Superscript Final Report

Uday Singh | Samurdha Jayasinghe | Tommy Orok | Michelle Zheng | Yu Wang
(urs2102) | (sj2564) | (to2240) | (myz2103) | (yw2684)

1. Introduction

1.1 Welcome to Superscript

Superscript is a type-inferred Lisp with syntax, focused on rapid development, that compiles into Javascript. Superscript is a compiled language with static type inference, using the Lisp core, that allows the user to write robust, yet concise programs. It is heavily influenced by both Arc and Clojure, two new languages introduced to the Lisp community, and OCaml.

Unlike Javascript, the functional-first nature of Superscript encourages users to think more before they code. Using static type inference based on the Hindley-Milner algorithm, our compiler tells users when their functions break due to type inconsistency. Similar to OCaml, it discourages users from thinking in terms of objects, and encourages clean data transformations through the use of S-expressions, which may be written using either Lisp-like, or imperative syntax (refer to Section 8, for Syntax Modifications to the Lisp core). This, combined with the flexibility of the Lisp family of languages, where functions and data are equivalent, gives users of Superscript a level of power unavailable in languages like Javascript.

Superscript enables sharp programmers to think and express clear functional ideas in a language that is slowly becoming the backbone of the web.

1.2 Program Structure

Superscript programs are built with expressions, and a program can be viewed as a single expression or a list of expressions executed in sequence. In Superscript, functions are data and data are functions.

In the following examples, the indicated result is what the expression would evaluate to, in the executable JavaScript produced by the Superscript compiler.

```
; Function call that prints Hello, world!
> (prn “Hello, world!”);;
Hello, world!

; Addition function applied to all integers from 1 to 10
> (+ 1 2 3 4 5 6 7 8 9 10);;
```

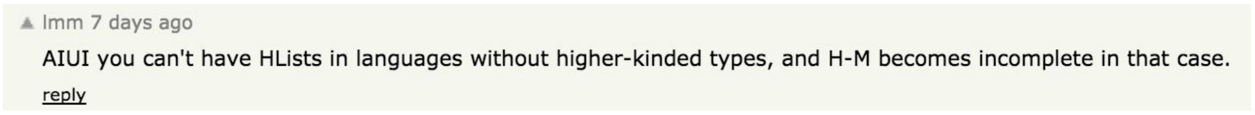
```
; Function call, applying the ‘head’ function to a list
; that consists of ‘+’ and the integers 1 through 10.
> (head ‘(+ 1 2 3 4 5 6 7 8 9 10));;
```
The idea that both functions and data are one and the same allows for rapid expression of ideas in Superscript that would otherwise be more difficult to express in Javascript. The above examples show not only how functions are executed, but how comments work (using the semicolon), and the power of code and data being one.

1.3 Related Work and Inspiration

There are actually a number of related works to Superscript, a lot of which we noticed when we were hunting for names - SLisp, LispyScript, Parenscript, and more. We did not actually consult any of them when designing Superscript, except Clojurescript. Clojurescript, however, is focused on covering many of the syntax ideas of Clojure in a Lisp which compiles to Javascript. Clojure involves many syntactic differences that Superscript does not have such as the notation for sets, maps, and vectors.

Our work is inspired by a Lisp called Arc developed by Paul Graham prior to Clojure as a response to the lack of Lisp development since the 1980s. We focused on Arc’s obsession of brevity (as in shortening the required amount of tokens to write a program, thus effectively decreasing the number of bugs). We tried not to fall into the Perl trap of requiring too few characters, but reducing the number of tokens so that programs remain transparent and short. We included an alternate syntax which preserves the semantics of Lisp, yet is very approachable for those who might be alarmed by the number of parentheses. Caramel can be preprocessed into Superscript which is then run by GEB (our compiler named after Hofstadter’s seminal work Gödel, Escher, Bach). Caramel allows Superscript to be more appealing to a more mainstream audience, switching out parentheses for indentation, as well as a few more syntactic features. Superscript still allows s-expressions as much like how John McCarthy worked on designing a syntax structure for Lisp, programmers preferred the untouched s-expressions.

