SPY - Final Report

Simplified **Py**thon

A strongly typed, general purpose language inspired by Python

Sanil Shah (ss4924)

Chapter 1: Introduction

SPY is a strongly typed, easy to use language inspired by features from Python and functional languages like OCaml and SML. The goal is to create a type-safe version language with functional elements that can be used easily by Python programmers with a similar syntax that will compile down to Javascript. The language is compiled down to Javascript to allow it to be easily run in a web environment. A lot of functional programming languages (like LISP) can often have complicated syntax that is hard to understand and unfamiliar to most programmers. Since Python is quickly becoming language of choice for introductory programming classes, SPY aims at offering a familiar yet functional alternative to first-time Python users and programmers who wish to dip their toes into functional programming. The language will not have all the robust features of Python, but will contain a small subset that can be used to create fairly robust algorithms. The language design aims at emphasizing type safety while minimizing the number of keywords, keeping the syntax as similar to Python as possible.

Chapter 2: Tutorial

This section will explore the simple ways to use the SPY language. It will also explain differences between SPY and Python to enable first time users to easily adapt to SPY syntax. Further it will explain how to use the compiler to compiler a .spy file into a .js file.

Requirements

There are several software requirements for running the SPY compiler. Since the compiler is written in Ocaml and compiles down to JS, both OCaml and node.js must be installed on the machine. Further several ocaml core libraries are required as well such as core, ocamlfind and menhir which can all be installed using the OCaml package manager, OPAM. node.js is used to run the compiled code and test if the output is correct.

Usage Instructions

The compiler can be built by running `"make"` in the SPY directory. This will produce a `"spy.out"` file that be run with a spy program as follows:

```
./spy.out example.spy
```

If example.spy is syntactically correct then this will produce an `"out.js"` file which can be run using the node command as follow:

```
node ./out.js
```

This will print out the output of the program to stdout. If the program has syntax errors the compiler will output the issues in the program which can then be fixed by the programmer.

Python vs. SPY

SPY is inspired by Python and the syntax is as close to Python syntax as possible. SPY is type safe but the types will be inferred by the compiler so the language will not need special keywords to specify the types. In fact, like Python there will be no explicit way to specify types, however type safety will be ensured by the compiler once types are inferred.

Comments

Python	SPY

`# This is a single line comment`
 	
`"""`
`This is a`
`multi line comment`
`"""`
	`# This is a single line comment`
	
	`Multiple line comments aren't allowed.`
	`Anything after a # symbol on a line will`
	`be ignored and treated as a comment.`

Literals

Python	SPY

`42`
`0.42`
`True`
`False`
	`42`
	`0.42`
	`true`
	`false`

Strings

Python	SPY

`'Hello world'`
`"Hello world"`
`"Hello \n world"`
`"Hello" + "world"`
`"Hello" + 5`
	`Single quoted strings aren't allowed`
	`"Hello world"`
	`"Hello \n world" (escape special chars)`
	`"Hello" ^ "world"`
	`Mixed type concat isn't allowed`

Lists

Python	SPY

`[1, "hi", 0.42]`
`[1, 2, 3]`
`[1] + [2, 3]`
`[1, 2] + [3, 4]`
	`Mixed type lists are aren't allowed`
	`[1, 2, 3]`
	`1 :: [2, 3] (cons)`
	`[1, 2] @ [3, 4] (append)`

Dictionaries

Python	SPY

`{"foo": 1, "bar": "baz"}`
`{"foo": 1, "bar": 2}`
	`Mixed value dictionaries aren't allowed`
	`{"foo": 1, "bar": 2}`

Functions

Blocks in Python are denoted by white-space (tabs or space characters). Instead, SPY will use the "end" keyword to denote the end of a block, and white-space will be ignored just like any other programming language. This will apply to conditionals as well as function definitions. New lines are used to determine the end of an expression. All other white-space will be used purely for convenience and readability.

Python	SPY
`def add(x, y):`	`add = lambda(x, y):`
`  return x + y`	`  x + y`
	`end`
`add = lambda x, y: x + y`	The function body must be on a newline.

Conditionals

Python	SPY
`if x < y:`	This isn't allowed in SPY. The else
`  return 42`	construct is required.
`if x < y:`	`if x < y:`
`  return 42`	`  42`
`elif x > y:`	`elif x > y:`
`  return -42`	`  -42`
`else:`	`else:`
`  return 0`	`  0`
	`end`

SPY Code Samples

```
# Euclid's GCD
gcd = lambda(a, b):
    if a == b:
        a
    elif a > b:
        gcd(a - b, b)
    else:
        gcd(b - a, a)
    end
end
```

```
# Combines two lists into a list of lists
combine = lambda(x, y):
    if len(x) != len(y) or len(x) == 0:
        []
    else:
        [hd(x), hd(y)] :: combine(tl(x), tl(y))

# Calling combine with incorrect arguments (compiler error)
a = combine(1, 2)

# Calling combine correctly (returns [[1,3], [2,4]])
b = combine([1, 2], [3, 4])
```

Chapter 3: Language Reference Manual

This chapter covers the rules for the SPY language syntax and semantics. It describes ways in which the language can be used and the behavior of the language when used in various ways.

Types

Types are dynamically inferred at runtime in Python and will similarly be inferred in SPY as well so that they do not have to be explicitly specified. The inferred types are described here to make compiler messages more understandable and in case explicit typing is necessary at any time in the future. The words used to describe the types will be reserved as keywords.

Primitive Types

There are five primitive types in SPY.

1. **void:** This is a special type used when expressions do not return a value. This is the case for statements like print or file reading operations that may be supported. These expressions have side effects (such as printing to stdout) but do not return a value and hence have a type void.
2. **bool:** This represents a logical value which can either be true or false. The keywords "true" and "false" are reserved to represent literals of this type.
3. **int:** The int type is used to represent literals between +2^31 and -2^31. Literals with no decimal points or floating parts will be inferred as integers.
4. **num:** The num type is used to represent numbers containing a decimal part or an exponential portion. Numbers like 0.42 or 1e5 will be inferred as floating point numbers.
5. **string:** The string type is used to represent literals containing text. It is a sequence of characters encoded in the ASCII format. Each character occupies 16 bits and will follow in order from the starting position which will contain the first character of the string. Strings will be inferred as any sequence of ASCII characters between double quotes or based on the usage of the caret operator.

Functions to allow casting between various types will be provided by the SPY language.

Complex Types

There are two complex types in SPY.

1. **list:** The list data structure can be used to store a sequence of values. This can be useful when keeping track of multiple primitives that need to be referred to repeatedly. It serves as a way to express one or more related values together. Unlike Python, all elements of a list is SPY must contain the same type of element. Lists can also contain other lists so long as every element list contains the same type of elements.
2. **dict:** The dictionary data structure is another important feature of SPY and are similar to dictionaries in Python. However unlike Python, all keys in the dictionary must be of the same type as well as all the values. Keys and values may have different types but all the keys have to

be homogenous and all the values have to be homogenous. The type of the dictionary will be inferred during initialization or when it is accessed.

Function Types

Like everything else in SPY, functions are expressions as well and evaluate to expressions. The type of a function is represented by a list of types of its formal arguments and its return type which may be one of the primitive or composite types, or a function type in itself..

Type Declarations

Explicit type declarations aren't permitted in SPY (just like Python). Assignment to variables is done by specifying a valid identifier followed by the equals sign (" = ") followed by any valid expression. Rules for valid identifiers are found below. Examples of assignments are shown here:

```
a = 1 # int
b = 1.5 # float
c = -0.00005 # float
d = "cat" # string
e = true # bool
f = [1, 2, 3, 4] # list(int)
g = ["a", "b", "c", "d"] # list(string)
h = {"cat": 1, "dog": 2} # dict(string, int)
i = {"cat": [1], "dog": [2]} # dict(string, list(int))
```

Lexical Conventions

The rules followed by the parser and the lexer to break the program into acceptable tokens are listed below. The rules for syntax are very similar to Python with minor differences but are listed out in completeness below to remove any ambiguity.

Comments

Only single line comments are allowed in SPY. Anything on a line following a "#" is considered a comment and ignored by the compiled. Multi line comments are not supported in SPY.

```
# This entire line is a comment
a = "cat"   # The rest of this line is a comment
```

Lines

Each line in SPY represents a separate expression (except for blocks which we will discuss later). Since there is no way to delimit expressions (like a semicolon in JS or Java), each line will be treated as a separate expression. SPY will recognize the linefeed (ASCII: '\n') as a line termination sequence.

Whitespace

All non line termination white-space characters can be used to separate tokens. This includes horizontal tab ('\t'), form feed ('\f') and carriage return ('\r). The white-space characters will be used exclusively to separate tokens and hold no other meaning.

Indentation

Unlike Python, SPY doesn't follow a strict indentation policy to represent code blocks. Instead the end of a code block will be indicated using the "end" keyword provided in the language. This allows users to use any indentation they want. The "end" keyword will be used to demarcate function definitions as well as if-else blocks.

Identifiers

Identifiers are used to represent various entities in SPY and to store their values. All reserved SPY keywords mentioned in the section below or types mentioned above cannot be valid identifiers. Further, reserved JS keywords will not be valid identifiers either. Identifiers can be named starting with at least a single lowercase letter, followed by any number of lowercase letters, uppercase letters, digits or underscores. Identifiers will not start with an uppercase letter, a digit or an underscore.

