure ("Unknown binop: " ^ string_of_bcode op))

in

let rec exec stack fp sp pc =
    if debug then (
        dump_stack stack 0 sp;
        Printf.printf "%d %d %d %s\n" fp sp pc (string_of_bcode text.(pc));
        flush stdout);
    match text.(pc) with
    Halt        -> raise (Failure "Halt")
  | Push(value) -> stack.(sp) <- value; exec stack fp (sp + 1) (pc + 1)
  | Pop         ->                      exec stack fp (sp - 1) (pc + 1)
  | Add | Sub | Mul | Div | Mod
  | Ceq | Cne | Clt | Cle | Cgt | Cge as op ->
        let lhs = stack.(sp - 2) and rhs = stack.(sp - 1) in
```

```
            stack.(sp - 2) <- handle_binop stack op lhs rhs;
            exec stack fp (sp - 1) (pc + 1)
      | And         -> stack.(sp - 2) <-
            Bool((bool_of_value stack.(sp - 2) &&
                 (bool_of_value stack.(sp - 1))));
            exec stack fp (sp - 1) (pc + 1)
      | Or          -> stack.(sp - 2) <-
            Bool((bool_of_value stack.(sp - 2) ||
                 (bool_of_value stack.(sp - 1))));
            exec stack fp (sp - 1) (pc + 1)
      | Strg(idx)   -> globals.(idx) <- stack.(sp - 1);
                       exec stack fp sp (pc + 1)
      | Ldg(idx)    -> stack.(sp) <- globals.(idx);
                       exec stack fp (sp + 1) (pc + 1)
      | Strl(idx)   -> stack.(fp + idx) <- stack.(sp - 1);
                       exec stack fp sp (pc + 1)
      | Ldl(idx)    -> stack.(sp) <- stack.(fp + idx);
                       exec stack fp (sp + 1) (pc + 1)
      | Call(-1)    -> print_endline (string_of_value stack.(sp - 1));
                       stack.(sp - 1) <- Null;
                       exec stack fp sp (pc + 1)
      | Call(-4)    ->
            (match stack.(sp - 1) with
                VList(l) -> stack.(sp - 2) <- VList(l @ [stack.(sp - 2)])
              | _        -> raise (Failure "Can only append() to list"));
            exec stack fp (sp - 1) (pc + 1)
      | Call(-5)    ->
            (match stack.(sp - 1) with
                VList(l) -> stack.(sp - 1) <- Int(List.length l)
              | _        -> raise (Failure "Can only call length() on a list"));
            exec stack fp sp (pc + 1)
      | Call(idx)   -> stack.(sp) <- Int(pc + 1); exec stack fp (sp + 1) idx
      | Ret(nargs)  -> let ret_fp = ptr_of_value (stack.(fp))
                       and ret_pc = ptr_of_value (stack.(fp - 1)) in
                       stack.(fp - 1 - nargs) <- stack.(sp - 1);
                       exec stack ret_fp (fp - nargs) ret_pc
      | Bz(idx)     -> exec stack fp (sp - 1)
                       (if not (bool_of_value stack.(sp - 1)) then
                            pc + idx
                        else
                            pc + 1)
      | Bnz(idx)    -> exec stack fp (sp - 1)
                       (if bool_of_value stack.(sp - 1) then
                           pc + idx
                       else
                           pc + 1)
      | Ba(idx)     -> exec stack fp sp (pc + idx)
      | Mlst(n)     ->
            stack.(sp - n) <- VList(Array.to_list (Array.sub stack (sp - n) n));
            exec stack fp (sp - n + 1) (pc + 1)
      | Ilst        ->
            (match stack.(sp - 2) with
                VList(l) -> (match stack.(sp - 1) with
                    Int(i) -> stack.(sp - 2) <- List.nth l i
                  | _      -> raise (Failure "Can only index list with int"))
              | _        -> raise (Failure "Cannot index non-list"));
            exec stack fp (sp - 1) (pc + 1)
      | Alst        ->
            (match stack.(sp - 3) with
                VList(l) -> (match stack.(sp - 2) with
                    Int(i) ->
                        stack.(sp - 3) <- VList(list_assign l i stack.(sp - 1))
                  | _ -> raise (Failure "Can only index list with int"))
              | _ -> raise (Failure "Cannot index non-list"));
            exec stack fp (sp - 2) (pc + 1)
      | Read        ->
            if (read_channel stack sp) then
```