We looked at Poly and Haskell for type inference to prevent classic type errors that plague Lisp, utilizing Hindley Milner’s Algorithm W. We strayed away from Arc’s function overloading as this would break Algorithm W and instead focused on implementing the best of the safety that comes from type inference with a mix between terse Lisp code which is readable and short that can be picked up easily by beginners to Lisp and safe enough to prevent many of the classic bugs that hurt Javascript. We also annoyed a lot of people on HackerNews as well, including someone who thought asking to have static type inference in a Lisp was just impossible, we hope we proved the wrong.


▲ lmm 7 days ago

AIUI you can't have HLists in languages without higher-kinded types, and H-M becomes incomplete in that case.

reply

Sorry llm, but we can.

We also utilized Node.js to evaluate Javascript on a cross-platform runtime environment so that Superscript could be extremely powerful for developing server-side web applications.

All in all, Superscript could not have developed without the open source community, and for that we are indebted.

2. Tutorial

This is a tutorial on Superscript that is for all users. Our Language Reference Manual is also designed like a tutorial, but this is more of a conversational attempt to get you into using Superscript. You do not have to know Lisp to walk through our tutorial. We will use the following syntax to cover code. When you see text in `this` typeface, it is in reference to command line functions.

Let's start by introducing the three characters important to understanding our tutorial. The `%` character is used to symbolize your Unix shell input. Make sure you have Node.js installed, if not, it can be downloaded from the Node.js site. In order to get started and print

```
% ./make
%  ./geb -s "(prn \"Hello, world!\");;"
```

This allows you to run individual lines of Superscript. Remember when using quotes to escape them in the command line, and to include the double semicolon for concluding statements.

```
% vim hello.ss
/* Type (prn "Hello, world!");; */
% ./geb hello.ss
```

If you want to run a program, you can use this method of compiling the file using geb, and you will see the output. We also used a `/* */` to symbolize a comment. This is how you comment as well in Superscript files. We will sometimes use the comment notation to provide extra information for the user.

Our second to last symbol we will introduce is the `>` which is short for any line of Superscript which can be run. Rather than writing for either compiling files, or running individual lines, we will use this character to display lines of Superscript code. Feel free to provide appropriate spacing for writing Superscript files or bringing everything into one line to run with the `./geb -s` command.

```
> (prn "Hello, world!");;
```

```
"Hello, world!"
```

Our last symbol is >> which stands for the return value of the expression. This will not be explicitly printed, but is useful to show for certain expressions. We will usually show the printed output below the returned value.

```
> (prn "Hello, world!");;
>> unit
"Hello, world!"
```

So let’s start with the classic “Hello, World!” program. Wait, we’ve actually done that before and we had no build up to it. Instead, we will try something different.

So here we will try it again, but instead do it with Devanagari script in Hindi. Let’s say hello, or better yet, “namaste” much like you do before a Yoga class.

```
> (prn "नमस्ते")
```

So that was terribly easy, and you’ll note that Superscript can print all Unicode characters allowed by Javascript. Essentially when you run when `hello.ss` has the line above in it:

```
% ./geb hello.ss
```

This creates a file called `a.js` which is then run by the Node.js runtime. If Javascript can do it, we can do it too. You can run the file for the same output by running:

```
% node a.js
```

You will notice a number of functions and respective types printed above the output, this is Superscript showing you the types of all available functions as it is run. You can see what functions are available when and their types, and the process of Algorithm W; this is by design to assist with debugging.

Superscript programs are built out of expressions such as integers, floats, strings, and booleans.

```
> 25;;
>> 25

> "foo";;
>> "foo"

> 24.4;;
```

```
>> 24.4

> true;;
>> true
```

Most expressions enclosed within multiple parentheses are also an expression. Many expressions together within parentheses are also known as an expression or actually, an s-expression, we also call these lists.

```
> (+ 2 3);;

>> 5
```

There are two types of lists. Quoted and unquoted lists. We understand this doesn't mean much right now, but all will be clear very soon.

```
/* Unquoted lists */
(+ 2 3)

/* Quoted lists */
'(+ 2 3)
```

The list above, or the unquoted list, is technically a function or is evaluated from left to right with the values of the tail being passed as arguments to the values of the head. The first, unquoted list returns 5. The second quoted list is a list with a +, a 2, and a 3 in it. In other words, lists with quotes are not evaluated, lists with quotes are evaluated as function calls. All s-expressions will return something. So what does this return?

```
> (+ (* 1 3) (+ 3 (+ 4 5)));;
```

If you said, 15, you would be correct, but as you are probably noticing, Superscript is not printing your values. So let's see what's happening behind the madness.