Identifier regular expression:

```
id = ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

Valid Identifiers	Invalid Identifiers
x	X (cannot start with uppercase letters)
name	1stName (cannot start with a digit)
angle1	name&age (invalid character)
last_name	lambda (reserved SPY keyword)
dateOfBirth	int (reserved SPY type)

Keywords

The below keywords are reserved and have special meaning in SPY. The are reserved for use by the programming language itself and may not be used as names of identifiers elsewhere in the program. They serve a special function and cannot be used to serve any other purpose. Keywords used by SPY are listed below:

```
lambda, end, if, elif, else, hd, tl, keys, and, or, not, true, false
```

Separators

This is the list of characters that separate tokens other than white-space as mentioned above. These are simple single characters that can be used to separate tokens and often have a special meaning when used in specified format to represent lists or dictionary literals.

Character	Name
'('	Left paren
')'	Right paren
'['	Left square
']'	Right square
'{'	Left brace
'}'	Right brace
','	Comma

Literals

Literals are used to represent values of primitive or composite types in SPY. There are six kinds of literals as listed below.

1. **Boolean literals:** The bool type literals can be two values, "true" or "false" represented by the ASCII characters that form the words. These are reserved keywords so they may not be used as identifiers and are written without the surrounding quotes to signify the boolean literals.

```
bool = true | false
```

2. **Numeric literals:** Numeric literals represent values of the num type. These can be a sequence of one or more digits starting with a non zero digit or a single zero. It can be used to represent numbers between 0 and 2^31. Further numeric literals can represent values of the float type. They can be used to represent fractional or decimal numbers. Float literals have whole number part followed by a dot followed by a fractional part. The whole number part can be defined by the same rules as integer literals above. The dot is mandatory. The fractional part can be defined as a sequence of any number of digits. Negative numeric literals will be created using the unary operator for negation which will be covered later in this manual.

```
n = (['1'-'9']['0'-'9']* | '0') | (['1'-'9']['0'-'9']* | '0') '.' ['0'-'9']*
```

3. **String literals:** String literals are used to represent textual data. They are denoted by a sequence of zero or more ASCII symbols enclosed in double quotes. Certain special characters are denoted by escape sequences, which are a backslash followed by the special character. The following escape sequences are supported to account for special characters like tabs, new lines, and

quotes: horizontal tab ("\t"), newline ("\n"), backslash ("\\"), double-quotes ("\"") and carriage return ("\r").

```
string = '"' ([^'"' '\'] | '\\' ['"' '\\' 'n' 'r' 't'])* '"'
```

4. **List literals:** List literals are used to represent values of the list type. They are denoted by a sequence of zero or more comma separated expressions surrounded by square braces. The square braces are required to specify a list literal. They may contain no elements in which case the list is considered to be empty. Examples of valid list literals are provided above under list.

5. **Dictionary literals:** A dictionary literal is used to store pairs of related values. This is similar to a map in other languages. It is denoted by a left brace ("{") followed by a list of zero or more comma separated key-value pairs and ending with the right brace. ("}"). The braces are required to specify a dictionary literal. The key-value pairs are denoted by an expression followed by a colon (":") followed by another expression. The first expression represents the key while the second expression represents the value. All keys must be of the same type and all values in the dictionary but also be of the same type, keeping the dictionary consistent.

Operators

The following characters are used as operators in the language and carry a special meaning that is described below. Their behavior is defined as part of the language and cannot be changed.

Character	Name
`'+'`	`addition`
`'-'`	`Subtraction or negation`
`'*'`	`multiplication`
`'/'`	`division`
`'%'`	`modulus`
`'^'`	`string concatenation`
`'::'`	`list insert`
`'@'`	`list append`
`'<'`	`less than`
`'<='`	`less than or equal to`
`'>'`	`greater than`
`'>='`	`greater than or equal to`
`'='`	`assign`
`'=='`	`equals`
`'!='`	`not equals`
`and`	`logical and`
`or`	`logical or`
`not`	`logical negation`

Operators in SPY may be unary or binary. There are only two unary operators (logical negations "not" and arithmetic negation "-"). All other operators are binary and the types of operands that they may act upon is defined below. Binary operators are used in the infix notation and are written between two expressions that act as the operands.

Arithmetic Operations

Arithmetic operations only allow operands of type num. The five arithmetic operations are plus, minus, multiply, divide and modulo. The minus sign can also be used as a unary operator to negate numbers. The operands can be expressions so long as both expressions evaluate to literals of the num type. The result of these operations is also a num type.

1. **Plus:** Add two values

```
1 + 2  # evaluates to 3
```

2. **Minus:** Subtract the second value from the first

```
3.0 - 1.5  # evaluates to 1.5
```

3. **Multiply:** Product of two values

```
5 * 2  # evaluates to 10
```

4. **Divide:** Divide the first value by the second

```
5.0 / 2.0  # evaluates to 2.5
```

5. **Modulo:** The remainder obtained when dividing the first value by the second

```
6 % 4  # evaluates to 2
```

6. **Unary negation:** The arithmetic negation of the given value

```
-3 # evaluates to -3
```

String Operations

There is only a single string operation. The caret ("^") symbol is used to concatenate two strings. Both operands must be string literals or expressions that evaluate to string literals. The result of the concatenation operation is also a string literal.

1. **Concat:** Combine the two strings into a single string

```
"cat" ^ "cat" # evaluates to "catcat"
```

Relational Operations

Relational operations require operands of either type num or string. The four relational operations are greater than, greater than or equal to, less than and less than or equal to. The operands are expressions that evaluate to literals of either num or string types or variables specified by identifiers of any of those types. Both operands must be of the same type. For string types comparison is done based on the ordering of the alphabet. The result type of these operations is of type bool (it may be either true or false).

1. **Less than:** Is the first value strictly less than the second value

```
1 < 2  # evaluates to true
```

2. **Less than equals:** Is the first value less than or equal to the second value

```
3.0 <= 1.5  # evaluates to false
```

3. **Greater than:** Is the first value strictly greater than the second value

```
"cat" > "cat"  # evaluates to false
```

4. **Greater than equals:** Is the first value greater than or equal to the second value

```
"cat" >= "cat"  # evaluates to true
```

Comparison Operations

There are two types of comparison operations. These are equals ("==") and not equals ("!="). These can be used with operands of any primitive or composite type. Once again both operands must be of the same type. The result of these operations is also a boolean. Comparison for composite types is done by structure (similar to python). This means that two lists will be equal if they contain the same elements in the same order, and two dictionaries will be equal if they contain the same keys with the same value for each key.

1. **Equals:** Is the first value logically, mathematically or structurally equal to the second value

```
"cat" == "cat"  # evaluates to true
"cat" == "dog"  # evaluates to false
3.0 == 3.0  # evaluates to true
3 == 4  # evaluates to false
[1, 2, 3] == [1, 2, 3] # evaluates to true
{1: "cat"} == {1: "dog"} # evaluates to false
```

2. **Not equals:** Are the two values logically, mathematically or structurally not equal

```
"cat" != "cat"  # evaluates to false
"cat" != "dog"  # evaluates to true
3.0 != 3.0  # evaluates to false
3 != 4  # evaluates to true
[1, 2, 3] != [1, 2, 3] # evaluates to false
{1: "cat"} != {1: "dog"} # evaluates to true
```

Assignment Operations

The assignment operator ("=") is used to assign a value to an identifier. The value may be of any primary, composite or function type. The identifier is on the left side of the "=" sign. The expression on the right side is evaluated and then assigned to the identifier on the left side.

1. **Assignment:** Assign the value on the right to the identifier on the left

```
a = 3 + 4  # a is assigned 7
b = "cat"  # b is assigned "cat"
```

Logical Operations

There are three types of logical operations. These are logical and ("and"), logical or ("or") and logical negation ("not"). These can be only be used with operands that are of the bool type. Any expression used with these operators must evaluate to a bool literal. The resulting type of these operations is also a bool type. The logical or and logical and operators are binary while the logical negation operator is unary.

1. **Logical and:** Are both the operands true. If not then this is false.

```
true and false  # evaluates to false
true and true  # evaluates to true
```

2. **Logical or:** Are either of the operands true. If not then this is false.

```
true or false  # evaluates to true
false or false  # evaluates to false
```

3. **Logical negation:** If the operand is true, this is false, if the operand is false, this is true.

```
not true  # evaluates to false
not false  # evaluates to true
```

List Operations

There are two types of list operations. These are cons ("::") and append ("@"). Both are binary operators used with the infix notations. The result of both operations is a list type. Both operands for these operations need to contain same type of elements. The correct usage for each is described below.

1. **Cons:** Adds the value on the left to the list on the right. The operand on the left must be of the same type as the elements in the list. The operand on the right must be a list.

```
3 :: [2, 1]  # evaluates to [3, 2, 1]
"cat" :: ["dog", "cow"]  # evaluates to ["cat", "dog", "cow"]
```

2. **Append:** Appends the two lists together. Both lists must have the same type of elements. Both operands must be valid lists.

```
[3] @ [2, 1]  # evaluates to [3, 2, 1]
["dog", "cow"] @ ["cat"]  # evaluates to ["dog", "cow", "cat"]
```

Dictionary Operations

There are three types of dictionary operations. This includes setting a value in a dictionary, getting a value from the dictionary and getting all the keys in the dictionary. Dictionary operations are different from all other operations in that they do not have a specific special character operator associated with them. However they do have a special syntax to access elements in the dictionary and to set values for the dictionary. Further the list of all the keys of the dictionary can be obtained by using the "keys" keyword. The usage for these is shown below.

1. **Get:** Gets the value of key in the dictionary. The return type of this operation is the same as the type of the values of the dictionary being accessed. The syntax is similar to Python. The name of the dictionary followed by the value of the key surrounded by square braces as shown below.

```
a = {"foo": 1, "bar": 2}
a["foo"]   # evaluates to 1
a["bar"]   # evaluates to 2
```

2. **Keys:** The return type of this operation is a list of the same type as the keys of the dictionary. It can be used as a way to iterate over the values in the dictionary. Keys acts as a unary operator where the operand must be a dictionary and the result is a list.