```
            exec stack fp sp (pc + 1)
        else
            (fp, sp, pc,
                (if stack.(sp - 1) = Channel(0) then !got_eof else false))
    | Write       -> write_channel stack sp; exec stack fp (sp - 1) (pc + 1)
    | Ent(num)    -> stack.(sp) <- Int(fp);
                     exec stack sp (sp + num + 1) (pc + 1)
    | Par(idx, n) ->
        let state = kernel_states.(!next_state) in
        Array.blit stack (sp - n) state.stack 0 n;
        kernel_states.(!next_state) <- { state with sp = n; pc = idx };
        next_state := (!next_state) + 1;
        exec stack fp (sp - n) (pc + 1)
    | Kent(n)     -> exec stack fp (sp + n) (pc + 1)
    | Term        -> (fp, sp, pc, true)
    | Run         -> (*if debug then Array.iteri
        (fun n s -> Printf.printf "state %d: %d %d %d\n" n s.fp s.sp s.pc)
        kernel_states; *)
        run_kernels ();
        (fp, sp, (pc + 1), true)
  and run_kernels () =
    if (Array.length kernel_states) = 0 ||
        Array.fold_left (fun f s -> f || s.finished) false kernel_states
    then (if debug then print_endline "program finished")
    else (Array.iteri (fun i s ->
        let (fp, sp, pc, finished) = exec s.stack s.fp s.sp s.pc in
            kernel_states.(i) <-
                { s with finished = finished; fp = fp; sp = sp; pc = pc})
        kernel_states;
        run_kernels ())
  in

  exec global_stack 0 0 0
```

Listing A.8: Makefile

```
OCAMLLEX=/usr/bin/ocamllex
OCAMLYACC=/usr/bin/ocamlyacc
OCAMLC=/usr/bin/ocamlc

TEST_DRIVER=./run_tests.sh
TEST_LOG=./tests.log
TEST_FILES=$(wildcard tests/*.pls) $(wildcard tests/*.out) $(wildcard tests/*.in)

MAIN=pls

OBJS=scanner.cmo\
     stdin_scanner.cmo\
     parser.cmo\
     ast.cmo\
     bytecode.cmo\
     compiler.cmo\
     vm.cmo\
     pls.cmo

all : pls tests

.PHONY : tests
tests: $(TEST_DRIVER) $(TEST_FILES) pls
        $(TEST_DRIVER)

pls : scanner.ml parser.ml parser.mli $(OBJS)
        $(OCAMLC) -o $(MAIN) $(OBJS)

scanner.ml : scanner.mll parser.ml parser.mli
        $(OCAMLLEX) scanner.mll
```

```
stdin_scanner.ml : scanner.mll bytecode.ml
        $(OCAMLLEX) stdin_scanner.mll

parser.ml parser.mli : parser.mly
        $(OCAMLYACC) parser.mly

%.cmo : %.ml
        $(OCAMLC) -c $<

%.cmi : %.mli
        $(OCAMLC) -c $<

.PHONY : clean
clean :
        rm -f $(MAIN) *.cmi *.cmo
        rm -f parser.mli parser.ml scanner.ml stdin_scanner.ml parser.output
        rm -f testlog.log

ast.cmo :
ast.cmx :
bytecode.cmo : ast.cmo
bytecode.cmx : ast.cmx
compiler.cmo : bytecode.cmo ast.cmo
compiler.cmx : bytecode.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
pls.cmo : vm.cmo scanner.cmo parser.cmi compiler.cmo bytecode.cmo ast.cmo
pls.cmx : vm.cmx scanner.cmx parser.cmx compiler.cmx bytecode.cmx ast.cmx
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
stdin_scanner.cmo : bytecode.cmo
stdin_scanner.cmx : bytecode.cmx
vm.cmo : stdin_scanner.cmo bytecode.cmo
vm.cmx : stdin_scanner.cmx bytecode.cmx
parser.cmi : ast.cmo
```

A.2 Tests

Listing A.9: run_tests.sh

```
#!/bin/bash
PLS=./pls
PLS_OPTIONS=

DIFF=/usr/bin/diff
DIFF_TARGET=/dev/null

TESTS_DIR=./tests
TEST_SCRIPTS=${TESTS_DIR}/*.pls

NUM_FAILED=0

VERBOSE=0

#GREEN='\e[0;32m'
#YELLOW='\e[1;33m'
#RED='\e[0;31m'
#NO_COLOR='\e[0m'