In order to print something, you have to use the `prn` function, however, `prn` requires a string for it to produce a string. This is an intentional strength of Superscript as the static type inference of Superscript requires functions to have the appropriate types for it to run a successful type check of the expression. The `prn` function requires a string and outputs a string. In order to do this we can use one of our many cast functions.

```
> (string_of_int (+ (* 1 3) (+ 3 (+ 4 5))));;
>> "15"

> (prn (string_of_int (+ (* 1 3) (+ 3 (+ 4 5)))));;
```

```
>> "15"
15
```

All casts are listed in the Language Reference Manual and will be used below. The format is pretty straight forward, string_of_int, string_of_float, string_of_boolean, and more.

So to go back to the expression, we notice that prefix notation, or putting the + before the elements in the unquoted list may seem a little odd compared to standard infix notation (if you want standard infix notation, we offer that. See the Infix Expression section in the Language Reference Manual or just put the operation within curly brackets. Prefix notation, however, has its benefits as we can keep adding arguments to the function.

```
> (+);;
>> 0

> (+ 1);;
>> 1

> (+ 1 1);;
>> 2

> (+ 1 1 1);;
>> 3
```

If we want to assign to foo the value of 42, the entire assignment expre will return 42 when evaluated.

```
> (= foo 42);;
>> 42

> (prn (string_of_int 42));;
>> 42
42
```

Although most operators evaluate from left to right, this is an exception which stores the value of 42 into foo. Now let’s add 42 to an unquoted list.

```
> (cons 4 '(8 15 16 23 42));;
>> ‘(4 8 15 16 23 42)
```

The `cons` function appends the first argument to the list in the second argument. You’ll note that we are using a quoted list as a data structure inside, and an unquoted list as a function expression. This allows Superscript to be a homoiconic language and allows the user to very clearly note the applicative order of Superscript and even compute Superscript expressions by hand similar to lambda calculus.

Similarly to `cons`, we have have two functions to take lists apart: `head` and `tail.` Although more traditional Lisps use the terms `car` and `cdr` respectively. We found that `head` and `tail` made more sense to new users as "Contents of the Address part of Register number" did not have the same ring as "head". To use these functions try this:

```
> (head '(4 8 15 16 23 42));;
>> 4

> (prn (string_of_int (int (head '(4 8 15 16 23 42)))));;
4

> (tail '(4 8 15 16 23 42));;
>> '(8 15 16 23 42)


> (print_list format_int (tail '(4 8 15 16 23 42))));;
[8,15,16,23,42]
```

You'll note that the standard Lisp formatting gives us 4 right parentheses at the end of this list may seem cumbersome. How is this dealt with in Lisp? As Paul Graham says, "we don't." Lisp programmers don't count parentheses and let their editors do the work for them. As for readability, we use indentation. If you write in Caramel, our preprocessor allows you to skip parentheses for indentation. Use the print_list function with format_int (or format_string or format_float, you get the idea), to print a list.

```
> (= x '(4 8 15 16 23 42));;
> (= y (tail x));;
> (= z y);;
> (print_list format_int (tail z));;
[15,16,23,42]
```

By allowing heterogeneous lists, Superscript allows exploratory programming with the strength of static type inference. All lists are lists of type SomeList of SomeType. In order to use the head of a list (or any individual element) in another computation, the result of calling head on the list must be annotated with a specific type (or dynamic cast) in order for type inference to work. In the below example, the result of head must be annotated to be an "int" in order to be passed to string_of_int, and to print the resulting string.

```
> (prn (string_of_int (int (head '(4 8 15 16 23 42)))));;
4
```

So the last few things we will teach you how to go through before we send you off to the language reference manual are booleans and lists.

So let's try a simple `if` statement.

```
> (if (is 0 0) 1 2);;
>> 1

> (if (isnt 0 0) 1 2);;
>> 2
```

We can use `is` and `isnt` for returning a boolean value which is fed into the if statement, we can also use standard comparable functions as well (<, >, <=, >=). These equality and comparison operators take 2 arguments which must be of the same type, such as int and int, or string and string. We also have logical operators such as `and, or & not`.