```
a = {"foo": 1, "bar": 2}
keys a  # evaluates to ["foo", "bar"]
```

Other useful dictionary operations such as "contains", "del" or "count" will be provided by the standard library.

Operator Precedence

The operator precedence specifies the order in which operations occur when there is more than one operator in an expression. All operators in SPY are left associative except for assign ("=") and cons ("::") which is left associative. The operator precedence can be overruled by using parentheses to dictate which operations occur first.

```
unary negation
*, /, %
+, -
<=, >=, <, >, ==, !=
not
and, or
^, ::, @, keys
=, <-
```

Functions

Functions are an integral part of any functional language. All functions in SPY are expressions. Functions are treated as first class citizens and can be passed to and from other functions as arguments or return values. Functions can be assigned to named identifiers using the "def" keyword as demonstrated above. Further smaller functions that can fit on a single line can be created using the lambda keyword with the syntax shown above. The argument list for functions is specified as a comma separated list of expressions containing one or more element and surrounded by parentheses. The parentheses are mandatory even if there are no arguments to the function.

Function declarations

Functions in SPY are simply expressions that can be assigned to variables. Functions are declared within a "lambda" and "end" block. For function declarations the lambda keyword must be followed by a left paren separator. This is then followed by zero or more identifiers that are the names for the arguments of this function and then a closing right paren. This should then be followed by a colon and a newline which indicates the start of the function block. The block ends with the end keyword. The body of the function can be any sequence of expressions. Local variables defined in the function will be accessible anywhere within the function, including in nested functions. The last expression of the function is returned to the calling function.

```
func_name = lambda(arg1, arg2,...):
  # block of expressions
  # last expression is returned
end
```

This syntax can be parsed with the following structure in the grammar:

```
id = lambda ( optional_args ) : <new-line> function_body end
```

Above, optional_args is a comma separated list of identifiers, function_body is a list of expressions separated by newlines in the code. The last expression of function_body is returned.

The types of a function including the return type and types of its argument list will be inferred by the compiler. This represents the type of the function and the types of the arguments when calling a function must match the types that are inferred by the compiler.

Function invocation

A function is invoked by using its name which is declared during function declaration. A function must be invoked with actual arguments that match the types of the formal arguments for the function. These arguments can be any expression so long as the expression evaluates to the expected type for the formal argument. A function invocation evaluates to a value that is of the type which is the return type of the function being invoked. The syntax for function invocation is described below:

```
a = func_name(arg1, arg2,...)

# Examples
b = factorial(3)  # b is assigned 3 * 2 * 1 which is 6
c = isPalindrome("dog" ^ "cow")  # c is assigned false. dogcow isn't a palindrome
```

This syntax can be parsed with the following structure in the grammar:

```
id ( optional_args )
```

The optional_args must be a list of expression that evaluate to types that match the function definition.

Expressions

Everything in SPY is an expression. This allows an entire program to be defined as a list of expressions. Expressions are composed of identifiers, literals, operators, function definitions and function calls. Expressions can be of the following types:

1. Unary operations
2. Binary operations
3. Assignment operations
4. Function definitions
5. Function invocations
6. Literals of primitive types
7. Literals of composite types
8. Block of expressions (expression list)
9. Identifiers
10. If statement, optional elif statements and else statement
11. Element of list or dictionary

A program is a sequence of expressions separated by newline characters.

Blocks and Scope

A block is simply a list of expressions that are collectively treated as one. In SPY blocks are encountered inside function declarations and if-elif-else statements.

The syntax for if-elif-else statements is exactly like Python (without the tab spacing) and ending with the "end" keyword. All expression between the colon (":") after the if statement and the next elif or else are treated as a block. Similarly everything between the colon after the elif statement and the next elif or else is treated a block.

Identifiers in SPY are statically scoped so that they can only be accessed within the block in which they are declared. Global variables may exist and may be reassigned but they must remain of the same type throughout the program. If an identifier cannot be found in its current scope then the parent scope is checked recursively until we get to the global scope. If it is still not found then an error is returned if an unfound identifier is accessed.

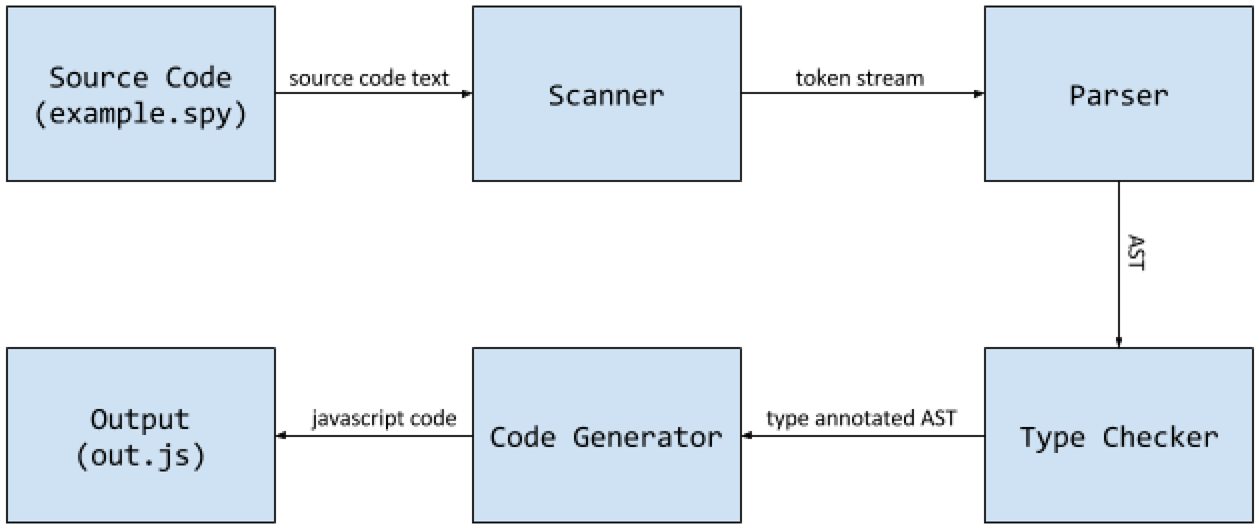
Chapter 4: SPY Architecture

Architecture Overview

The SPY compiler consists for four major components.

1. The **scanner** lexically analyzes the input program and converts it into a stream of tokens. Any invalid characters are surfaced by the scanner as a syntax error.
2. The **parser** accepts the stream of tokens from the scanner and outputs an abstract syntax tree (AST) with ambiguous data types. Any incorrect syntax errors are surfaced by the parser.
3. The **type checker** consumes the ambiguous AST generated by the parser and generates a strongly typed, annotated AST. Any type mismatch or argument count mismatch errors are surfaced by the type checker.
4. The **code generator** is the final step of the process and it outputs compiled valid javascript code. Any remaining type errors or scoping errors are surfaced by the code generator.

Architecture Diagram



Chapter 5: Test Plan

The test plan for the SPY compiler aimed at being both simple but robust at the same time. I wanted to ensure that the type inference mechanism was tested thoroughly while also making sure that the functionality was preserved for the functions being written. The best way to do this was through automated end to end integration tests.

These tests covered both passing and failing test cases. In this way I was able to test functionality (for passing cases) as well and the correct functionality of the compiler in the case of a syntax or type error.

The testing mechanism was borrowed from the JSJS compiler and is as follows:

1. A test file (pass-test1.spy) and a test output file (pass- test1.txt) was created for each test case.
2. The test name specifies whether the test is expected to pass or fail (preceding the "-")
3. The .spy file is then compiled and the resulting JS is executed using node.
4. If the compilation fails then the test runner ensures that the test was expected to fail.
    a. In this case the output from running the test is the string version of the exception that was thrown by the compiler
    b. The output file for this test case is compared against the exception that was surfaced to ensure that the correct error was surfaced
5. If the compilation is successful then the output of running the code is compared against the test output file using a diff to make sure that they are the same. The output in the test output file is considered the golden standard (created manually) and any diff in the output would indicate an error.

These automated tests proved to be crucial in developing SPY since they allowed me to quickly and easily run the entire test suite while making incremental changes to the compiler. More tests can always be added by simply adding two more files to represent a test case.

The test cases devised were entirely decided based on the functionality that needed to be tested. Each of the various features in the SPY language were tested separately for both failure and passing and more complex programs were written as well to ensure that the code worked correctly even with a larger program to compile.

Chapter 6: Lessons Learned

I learned a lot for this class and from working on this project. I would have loved to work on it as part of a group since a large part of this class also revolves around working with other people but there was plenty to take away from the project.

I learned that compilers are extremely complicated and a lot of thought, design, and strong opinions need to go into coming up with a new language. Further, I realized that new languages should only be created to serve a very specific purpose (or not be created at all).

Learning OCaml and working with a functional language was another incredible learning. They are extremely powerful and it always amazes me how things just work as soon as it compiles. The SPY language was inspired by Ocaml and Python in that it tries to be functional as far as possible so I really enjoyed working with and trying to create a functional language.

I also became acutely aware of the amazing functionality that modern day compilers offer that we tend to take for granted as software engineers. I have a newfound respect for programming languages created so far, finally being able to understand the complexity and effort behind creating one.

Finally, as advice for students in future semesters, I'd recommend starting early. Building a compiler is pretty much an endless process and it's extremely tempting to continue adding features. Or there are never enough tests. Or there is an edge case that you missed. Or there will be some functionality that you will suddenly want to add. Starting early will enable you to gain the most from this class.

Note: This compiler was initially supposed to compile to C which would have been considerably harder to implement with type inference as well. I ran out of time to compile to C which is why this project instead generates Javascript which is more forgiving. Further my lack of familiarity with C and insistence on having complex data types like lists and dictionaries made C a much harder language to compile to. This is partially why I'd advise future students to start early and to design their input and output languages in a way that makes creating a compiler easier.

Chapter 7: SPY Compiler Source Code

scanner.mll