LOG_FILE=testlog.log
LOG_OUTPUT="Test started at $(date)"

print_usage() {
```

```
  echo "Usage: $0 [-h] [-v] [-o file]"
  echo "    -h: show this help"
  echo "    -v: verbose output"
  echo "    -o <file>: log output to 'file' (default: ./testlog.log)"
  exit 0
}

log_string() {
  LOG_OUTPUT="${LOG_OUTPUT}\n$@"
}

output() {
  echo -e "$@"
  log_string "$@"
}

debug() {
  if [ $VERBOSE -eq 1 ]; then
    output "$@"
  else
    log_string "$@"
  fi
}

run_test() {
  local script_name=$1;
  local input_file=${script_name/%pls/in}
  local output_file=${script_name/%pls/out}

  local test_name=$(basename ${script_name})
  test_name=${test_name%.pls}

  # gtest? never heard of it!
  output  "[ RUN      ] ${test_name}"

  if [ ! -e ${input_file} ]; then
    output "[      WARN ] ${test_name}: input file \"$(basename $input_file)\" does not exist - skipping"
    return 0
  fi

  if [ ! -e ${output_file} ]; then
    output "[      WARN ] ${test_name}: output file \"$(basename $output_file)\" does not exist - skipping"
    return 0
  fi

  local script_output=$(${PLS} ${PLS_OPTIONS} ${script_name} < ${input_file})

  debug "output:\n$script_output"
  local diff_output=$(${DIFF} ${output_file} <(echo "${script_output}"))
  #echo "${diff_output}"
  if [ -z "${diff_output}" ]; then
    output  "[        OK ] ${test_name}"
  else
    debug "diff failed:\n${diff_output}"
    output  "[      FAIL ] ${test_name}"
    NUM_FAILED=$(($NUM_FAILED + 1))
  fi
}

#####

while getopts hvo: c; do
  case $c in
    v) VERBOSE=1;;
    h) print_usage;;
    o) LOG_FILE=${OPTARG}
  esac
```

```
done

for script in ${TEST_SCRIPTS}; do
  run_test ${script}
done

if [ $NUM_FAILED -eq 0 ]; then
  output "All tests passed!"
else
  output "Error: $NUM_FAILED tests failed!"
  echo "See ${LOG_FILE} for details"
fi

echo -e "$LOG_OUTPUT" > ${LOG_FILE}

exit $NUM_FAILED
```

Listing A.10: basic_func.pls

```
function foo()
{
    return 1;
}

function bar(x, y)
{
    print(x);
    print(y);
    return x + y;
}

function baz(n, x)
{
    test = n;
    if (test)
    {
        return x;
    }
    else
    {
        return -x;
    }
}

print(foo());
print(bar(2, 3));
print(baz(2, 42));
print(baz(0, ((2 * 10) + (8 * 8)) / 2));
```

Listing A.11: basic_func.in

```
```

Listing A.12: basic_func.out

```
1
2
3
5
42
-42
```

Listing A.13: basic_kernel.pls

```
a = 0;
```

```
function f(x)
{
    return x + a;
}

kernel k(in x, out y, z)
{
    while (true)
    {
        a = read(x);
        write(y, f(z));
    }
}

k(stdin, stdout, 1);
```

Listing A.14: basic_kernel.in

```
1
2
3
```

Listing A.15: basic_kernel.out

```
2
3
4
```

Listing A.16: basic_pipeline.pls

```
kernel plus_n(in x, out y, n)
{
    while (true)
    {
        write(y, read(x) + n);
    }
}

kernel identity(in x, out y)
{
    while (true)
    {
        a = read(x);
        write(y, a);
    }
}

channel a;

identity(stdin, a);
plus_n(a, stdout, 1);
```

Listing A.17: basic_pipeline.in

```
1
2
3
```

Listing A.18: basic_pipeline.out

```
2
3
4
```

Listing A.19: control_structures.pls

```
// IF TESTS
bool = true;
if (bool)
{
    print("true");
    print(bool);
}
else
{
    // should not print
    print("false");
    print(bool);
}

bool = false;

if (bool)
{
    print("true");
    print(bool);
}
else
{
    // should not print
    print("false");
    print(bool);
}

// TODO: elseif

// WHILE TESTS
run = true;
i = 0;
while(run)
{
    print(i);

    if (i == 10)
    {
        run = false;
    }

    i = i + 1;
}

// FOR TESTS
for (j = 0; j < 10; j = j + 1)
{
    print(j);
}
```

Listing A.20: control_structures.in

```
```

Listing A.21: control_structures.out

```
true
true
false
false
0
1
2
3
```

```
4
5
6
7
8
9
10
0
1
2
3
4
5
6
7
8
9
```

Listing A.22: data_types.pls

```
// INTEGERS
int = 1;
print(int);

// arithmetic returns int
print(int + 1);
print(int - 2); // negative numbers!