```
{
  open Parser
}

let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ascii = [' '-'!' '#'-'[' ']'-'~']
let whitespace = [' ' '\t' '\r']
let escape_chars = ['\\' '"' 'n' 't' 'r']
let number = digit+ '.'? digit*
let str = (ascii | '\\' escape_chars)*
let id = ['a'-'z'] (alpha | digit | '_')*

rule token = parse
  whitespace    { token lexbuf }
  |  '#'        { comment lexbuf }
  | '\n'        { EOL }
  | '('         { LPAREN }
  | ')'         { RPAREN }
  | '['         { LSQUARE }
  | ']'         { RSQUARE }
  | '{'         { LBRACE }
  | '}'         { RBRACE }
  | ':'         { COLON }
  | ','         { COMMA }
  | '+'         { PLUS }
  | '-'         { MINUS }
  | '*'         { MULTIPLY }
  | '/'         { DIVIDE }
  | '%'         { MODULUS }
  | '<'         { LT }
  | '>'         { GT }
  | '='         { ASSIGN }
  | '!'         { NOT }
  | '^'         { CARET }
  | '@'         { APPEND }
  | "::"        { CONS }
  | "<="        { LTE }
  | ">="        { GTE }
  | "=="        { EQ }
  | "!="        { NEQ }
  | "and"       { AND }
  | "or"        { OR }
  | "not"       { NOT }
  | "lambda"    { LAMBDA }
  | "true"      { TRUE }
  | "false"     { FALSE }
  | "if"        { IF }
  | "elif"      { ELIF }
  | "else"      { ELSE }
  | "end"       { END }
  | "none"      { NONE }
  | eof         { EOF }
  | number as num         { NUM_LIT(float_of_string num) }
  | '"' (str as s) '"'    { STR_LIT(s) }
  | id as identifier      { ID(identifier) }
  | _ as char             { raise (Failure("SyntaxError: Invalid syntax -> " ^
Char.escaped char)) }

and comment = parse
    '\n'  { token lexbuf }
  | _     { comment lexbuf }
```

ast.ml

```
type op =
    Add | Sub | Mult | Div | Mod
  | Caret
  | And | Or
  | Eq | Neq | Lt | Lte | Gt | Gte
  | Cons | Append

type uop = Neg | Not

type expr =
    VoidLit
  | NumLit of float
  | StringLit of string
  | BoolLit of bool
  | ListLit of expr list
  | DictLit of (expr * expr) list
  | FunLit of string list * expr
  | Val of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of expr * expr list
  | Element of string * expr
  | Block of expr list
  | If of (expr * expr) list * expr
  | NoExpr

type program = expr list

type pType =
    T of string
  | TAny
  | TNum
  | TString
  | TBool
  | TVoid
  | TFun of funcType
  | TList of pType
  | TDict of pType * pType
and funcType = pType list * pType

type aexpr =
    AVoidLit of pType
  | ANumLit of float * pType
  | AStringLit of string * pType
  | ABoolLit of bool * pType
  | AListLit of aexpr list * pType
  | ADictLit of (aexpr * aexpr) list * pType
  | AFunLit of string list * aexpr * pType
  | AVal of string * pType
  | ABinop of aexpr * op * aexpr * pType
  | AUnop of uop * aexpr * pType
  | AAssign of string * pType * aexpr * pType
  | ACall of aexpr * aexpr list * pType
  | AElement of string * aexpr * pType
  | ABlock of aexpr list * pType
  | AIf of (aexpr * aexpr) list * aexpr * pType
```

parser.mly

```
%{ open Ast %}

/* Tokens */
%token EOL EOF
%token PLUS MINUS MULTIPLY DIVIDE MODULUS
%token LT LTE GT GTE EQ NEQ TRUE FALSE
```

```
%token AND OR NOT
%token LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE
%token ASSIGN LAMBDA END IF ELIF ELSE
%token CARET CONS APPEND
%token COLON COMMA
%token NONE
%token <float> NUM_LIT
%token <string> STR_LIT
%token <string> ID

%right ASSIGN
%right CONS
%left APPEND
%left CARET
%left OR
%left AND
%left NOT
%left LTE GTE LT GT EQ NEQ
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULUS
%left NEG

%start program
%type <Ast.program> program

%%

program:
  expr_list EOF   { $1 }

delimited_expr:
    EOL           { NoExpr }
  | expr EOL      { $1 }

expr_list:
  /* nothing */              { [] }
  | delimited_expr expr_list { if $1 = NoExpr then $2 else $1 :: $2 }

fun_literal:
  LAMBDA LPAREN formals_opt RPAREN COLON EOL expr_list END  { FunLit($3, Block($7))
}

formals_opt:
  /* nothing */  { [] }
  | formal_list  { List.rev $1 }

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

if_expr:
  IF expr COLON EOL expr_list elif_list ELSE COLON EOL expr_list END  { If(($2,
Block($5)) :: $6, Block($10)) }

elif_list:
  /* nothing */     { [] }
  | elif elif_list  { $1 :: $2 }

elif:
  ELIF expr COLON EOL expr_list  { ($2, Block($5)) }

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

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

kv_pairs:
  /* nothing */   { [] }
  | kv_pair_list  { List.rev $1 }

kv_pair_list:
    expr COLON expr                         { [($1, $3)] }
  | kv_pair_list COMMA expr COLON expr  { ($3, $5) :: $1 }

literals:
    NUM_LIT    { NumLit($1) }
  | STR_LIT    { StringLit($1) }
  | TRUE       { BoolLit(true) }
  | FALSE      { BoolLit(false) }
  | ID         { Val($1) }
  | NONE       { VoidLit }
  | LSQUARE actuals_opt RSQUARE  { ListLit($2) }
  | LBRACE kv_pairs RBRACE       { DictLit($2) }
  | fun_literal   { $1 }

expr:
  literals              { $1 }
  | if_expr             { $1 }
  | expr PLUS expr      { Binop($1, Add, $3) }
  | expr MINUS expr     { Binop($1, Sub, $3) }
  | expr MULTIPLY expr  { Binop($1, Mult, $3) }
  | expr DIVIDE expr    { Binop($1, Div, $3) }
  | expr MODULUS expr   { Binop($1, Mod, $3) }
  | expr EQ expr        { Binop($1, Eq, $3) }
  | expr NEQ expr       { Binop($1, Neq, $3) }
  | expr GT expr        { Binop($1, Gt, $3) }
  | expr GTE expr       { Binop($1, Gte, $3) }
  | expr LT expr        { Binop($1, Lt, $3) }
  | expr LTE expr       { Binop($1, Lte, $3) }
  | expr AND expr       { Binop($1, And, $3) }
  | expr OR expr        { Binop($1, Or, $3) }
  | expr CONS expr      { Binop($1, Cons, $3) }
  | expr APPEND expr    { Binop($1, Append, $3) }
  | expr CARET expr     { Binop($1, Caret, $3) }
  | MINUS expr %prec NEG  { Unop(Neg, $2) }
  | NOT expr            { Unop(Not, $2) }
  | ID ASSIGN expr      { Assign($1, $3) }
  | LPAREN expr RPAREN  { $2 }
  | ID LPAREN actuals_opt RPAREN  { Call(Val($1), $3) }
  | ID LSQUARE expr RSQUARE       { Element($1, $3) }
```

utils.ml

```
open Ast

module CharMap = Map.Make(String)

(* Functions to print AST *)
let string_of_op = function
    Add       -> "+"
  | Mult      -> "*"
  | Sub       -> "-"
  | Div       -> "/"
  | Mod       -> "%"
  | Caret     -> "^"
  | Append    -> "@"
  | And       -> "&&"
  | Or        -> "||"
  | Lte       -> "<="
  | Gte       -> ">="
  | Neq       -> "!="
  | Eq        -> "=="
  | Lt        -> "<"
  | Gt        -> ">"
  | Cons      -> "::"
;;
```

```
let string_of_uop = function
    Not       -> "!"
  | Neg       -> "-"
;;

let rec string_of_expr = function
    VoidLit               -> "VOID"
  | NoExpr                -> "\n"
  | NumLit(n)             -> string_of_float n
  | StringLit(s)          -> s
  | BoolLit(true)         -> "True"
  | BoolLit(false)        -> "False"
  | ListLit(l)            -> "[" ^ String.concat ", " (List.map string_of_expr l) ^
"]"
  | DictLit(l)            -> let pairs = List.map (fun (k, v) -> string_of_expr k ^
": " ^ string_of_expr v) l in "{" ^ (String.concat ", " pairs) ^ "}"
  | FunLit(args, s)       -> "lambda(" ^ String.concat ", " args ^ "):\n" ^
string_of_expr s
  | Val(s)                -> s
  | Binop(e1, op, e2)     -> string_of_expr e1 ^ " " ^ string_of_op op ^ " " ^
string_of_expr e2
  | Unop(op, e)           -> string_of_uop op ^ " " ^ string_of_expr e
  | Assign(v, e)          -> v ^ " = " ^ string_of_expr e
  | Call(Val(f), args)    -> f ^ "(" ^ String.concat ", " (List.map string_of_expr
args) ^ ")"
  | Call(_, args)         -> "==== THIS SHOULD NEVER HAPPEN ===="
  | Element(s, e)         -> s ^ "[" ^ string_of_expr e ^ "]"
  | Block(es)             -> String.concat "\n" (List.map string_of_expr es) ^
"\nend\n"
  | If(esl, s)            -> "if " ^ string_of_expr (fst(List.hd esl)) ^ ":\n" ^
                             string_of_expr (snd(List.hd esl)) ^
                             String.concat "\n" (List.map string_of_elif (List.tl
esl)) ^
                             "else:\n" ^ string_of_expr s
and string_of_elif el =
    "elif " ^ string_of_expr (fst el) ^ ":\n" ^ string_of_expr (snd el)
;;

let string_of_program prog =
             String.concat "\n" (List.map string_of_expr prog)
;;

let string_of_type t =
  let rec helper t chr map =
    match t with
      TNum    -> "num", chr, map
    | TString -> "string", chr, map
    | TBool   -> "bool", chr, map
    | TAny    -> "any", chr, map
    | TVoid   -> "void", chr, map
    | T(x)    ->
      let gen_chr, new_chr, new_map = if CharMap.mem x map
        then Char.escaped (Char.chr (CharMap.find x map)), chr, map
        else
          let c = Char.escaped (Char.chr chr) in
          c, (chr + 1), CharMap.add x chr map
      in
      Printf.sprintf "%s" gen_chr, new_chr, new_map
    | TList(t) ->
      let st, c, m = helper t chr map in
      (Printf.sprintf "list %s" st), c, m
    | TDict(kt, vt) ->
      let st1, c1, m1 = helper kt chr map in
      let st2, c2, m2 = helper vt c1 m1 in
      (Printf.sprintf "<%s:%s>" st1 st2), c2, m2
    | TFun(args_type, ret_type) ->
      let fold_func acc arg =
        let (c, m) = snd(acc) in
        let argt, c1, m1 = helper arg c m in
        (fst(acc) @ [argt], (c1, m1)) in
      let sargs, (c, m) = List.fold_left fold_func ([], (chr, map)) args_type in
      let rs, c, m = helper ret_type c m in
      let sargs = String.concat ", " sargs in
      Printf.sprintf "(%s) -> %s" sargs rs, c, m
  in
  let s, _, _ = helper t 65 CharMap.empty in s
;;

let rec string_of_aexpr ae =
  match ae with
    ABoolLit(b, t)          -> Printf.sprintf "(%s: %s)" (string_of_bool b)
(string_of_type t)
  | ANumLit(x, t)           -> Printf.sprintf "(%s: %s)" (string_of_float x)
(string_of_type t)
  | AStringLit(s, t)        -> Printf.sprintf "(%s: %s)" s (string_of_type t)
  | AVoidLit(t)             -> Printf.sprintf "(VOID)"
  | AVal(s, t)              -> Printf.sprintf "(%s: %s)" s (string_of_type t)
  | ABinop(e1, op, e2, t)   -> Printf.sprintf "(%s %s %s: %s)" (string_of_aexpr e1)
(string_of_op op) (string_of_aexpr e2) (string_of_type t)
  | AUnop(op, e, t)         -> Printf.sprintf "(%s %s: %s)" (string_of_aexpr e)
(string_of_uop op) (string_of_type t)
  | AAssign(id, t1, e, t2)  -> Printf.sprintf "%s: %s = %s : %s" id (string_of_type
t1) (string_of_aexpr e) (string_of_type t2)
```

```
  | AListLit(aes, t)        -> Printf.sprintf "[%s]:%s" (String.concat ","
(List.map string_of_aexpr aes)) (string_of_type t)
  | ADictLit(kvpairs, t)    -> Printf.sprintf "dict < >: %s" (string_of_type t)
  | AIf(apairs, ae, t)      -> Printf.sprintf "if {} elif {} else {}: %s"
(string_of_type t)
  | ABlock(aes, t)          -> Printf.sprintf "{ %s }" (String.concat "\n"
(List.map string_of_aexpr aes))
  | ACall(afn, aargs, t)    -> Printf.sprintf "%s(%s) : %s" (string_of_aexpr afn)
(String.concat "," (List.map string_of_aexpr aargs)) (string_of_type t)
  | AElement(id, ae, t)     -> Printf.sprintf "%s[%s] : %s" id (string_of_aexpr ae)
(string_of_type t)
  | AFunLit(ids, body, t)   -> begin
      let args_with_types, ret_type = (match t with
            TFun(args_type, ret_type) -> List.combine ids args_type, ret_type
          | _ -> raise (Failure("not a valid function"))) in
      let fargs = String.concat ", " (List.map (fun (id, typ) -> id ^ " : " ^
string_of_type typ) args_with_types) in
      let fsig = "(" ^ fargs ^ ")" ^ " : " ^ (string_of_type ret_type) in
      let fbody = string_of_aexpr body in
      String.concat " " ["lambda"; fsig; "="; "{"; fbody; "}"]
    end
;;

let string_of_aprogram aprog =
    String.concat "\n" (List.map string_of_aexpr aprog)
;;
```

library.ml

```
open Ast

module NameMap = Map.Make(String);;

let library_functions = [
  ("print", TFun([T("A")], TVoid));
  ("print_str", TFun([TString], TVoid));
  ("print_num", TFun([TNum], TVoid));
  ("print_bool", TFun([TBool], TVoid));
  ("print_list", TFun([TList(T("AI"))], TVoid));
  ("print_dict", TFun([TDict(T("AJ"), T("AK"))], TVoid));
  ("to_str", TFun([TNum], TString));
  ("hd", TFun([TList(T("B"))], T("B")));
  ("empty", TFun([TList(T("C"))], TBool));
  ("tl", TFun([TList(T("D"))], TList(T("D"))));
  ("len", TFun([TList(T("E"))], TNum));
  ("filter", TFun([TFun([T("F")],TBool); TList(T("F"));], TList(T("F"))));
  ("map", TFun([TFun([T("G")],T("H")); TList(T("G"));], TList(T("H"))));
  ("foldl", TFun([TFun([T("I"); T("J")], T("I")); T("I"); TList(T("J"))], T("I")));
  ("contains", TFun([TDict(T("K"), T("L")); T("K")], TBool));
  ("del", TFun([TDict(T("M"), T("N")); T("M");], TDict(T("M"), T("N"))));
  ("keys", TFun([TDict(T("O"), T("P"));], TList(T("O"))));
  ("count", TFun([TDict(T("Q"), T("R"))], TNum));
];;

let built_in_functions = List.fold_left
    (fun acc (id, t) -> NameMap.add id t acc)
    NameMap.empty library_functions
;;
```

spy.ml

```
open Ast
open Lexing
open Parsing
open Core.Std
open Codegen

let inchan = if Array.length Sys.argv > 1
             then In_channel.create ~binary:true Sys.argv.(1)
             else In_channel.stdin
;;

(*
let _ =
    let lexbuf = Lexing.from_channel inchan in
    let program = Parser.program Scanner.token lexbuf in
    print_endline (Utils.string_of_program program)
;;
*)

(*
let _ =
    let lexbuf = Lexing.from_channel inchan in
    let program = Parser.program Scanner.token lexbuf in
    let inferred_program = try Typechecker.type_check program
    with e -> Printf.printf "Error: %s\n" (Exn.to_string e); exit 1 in
    print_endline (Utils.string_of_aprogram inferred_program)
;;
*)
```

```
let _ =
    let lexbuf = Lexing.from_channel inchan in
    let program = Parser.program Scanner.token lexbuf in
    let inferred_program = try Typechecker.type_check program
    with e -> Printf.printf "Error: %s\n" (Exn.to_string e); exit 1 in
    let s_prog = try Codegen.js_of_aprog inferred_program
      with e -> Printf.printf "Error: %s\n" (Exn.to_string e); exit 1 in
    let outc = Out_channel.create "out.js" in
    Printf.fprintf outc "%s" s_prog; Out_channel.close outc
;;
```

typechecker.ml

```
open Ast
open Lexing
open Parsing
open Utils

module NameMap = Map.Make(String)
module GenericMap = Map.Make(Char)
module KeywordsSet = Set.Make(String)

let type_variable = (ref ['A'; 'A'; 'A']);;

let js_keywords = ["break"; "case"; "class"; "catch"; "const"; "continue";
                   "debugger"; "default"; "delete"; "do"; "else";
                   "export"; "extends"; "finally"; "for"; "function"; "if";
                   "import"; "in"; "instanceof"; "new"; "return"; "super";
                   "switch"; "this"; "throw"; "try"; "typeof"; "var"; "void";
                   "while"; "with"; "yield"; "val"; ];;

let spy_lib = ["print"; "print_num"; "print_str"; "print_bool";
               "print_list"; "print_map"; "to_str"; "hd"; "empty";
               "tl"; "contains"; "del"; "keys"; "len"; "filter"; "map";
               "foldl"; "count"; ];;

let all_keywords = List.fold_left (fun acc x -> KeywordsSet.add x acc)
    KeywordsSet.empty (spy_lib @ js_keywords);;

let spy_keywords = List.fold_left (fun acc x -> KeywordsSet.add x acc)
    KeywordsSet.empty spy_lib;;

let get_new_type () =
  let rec helper cs =
    let ys = match List.rev cs with
```