// true if non-zero
if (0)
{
    print("DONT PRINT");
}

if (1)
{
    print("1==true");
}

if (-1)
{
    print("-1==true");
}

if (2-1 == 1)
{
    print("2-1==1");
}

if (1 > 0)
{
    print("1>0");
}

if (-1 < 0)
{
    print("-1<0");
}

if (1 == true)
{
    print("1==true");
}

// FLOATS
float = 1.;
print(float);
```

```
float = 0.1;
print(float);

float = 1e3;
print(float);

float = 0.42e2;
print(float);

print(-float);

// float <arithmetic op> int = float
print(float + 1);
print(float - 10);
print(float * 2);

// and the other way around
print(1 + float);
print(10 - float);
print(2 * float);

// float is true if non-zero
if (0.0)
{
    print("DONT PRINT");
}

if (0.1)
{
    print("0.1==true");
}

if (-0.1e1)
{
    print("-0.1e1==true");
}

// STRINGS
string = "hello";
print(string);

string = string + " world";
print(string);

print("embedded\"quotes");

string = "float: " + 123.45;
print(string);

string = "int: " + 42;
print(string);

string = "bool: " + true + " " + false;
print(string);

// strings always == true
if ("")
{
    print("(empty string)==true");
}

if (string)
{
    print("string==true");
}
```

```
// lists tested seperately
```

Listing A.23: data_types.in

```
```

Listing A.24: data_types.out

```
1
2
-1
1==true
-1==true
2-1==1
1>0
-1<0
1==true
1.
0.1
1000.
42.
-42.
43.
32.
84.
43.
-32.
84.
0.1==true
-0.1e1==true
hello
hello world
embedded"quotes
float: 123.45
int: 42
bool: true false
(empty string)==true
string==true
```

Listing A.25: elseif.pls

```
function foo(x)
{
    if (x < 1)
    {
        return "x<1";
    }
    elseif (x == 1)
    {
        return "x==1";
    }
    elseif (x < 5)
    {
        return "1<x<5";
    }
    else
    {
        return "x>=5";
    }
}

print(foo(0));
print(foo(1));
print(foo(2));
print(foo(5));
```

Listing A.26: elseif.in

```
```

Listing A.27: elseif.out

```
x<1
x==1
1<x<5
x>=5
```

Listing A.28: expr.pls

```
// Test for precedence, correct expression evaluation, etc.
print(1 + 1);
print(1 + 2 * 3);
print((1 + 2) * 3);
print(2 / 2 + 3 * 4 - 2);
print(-2 / 2 + -3 * 4 - 2);
print(4 / (2 + 2) * (4 - 2));
print(2 / 2 + 3 * 4 - 2 == 11);
print(2 / 2 + 3 * 4 - 2 != 11);
print(2 / 2 + 3 * 4 - 2 > 10);
print(2 / 2 + 3 * 4 - 2 < 12);
print(2 / 2 + 3 * 4 - 2 >= 10);
print(2 / 2 + 3 * 4 - 2 <= 12);
print(2 % 3);
print(3 % 2);
print(5 % 2 + 1);
print(5 * 2 % 3 + 10);
print(true == true);
print(true != false);
print(false == false);
print (false == true);
print(!true);
print(!false);
print(true && true);
print(true && false);
print(false && false);
print(true || true);
print(true || false);
print(false || false);
print(!false || false);

a = 1;
b = 1;
print(a = 42);
print(a);
print(b = a = 43);
print(a);
print(b);
print(b = (a = 43) - 1);
```

Listing A.29: expr.in

```
```

Listing A.30: expr.out

```
2
7
9
11
-15
2
true
false
```

```
true
true
true
true
2
1
2
11
true
true
true
false
false
true
true
false
false
true
true
false
true
42
42
43
43
43
42
```

Listing A.31: fibonacci.pls

```
channel loop_channel;

kernel fibonacci(n, seed_1, seed_2, in source, out feedback, out output)
{
    // seed the input with the values
    write(feedback, seed_1);
    write(feedback, seed_2);

    // write the first two values as output
    write(output, seed_1);
    write(output, seed_2);

    for (counter = 2; counter <= n; counter = counter + 1)
    {
        // read the two values
        f_1 = read(source);
        f_2 = read(source);