```
        [] -> ['A']
      | 'Z' :: xs -> 'A' :: List.rev (helper (List.rev xs))
      | x :: xs -> (Char.chr ((Char.code x) + 1)) :: xs
    in List.rev ys
  in
  let curr_type_var = !type_variable in
  type_variable := helper (curr_type_var);
  T(String.concat "" (List.map Char.escaped curr_type_var))
;;

let merge_env env =
  let locals, globals = env in
  let merged_globals = NameMap.merge (fun k k1 k2 -> match k1, k2 with
        Some k1, Some k2 -> Some k1
      | None, k2 -> k2
      | k1, None -> k1)
      locals globals in
  NameMap.empty, merged_globals
;;

let rec annotate_expr e env =
  match e with
    VoidLit       -> AVoidLit(TVoid), env
  | NumLit(n)     -> ANumLit(n, TNum), env
  | BoolLit(b)    -> ABoolLit(b, TBool), env
  | StringLit(s)  -> AStringLit(s, TString), env

  | Binop(e1, op, e2) ->
    let ae1 = fst(annotate_expr e1 env)
    and ae2 = fst(annotate_expr e2 env)
    and new_type = get_new_type () in
    ABinop(ae1, op, ae2, new_type), env

  | Unop(op, e) ->
    let ae = fst(annotate_expr e env)
    and new_type = get_new_type () in
    AUnop(op, ae, new_type), env

  | Val(id) ->
    let locals, globals = env in
    let typ = if NameMap.mem id locals
              then NameMap.find id locals
              else if NameMap.mem id globals
              then NameMap.find id globals
              else raise (Failure("Error: value " ^ id ^ " was used before it was
defined")) in
    AVal(id, typ), env
  | FunLit(formals, e) -> begin
      List.iter (fun k -> if KeywordsSet.mem k all_keywords
                          then raise (Failure("Error: Cannot redefine keyword " ^
k))
                          else ()) formals;
      let annotated_args = List.map (fun formal -> (formal, get_new_type ()))
formals in
      let locals, globals = merge_env env in
      let new_locals = List.fold_left (fun map (it, at) -> if NameMap.mem it map
                                                           then raise
(Failure("Error: " ^ it ^ " cannot be redefined in the current scope"))
                                                           else NameMap.add it at
map) locals annotated_args in
      let new_env = (new_locals, globals) in
      let ae, _ = (match e with
            Block(es) -> begin
                let aes, _ = List.fold_left (fun (aes, env) e -> let ae, env =
annotate_expr e env in (ae :: aes, env)) ([], new_env) es
                in ABlock(List.rev aes, get_new_type()), new_env
              end
            | _ -> raise (Failure("Unreachable state: FunLit"))) in
      let arg_types = List.map snd annotated_args in
      AFunLit(formals, ae, TFun(arg_types, get_new_type ())), env
    end

  | Call(fn, args) -> begin
      let afn, _  = annotate_expr fn env in
      let aargs = List.map (fun arg -> fst (annotate_expr arg env)) args in
      ACall(afn, aargs, get_new_type ()), env
    end

  | Assign(id, e) -> begin
      let locals, globals = env in
      if NameMap.mem id locals
      then raise (Failure("Error: " ^ id ^ " cannot be redefined in the current
scope"))
      else if KeywordsSet.mem id all_keywords
      then raise (Failure("Error: Cannot redefine keyword " ^ id))
      else let t = get_new_type () in
        let new_locals = NameMap.add id t locals in
        let ae, _ = match e with
            FunLit(_) -> annotate_expr e (new_locals, globals)
          | _ -> annotate_expr e env in
        AAssign(id, t, ae, TVoid), (new_locals, globals)
    end
  | ListLit(es) ->
    let aes = List.map (fun e -> fst (annotate_expr e env)) es in
    AListLit(aes, TList(get_new_type ())), env

  | DictLit(kvpairs) ->
    let apairs = List.map (fun (k, v) -> fst (annotate_expr k env), fst
(annotate_expr v env)) kvpairs in
    ADictLit(apairs, TDict(get_new_type (), get_new_type ())), env

  | If(pepairs, e) ->
    let apairs = List.map (fun (p, e) -> fst (annotate_expr p env), fst
(annotate_expr e env)) pepairs
    and ae = fst (annotate_expr e env) in
    AIf(apairs, ae, get_new_type ()), env

  | Block(es) -> begin
      let new_env = merge_env env in
      let aes, new_env = List.fold_left (fun (aes, env) e -> let ae, env =
annotate_expr e env in (ae :: aes, env)) ([], new_env) es in
      ABlock(List.rev aes, get_new_type ()), env
    end

  | Element(id, e) -> begin
      let locals, globals = env in
      let typ = if NameMap.mem id locals
                then NameMap.find id locals
                else if NameMap.mem id globals
                then NameMap.find id globals
                else raise (Failure("Error: value " ^ id ^ " was used before it was
defined")) in
      let t = (match typ with
          TList(t) -> t
        | TDict(t1, t2) -> t2
        | _ -> get_new_type ()) in
      let ae, _ = annotate_expr e env in
      AElement(id, ae, t), env
    end
  | _ -> raise (Failure("Error: unexpected expression found"))
;;

let type_of = function
    AVoidLit(t)           -> t
  | ANumLit(_, t)         -> t
  | ABoolLit(_, t)        -> t
  | AStringLit(_, t)      -> t
  | ABinop(_, _, _, t)    -> t
  | AUnop(_, _, t)        -> t
  | AListLit(_, t)        -> t
  | ADictLit(_, t)        -> t
  | ABlock(_, t)          -> t
  | AAssign(_, _, _, t)   -> t
  | AVal(_, t)            -> t
  | AIf(_, _, t)          -> t
  | ACall(_, _, t)        -> t
  | AFunLit(_, _, t)      -> t
  | AElement(_, _, t)     -> t
;;

let rec get_constraints ae =
  match ae with
    AVoidLit(_) | ANumLit(_) | ABoolLit(_) | AStringLit(_) | AVal(_) -> []

  | ABinop(ae1, op, ae2, t) ->
    let et1 = type_of ae1 and et2 = type_of ae2 in
    let opc = match op with
        Add | Sub | Mult | Div | Mod  -> [(et1, TNum); (et2, TNum); (t, TNum)]
      | Caret                         -> [(et1, TString); (et2, TString); (t,
TString)]
      | And | Or                      -> [(et1, TBool); (et2, TBool); (t, TBool)]
      | Lt | Lte | Gt | Gte           -> [(et1, et2); (t, TBool)]
      | Eq | Neq                      -> [(et1, et2); (t, TBool)]
      | Cons -> (match et2 with
            TList(x) -> [(et1, x); (t, et2)]
          | T(_)     -> [(et2, TList(et1)); (t, TList(et1))]
          | _ -> raise (Failure("Type error: Lists can only contain one type")))
      | Append -> (match et1, et2 with
            TList(_), TList(_) -> [(et1, et2); (t, et2)]
          | T(_), TList(_)     -> [(et1, et2); (t, et2)]
          | TList(_), T(_)     -> [(et2, et1); (t, et1)]
          | _, _ -> raise (Failure("Type error: Lists can only contain one type")))
in
    (get_constraints ae1) @ (get_constraints ae2) @ opc

  | AUnop(op, ae, t) ->
    let et = type_of ae in
    let opc = (match op with
          Not -> [(et, TBool); (t, TBool)]
        | Neg -> [(et, TNum); (t, TNum)]) in
    (get_constraints ae) @ opc

  | AIf(apairs, ae, t) ->
    let plist = List.map fst apairs in
    let elist = List.map snd apairs in
    let p_conts = List.map (fun p -> (type_of p, TBool)) plist in
    let e_conts = List.map (fun e -> (type_of e, type_of ae)) elist in
    (t, type_of ae) :: (List.flatten (List.map get_constraints plist)) @
(List.flatten (List.map get_constraints elist)) @ (get_constraints ae) @ p_conts @
e_conts

  | AListLit(aes, t) ->
    let list_type = match t with
        TList(x) -> x
      | _ -> raise (Failure("Unreachable state: ListLit")) in
    let elem_conts = List.map (fun ae -> (list_type, type_of ae)) aes in
    (List.flatten (List.map get_constraints aes)) @ elem_conts

  | ADictLit(kvpairs, t) ->
    let kt, vt = match t with
        TDict(kt, vt) -> kt, vt
      | _ -> raise (Failure("Unreachable state: DictLit")) in
    let klist = List.map fst kvpairs in
    let vlist = List.map snd kvpairs in
    let k_conts = List.map (fun k -> (kt, type_of k)) klist in
    let v_conts = List.map (fun v -> (vt, type_of v)) vlist in
    [(t, TDict(kt, vt))] @ (List.flatten (List.map get_constraints klist)) @
(List.flatten (List.map get_constraints vlist)) @ k_conts @ v_conts

  | ABlock(aes, t) ->
    let last_type = (match List.hd (List.rev aes) with
          AAssign(id, _, _, _) -> raise (Failure("Type error: Assignment
expressions cannot be returned"))
        | ae -> type_of ae) in
    (t, last_type) :: (List.flatten (List.map get_constraints aes))

  | AAssign(_, t, ae, _) -> (t, type_of ae) :: (get_constraints ae)

  | AFunLit(_, ae, t) -> (match t with
        TFun(_, ret_type) -> (type_of ae, ret_type) :: (get_constraints ae)
      | _ -> raise (Failure("Unreachable state: FunLit")))

  | ACall(afn, aargs, t) ->
    let typ_afn = (match afn with
          AVal(_) -> type_of afn
        | _ -> raise (Failure("Unreachable state: Call"))) in
    let sign_conts = (match typ_afn with
          TFun(arg_types, ret_type) -> begin
            let l1 = List.length aargs and l2 = List.length arg_types in
            if l1 <> l2
            then raise (Failure("Error: Expected number of argument(s): " ^
(string_of_int l2) ^ ", got " ^ (string_of_int l1) ^ " instead."))
            else let arg_conts = List.map2 (fun ft at -> (type_of at, ft))
```

```ocaml
arg_types aargs in
            (ret_type, t) :: arg_conts
          end
        | T(_) -> [(typ_afn, TFun(List.map type_of aargs, t))]
        | _ -> let st = Utils.string_of_type typ_afn in
               let ft = TFun(List.map type_of aargs, t) in
               let sft = Utils.string_of_type ft in
               raise (Failure("Type error: expected value of type " ^ st ^ ", got
type " ^ sft))) in
    (get_constraints afn) @ (List.flatten (List.map get_constraints aargs)) @
sign_conts
  | AElement(_, ae, t) -> (get_constraints ae)
;;

let rec substitute pt ch typ =
  match typ with
    TNum | TBool | TString | TVoid | TAny -> typ
  | T(c) -> if c = ch then pt else typ
  | TFun(t1, t2) -> TFun(List.map (substitute pt ch) t1, substitute pt ch t2)
  | TList(t) -> TList(substitute pt ch t)
  | TDict(kt, vt) -> TDict(substitute pt ch kt, substitute pt ch vt)
;;

let apply subs t =
    List.fold_right (fun (ch, pt) typ -> substitute pt ch typ) subs t
;;

let rec resolve_type ch t =
  match t with
    T(c) when ch = c -> false
  | TList(tlist) -> resolve_type ch tlist
  | TDict(kt, vt) -> (resolve_type ch kt) && (resolve_type ch vt)
  | TFun(argst, ret_t) -> let res = List.map (resolve_type ch) (ret_t :: argst) in
List.fold_left ( && ) true res
  | _ -> true
;;

let rec unify = function
    [] -> []
  | (t1, t2) :: cs ->
    let subs2 = unify cs in
    let subs1 = unify_one (apply subs2 t1) (apply subs2 t2) in
    subs1 @ subs2

and unify_one t1 t2 =
  match t1, t2 with
    TNum, TNum | TBool, TBool | TString, TString | TVoid, TVoid -> []
  | T(x), z | z, T(x) ->
      if (t1 = t2 || resolve_type x z)
      then [(x, z)]
      else raise (Failure("Type error: expected value of type " ^
(Utils.string_of_type t2) ^ ", got type " ^ (Utils.string_of_type t1)))
  | TList(t1), TList(t2) -> unify_one t1 t2
  | TDict(kt1, vt1), TDict(kt2, vt2) ->
          let _ = (match kt1 with
                  TNum | TBool | TString | T(_) -> ()
                | _ -> raise (Failure("Type Error: Cannot have key of type " ^
(Utils.string_of_type kt1) ^ " in a Dict."))) in
          unify [(kt1, kt2) ; (vt1, vt2)]
  | TFun(a, b), TFun(x, y) ->
      let l1 = List.length a and l2 = List.length x in
      if l1 = l2
      then unify ((List.combine a x) @ [(b, y)])
      else raise (Failure("Error: Expected number of argument(s): " ^
(string_of_int l1) ^ ", got " ^ (string_of_int l2) ^ " instead."))
  | _ -> raise (Failure("Type error: expected value of type " ^
(Utils.string_of_type t2) ^ ", got type " ^ (Utils.string_of_type t1)))
;;

let rec apply_expr subs ae =
  match ae with
    ABoolLit(b, t)            -> ABoolLit(b, apply subs t)
  | ANumLit(n, t)             -> ANumLit(n, apply subs t)
  | AStringLit(s, t)          -> AStringLit(s, apply subs t)
  | AVoidLit(t)               -> AVoidLit(apply subs t)
  | AVal(s, t)                -> AVal(s, apply subs t)
  | AAssign(id, t, ae, _)     -> AAssign(id, apply subs t, apply_expr subs ae,
TVoid)
  | ABinop(ae1, op, ae2, t)  -> ABinop(apply_expr subs ae1, op, apply_expr subs
ae2, apply subs t)
  | AUnop(op, ae, t)          -> AUnop(op, apply_expr subs ae, apply subs t)
  | AListLit(aes, t)          -> AListLit(List.map (apply_expr subs) aes, apply subs
t)
  | ADictLit(kvpairs, t)      -> ADictLit(List.map (fun (k, v) -> (apply_expr subs
k, apply_expr subs v)) kvpairs, apply subs t)
  | AIf(apairs, ae, t)        -> AIf(List.map (fun (p, e) -> (apply_expr subs p,
apply_expr subs e)) apairs, apply_expr subs ae, apply subs t)
  | ABlock(aes, t)            -> ABlock(List.map (apply_expr subs) aes, apply subs
t)
  | AFunLit(ids, ae, t)       -> AFunLit(ids, apply_expr subs ae, apply subs t)
  | ACall(afn, aargs, t)      -> ACall(apply_expr subs afn, List.map (apply_expr
subs) aargs, apply subs t)
  | AElement(id, ae, t)       -> AElement(id, apply_expr subs ae, apply subs t)
;;
```

```
let infer expr env =
  let aexpr, env = annotate_expr expr env in
  let constraints = get_constraints aexpr in
  let subs = unify constraints in
  let inferred_expr = apply_expr subs aexpr in
  inferred_expr, env
;;

let type_check program =
  let built_in_functions = Library.built_in_functions in
  let env = (NameMap.empty, built_in_functions) in
  let infer_helper (acc, env) expr =
    let inferred_expr, env = try infer expr env
                             with e -> raise e in
    let inferred_expr, env = (match inferred_expr with
        AAssign(id, t, ae, _) ->
          let subs = [] in
          let locals, globals = env and aet = type_of ae in
          let locals = NameMap.add id aet locals in
          let ret_ae = AAssign(id, apply subs t, ae, TVoid) in
          ret_ae, (locals, globals)
      | _ -> inferred_expr, env) in
    (inferred_expr :: acc, env) in
  let inferred_program, _ = List.fold_left infer_helper ([], env) program in
  List.rev inferred_program
;;
```

codegen.ml

```
open Ast
open Utils

module NameMap = Map.Make(String);;

let js_lib = "
  immutable.min.js code inserted here.
  not outputted in the report for convenience
  and readability.
" ;;

let merge_env env =
  let locals, globals = env in
  let merged_globals = NameMap.merge (fun k k1 k2 -> match k1, k2 with
        Some k1, Some k2 -> Some k1
      | None, k2 -> k2
```

```
       | k1, None -> k1)
       locals globals in
  NameMap.empty, merged_globals
;;

let get_random () = 1000 + (Random.int 1000)

let block_template ret opt_body = match opt_body with
    None -> Printf.sprintf "(function() { return %s; })()" ret
  | Some(exprs) -> Printf.sprintf  "(function() { %s; \n return %s; })()" exprs ret
;;

let if_template expr_pairs else_expr =
  let id = "res_" ^ string_of_int(get_random ()) in
  let elif_temp = format_of_string "  else if (%s) { %s = %s; } \n" in
  let if_pair = List.hd expr_pairs in
  let elif_pairs = List.tl expr_pairs in
  let str_elifs = List.fold_left (fun acc elif -> acc ^ (Printf.sprintf elif_temp
(fst elif) id (snd elif))) "" elif_pairs in
  "(function() { " ^
  "  let " ^ id ^ ";\n" ^
  "  if (" ^ (fst if_pair) ^ ") { " ^ id ^ " = " ^ (snd if_pair) ^ "; } \n" ^
  str_elifs ^
  "  else { " ^ id ^ " = " ^ else_expr ^ "; } \n" ^
  "return " ^ id ^ " })()"
;;

let rec js_of_aexpr aexpr env = match aexpr with
    ANumLit(x, _)     -> (Printf.sprintf "%s" (string_of_float x)), env
  | AStringLit(s, _)  -> (Printf.sprintf "\"%s\"" s), env
  | ABoolLit(b, _)    -> (Printf.sprintf "%s" (string_of_bool b)), env
  | AVoidLit(_)       -> "(undefined)", env
  | AUnop(o, e, _)    -> (Printf.sprintf "%s%s" (string_of_uop o) (fst (js_of_aexpr
e env))), env
  | ABinop(e1, o, e2, _) ->
    let s2, _ = js_of_aexpr e1 env
    and s3, _ = js_of_aexpr e2 env in (match o with
        Cons     -> (Printf.sprintf "(%s).insert(0, %s)" s3 s2)
      | Append   -> (Printf.sprintf "(%s).concat(%s)" s2 s3)
      | Caret    -> (Printf.sprintf "(%s + %s)" s2 s3)
      | Eq       -> (match (Typechecker.type_of e1) with
                        TList(_) | TDict(_) -> (Printf.sprintf "(Immutable.is(%s,
%s))" s2 s3)
                      | _ -> (Printf.sprintf "(%s %s %s)" s2 (string_of_op o) s3))
      | _        -> (Printf.sprintf "(%s %s %s)" s2 (string_of_op o) s3)), env

  | AVal(s, _) -> (match s with
        "print_num" | "print_bool" | "print" | "print_str" | "print_list" |
"print_dict" -> s, env
      | "to_str" | "hd" | "tl" | "empty" | "map" | "filter" | "foldl" | "len" -> s,
env
      | "contains" | "keys" | "del" | "count" -> s, env
      | _ ->  let locals, globals = env in
              if NameMap.mem s locals
              then (NameMap.find s locals), env
              else if NameMap.mem s globals
              then (NameMap.find s globals), env
              else raise (Failure("Error: not found in globals " ^ s)))

  | AAssign(id, _, expr, _) ->
      let locals, globals = env in
      let alias = id ^ string_of_int(get_random ()) in
      let new_locals = NameMap.add id alias locals in
      let new_env = new_locals, globals in
      (Printf.sprintf "var %s = %s" alias (fst (js_of_aexpr expr new_env))),
new_env

  | ABlock(es, _) ->
    let new_env = merge_env env in
    let fold_func acc e =
      let e1, env1 = (js_of_aexpr e (snd acc)) in
      (e1 :: (fst acc)), env1 in
    let es, _ = List.fold_left fold_func ([], new_env) es in
    (match es with
        []      -> ""
      | x :: [] -> block_template x None
      | x :: xs -> block_template x (Some (String.concat ";\n" (List.rev xs)))),
env