        // calculate the next number
        fib = f_1 + f_2;

        // write the number to the output
        write(output, fib);

        // write the next two numbers back in to the feedback loop
        write(feedback, f_2);
        write(feedback, fib);
    }

    print("All done!");
}

// print the first 10 fibonacci numbers (starting with 0 and 1) to stdout
fibonacci(10, 0, 1, loop_channel, loop_channel, stdout);
```

Listing A.32: fibonacci.in

```
```

Listing A.33: fibonacci.out

```
0
1
1
2
3
5
8
13
21
34
55
All done!
```

Listing A.34: func_recursion.pls

```
function fact_impl(i, n, acc)
{
    if (i >= n)
    {
        return acc * i;
    }
    else
    {
        return fact_impl(i + 1, n, acc * i);
    }
}

function fact(n)
{
    return fact_impl(1, n, 1);
}

print(fact(5));

function is_odd(i, n)
{
    if (i % 2 != 0)
    {
        print(i + " is odd");
    }

    if (i >= n)
    {
        return null;
    }

    is_even(i + 1, n);
}

function is_even(i, n)
{
    if (i % 2 == 0)
    {
        print(i + " is even");
    }

    if (i >= n)
    {
        return null;
    }
```

```
    is_odd(i + 1, n);
}

is_even(0, 10);
```

Listing A.35: func_recursion.in

```
```

Listing A.36: func_recursion.out

```
120
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

Listing A.37: global_vars.pls

```
global = 5;

function print_global()
{
    print(global);
}

print_global();

kernel g(in input, out output)
{
    while(true)
    {
        global = read(input);
        print_global();
        write(output, global);
    }
}

g(stdin, stdout);
```

Listing A.38: global_vars.in

```
1.23
hello
true
false
42
```

Listing A.39: global_vars.out

```
5
1.23
1.23
hello
hello
true
true
false
```

```
false
42
42
```

Listing A.40: list_test.pls

```
function print_list(l)
{
    foreach(x : l)
    {
        print(x);
    }
}
a = [1, 2, 3];
print(a[0]);

a = append(a, 42);
print(a[3]);

print_list(a);

print([5,6,7][1]);

b = [[1, 2], [3, 4]];
print(b[0][1]);

//a[1] = 4;
//print_list(a);
```

Listing A.41: list_test.in

```
```

Listing A.42: list_test.out

```
1
42
1
2
3
42
6
2
```

Listing A.43: scope.pls

```
i = 0;

{
    print ("i == " + i);
    i = 2;
}

print("i == " + i);

{
    j = 2;
}

// uncommenting the next line will produce an error
// print(j);

for (j = 0; j < 2; j = j + 1)
{
    print("j == " + j);
```

```
}

// uncommenting this line will produce an error
//print(j);

{
    k = 5;
    {
        l = 10;
        print ("k == " + k);
        k = 6;
    }

    print("k == " + k);

    // uncommenting this line will produce an error
    // print(l);
}
```

Listing A.44: scope.in

```
```

Listing A.45: scope.out

```
i == 0
i == 2
j == 0
j == 1
k == 5
k == 6
```

Listing A.46: splitter.pls

```
// On an input of the integers 1 to 10, this program will output 1-8. This is
// because each "buffer" kernel does two reads. Since 10%2 == 2, these reads
// will hang forever. Therefore we just terminate the program.

kernel src(in data, out even, out odd)
{
    while(true)
    {
        token = read(data);

        is_even = token % 2 == 0;

        if (is_even)
        {
          write(even, token);
        }
        else
        {
          write(odd, token);
        }
    }
}

kernel buffer(in data_in, out data_out)
{
    while (true)
    {
        first = read(data_in);
        second = read(data_in);

        write(data_out, first);
        write(data_out, second);
```

```
    }
}

kernel sink(in even_data, in odd_data, out output)
{
    while (true)
    {
        write(output, read(odd_data));
        write(output, read(even_data));
    }
}

channel src_to_even, src_to_odd, even_to_sink, odd_to_sink;

src(stdin, src_to_even, src_to_odd);
buffer(src_to_even, even_to_sink);
buffer(src_to_odd, odd_to_sink);
sink(even_to_sink, odd_to_sink, stdout);
```

Listing A.47: splitter.in

```
1
2
3
4
5
6
7
8
9
10
```

Listing A.48: splitter.out

```
1
2
3
4
5
6
7
8
```

Bibliography

[1] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74 : proceedings of IFIP Congress 74*. North-Holland American Elsevier, 1974.

[2] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.