  | AIf(apairs, ae, _) ->
    let spairs = List.map (fun (p, e) -> ((fst (js_of_aexpr p env)), (fst
(js_of_aexpr e env)))) apairs in
    let se, _ = js_of_aexpr ae env in
    (if_template spairs se), env

  | AFunLit(formals, ae, _) ->
    let locals, globals = merge_env env in
    let aliases = List.map (fun formal -> formal ^ string_of_int(get_random ()))
formals in
    let new_locals = List.fold_left2 (fun acc formal alias -> NameMap.add formal
alias acc) locals formals aliases in
    let string_formals = String.concat "," aliases in
    let new_env = new_locals, globals in
    let string_ae, _ = (match ae with
          ABlock(aes, _) -> begin
            let fold_func acc e =
              let e1, env1 = (js_of_aexpr e (snd(acc))) in
              (e1 :: (fst acc)), env1 in
            let es, _ = List.fold_left fold_func ([], new_env) aes in
            (match es with
                [] -> ""
              | x :: [] -> block_template x None
              | x :: xs -> block_template x (Some (String.concat ";\n" (List.rev
xs)))), env
          end
        | _ -> raise (Failure("Unreachable state: FunLit"))) in
    (Printf.sprintf "(function(%s) { return (%s) })" string_formals string_ae), env

  | ACall(e, es, _) ->
    let id = match e with
        AVal(s, _) -> if Typechecker.KeywordsSet.mem s Typechecker.spy_keywords
                      then s
                      else (fst (js_of_aexpr e env))
      | _ -> raise (Failure("Error: not a function call")) in
    let es = List.map (fun e -> (fst (js_of_aexpr e env))) es in
    let arg1 = List.hd es in
    let fn_call = (match id with
        "print_num" | "print_bool" | "print" | "print_str" | "print_list" |
"print_dict" -> "console.log(" ^ (String.concat "," es) ^ ")"
      | "to_str"   -> "(" ^ arg1 ^ ").toString()"
      | "hd"       -> "(" ^ arg1 ^ ").get(0)"
      | "tl"       -> "(" ^ arg1 ^ ").delete(0)"
      | "empty"    -> "(" ^ arg1 ^ ").isEmpty()"
      | "len"      -> "(" ^ arg1 ^ ").count()"
      | "filter"   -> "(" ^ (List.nth es 1) ^ ").filter(" ^ arg1 ^ ")"
      | "foldl"   -> "(" ^ (List.nth es 2) ^ ").reduce(" ^ arg1 ^ ", " ^ (List.nth
es 1) ^ ")"
      | "map"      -> "(" ^ (List.nth es 1) ^ ").map(" ^ arg1 ^ ")"
      | "contains" -> "(" ^ arg1 ^ ").has(( " ^ (List.nth es 1) ^ ").toString())"
      | "count"    -> "(" ^ arg1 ^ ").count()"
      | "keys"     -> "Immutable.fromJS(Array.from((" ^ arg1 ^ ").keys()))"
      | "del"      -> "(" ^ arg1 ^ ").remove((" ^ (List.nth es 1) ^ ").toString())"
      | _ -> Printf.sprintf "%s(%s)" id (String.concat "," es)) in
    fn_call, env

  | AListLit(es, _) -> "Immutable.List.of(" ^ (String.concat ", " (List.map (fun e
-> (fst (js_of_aexpr e env))) es)) ^ ")" , env

  | ADictLit(kvpairs, _) ->
    let pairs = List.map (fun (k, v) -> (fst (js_of_aexpr k env)) ^ ":" ^ (fst
(js_of_aexpr v env))) kvpairs in
```

```
    "Immutable.Map({ " ^ (String.concat "," pairs) ^ " })", env

  | AElement(id, ae, _) ->
      let locals, globals = env in
      if NameMap.mem id locals
      then "(" ^ (NameMap.find id locals) ^ ").get((" ^ (fst (js_of_aexpr ae env))
^ ").toString())", env
      else if NameMap.mem id globals
      then "(" ^ (NameMap.find id globals) ^ ").get((" ^ (fst (js_of_aexpr ae env))
^ ").toString())", env
      else raise (Failure("Error: not found in globals " ^ id))
;;

let js_of_aprog prog =
  let js_exprs, _ = List.fold_left (fun (acc_js, acc_env) aexpr ->
    let e1, env1 = js_of_aexpr aexpr acc_env in
    (e1 :: acc_js), env1) ([], (NameMap.empty, NameMap.empty)) prog in
  let base = js_lib ^ "\n\n" in
  base ^ (String.concat ";\n" (List.rev js_exprs))
```

runtests.ml (test runner modified from JSJS open source code)

```
open Printf

let test_location = "test/"

type test_status = Pass | Fail
type color = Red | Green

let colorize msg c =
  let pad = match c with
      Red -> "31"
    | Green -> "32" in
  let template = format_of_string "
    \027[%sm%s" in
  printf template pad msg;
  flush stdout;
;;

let run_cmd cmd =
  let chan = Unix.open_process_in cmd in
  let res = ref ([] : string list) in
  let rec helper () =
    let line = input_line chan in
    res := line :: !res;
    helper () in
  try helper ()
  with End_of_file ->
    let status = Unix.close_process_in chan in
    let cmd_result = match status with
        Unix.WEXITED(c) -> if c == 0 then Pass else Fail
      | _ -> Fail in
    (List.rev !res, cmd_result)
;;

let diff_output lines filename =
  let dump_to_file lines fname =
    let oc = open_out fname in
    List.iter (fun line -> fprintf oc "%s\n" line) lines;
    close_out oc in
  let _ = dump_to_file lines "test_output.txt" in
  let cmd = sprintf "diff -B test_output.txt %s" filename in
  let diff_output, status = run_cmd cmd in
  begin
    match status with
      Pass -> None
    | Fail -> Some(String.concat "\n" diff_output)
  end
;;

let run_testcase fname =
  let test_type, test_name =
    match (Str.split (Str.regexp "-") fname) with
      "fail" :: x :: [] -> Fail, x
    | "pass" :: x :: [] -> Pass, x
    | _ -> raise (Failure ("Invalid file format - " ^ fname ^ ". Must have only one
'-'")) in

  let fpath = Filename.concat test_location fname in
  let cmd = sprintf "./spy.out %s" fpath in

  let output_filename = Str.replace_first (Str.regexp "spy") "txt" fname in
  let output_path = Filename.concat test_location output_filename in

  let cmd_output, status = run_cmd cmd in
  match test_type, status with
      Pass, Pass -> begin
        let node_output, status = run_cmd "node out.js" in
        (match diff_output node_output output_path with
            None -> colorize (sprintf "✓ " ^ fname) Green; Pass
          | Some(op) -> begin
              colorize (sprintf "✖ %s" fname) Red;
              colorize (sprintf "%s\n\n" op) Red;
            end; Fail)
      end

    | Fail, Fail ->
      (match diff_output cmd_output output_path with
          None -> colorize (sprintf "✓ %s" fname) Green; Pass
        | Some(op) -> begin
            colorize (sprintf "✖ %s" fname) Red;
            colorize (sprintf "%s\n\n" op) Red;
          end; Fail)

    | Pass, Fail -> begin
        colorize (sprintf "✖ %s" fname) Red;
        colorize "Expected test case to pass, but it failed" Red;
      end; Fail

    | Fail, Pass -> begin
        colorize (sprintf "✖ %s" fname) Red;
        colorize "Expected test case to fail, but it passed" Red;
      end; Fail
;;

let run testcases () =
  let total = List.length testcases in
  let passing = List.fold_left (fun acc t -> acc + (match run_testcase t with Pass
-> 1 | Fail -> 0)) 0 testcases in
  let template = format_of_string "\027[37m

    Test Summary
    -------------------------
    All testcases complete.
    Total Testcases : %d
    Total Passing   : %d
    Total Failed    : %d
      \n" in
  let failures = total - passing in
  Printf.printf template total passing failures;
  if failures = 0 then Pass else Fail
;;

(* returns a list of file names in a directory *)
let get_files dirname =
    let d = Unix.opendir dirname in
    let files = ref ([] : string list) in
    let rec helper () =
      let fname = Unix.readdir d in
      files := fname :: !files;
      helper () in
    try helper () with End_of_file -> Unix.closedir d; files
;;

let init () =
  let files = !(get_files test_location) in
  let testcases = List.filter
      (fun f ->
         try ignore (Str.search_forward (Str.regexp ".spy") f 0); true
         with Not_found -> false)
      files in
  match (run testcases ()) with
    Pass -> exit 0
  | Fail -> exit 1
;;

init ();
```

Makefile

```
spy:
    ocamlbuild -j 0 -lib str -r -use-ocamlfind -pkgs core -use-menhir spy.native
    @mv spy.native spy.out

.PHONY : clean
clean:
    ocamlbuild -clean

.PHONY: test
test:
    ocamlc -o run-tests.out str.cma unix.cma test/runtests.ml
    @rm test/*.cm*

.PHONY: run-test
run-test:
    @make
    @make test
    @./run-tests.out
    @rm test_output.txt
```

Appendix

This project would not have been possible without the help of online resources and open source code. The following references were invaluable in developing this project:

1. http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
2. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.Make.html
3. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html
4. http://jsjs-lang.org/
5. http://www.cs.columbia.edu/~sedwards/classes/2016/4115-spring/reports/Scolkam.pdf
6. https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system
7. https://en.wikipedia.org/wiki/Type_inference
8. https://github.com/prakhar1989/JSJS/blob/master/src/typecheck.ml
9. Various stack overflow posts