Now before we send you off into the language reference manual, we want to cover the last thing. Throughout this tutorial, we have been introducing you to the function and the very simple syntax behind Superscript. Most of the elements you see in Superscript are functions, and as such are easy to create yourself. To create a function, much like Javascript, all you have to do is assign an anonymous function to an identifier, and then you can call the identifier to run the function.

```
> (= function_name (fn (arg1 arg2 … argn) (function_body));;

> /* Note the use of infix below */
> (= fib (fn (x) (if (is x 0) 0 (if (is x 1) 1 (+ (fib {x - 1}) (fib {x -2}
))))));;
> (prn (string_of_int (fib 8)));;
21
```

And there you have it, you can now define and create your own functions in Superscript and run them like the way you learned. We have all sorts of more cute tricks and goodies inside the Language Reference Manual, but don't want to inundate you now, and instead would love to see you start writing programs. The LRM is also written as a semi-tutorial, so don't hesitate to just go through it section by section.

Happy hacking! (John McCarthy is probably smiling)

3. Language Reference Manual

3.1 Introduction

This is a language reference manual based on the previous language reference manual we submitted, and updated to reflect the current state of Superscript at the time of submission plus the geb compiler.

3.2 Lexical Conventions

3.2.1 Reserved Keywords

Superscript has a set of reserved keywords, which cannot be used as identifiers. Superscript also comes with built-in functions which generate specific Javascript code when invoked. It also has a standard library, written in Superscript, which is automatically imported (concatenated into the beginning of any user-defined code).

Reserved Keywords	**Built-in functions**	**Standard Library Functions**
true, false, fn, if, eval, do	+, -, *, /, +., -., *., /., mod is, isnt, <, <=, >, >=, and, or, not, ++, do, eval, call, dot, module, cons, head, tail, pr, prn, int_of_string, string_of_float, float_of_string, string_of_boolean, boolean_of_string, string_of_int, int, float, boolean, string, list, type	identity, length, nth, first, second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth, last, map, fold_left, fold_right, filter, append, reverse, drop, take, intersperse, member, zipwith, zipwith3, zip, unzip, format_boolean, format_int, format_string, format_float, stringify_list, format_boolean2d, format_int2d, format_float2d, format_string2d, print_list

3.2.2 Punctuation

3.2.2.1 Comments

Comments in Superscript are C-style:

```
>/* this is a comment */
```

3.2.2.2 Double semicolon

A program in Superscript is a list of expressions, and the expressions are separated by double semicolons:

```
> ( prn "hello" );; ( prn "world" );;  ( prn "people");;
hello
world
people
```

3.2.2.3 Parentheses

Parentheses must enclose the representation of a list, which must be quoted if it is not intended to be a function call:

```
> '( 1 2 3 );;
>> ‘(1 2 3)
```

Parentheses must be used to wrap a function definition, to wrap its arguments, and to wrap the function body. We used the Scheme lambda construction but removed the keyword lambda and replaced it with `fn` similar to Arc to encourage constant use of lambda functions, similar to Javascript function assignment:

```
> (fn (x y) (+ x y));;
```

Parentheses must be used to wrap any function call, including a standard library function call or built-in operation.

> `(average 2 4);;` /*average is a function that returns the average value of its parameters*/
3

```
> (if true 1 2);;
1
```

All unquoted lists are interpreted as function calls. Hence, parentheses cannot be used to wrap an unquoted list that is not a function call. The following is a syntax error, and cannot be evaluated as you are trying to evaluate the operator 1 on the arguments 2 and 3:

```
> (1 2 3);;
Line:1 char:1..6: Syntax error. Function call on inappropriate object.
```

3.2.2.4 Curly Braces

Braces can be used to wrap an infix expression and to explicitly indicate the order of operations and associativity. See Section 3.8 for more information about syntactic sugar and Syntax

Modifications that allow a more imperative programming syntax. Infix expressions can be used in both Superscript and Caramel syntax. See the section on Infix Expressions below.

```
> {1 + 2};;
>> 3
```

3.3. Data Types

3.3.1. Type Inference

Superscript uses static type inference, based on the Hindley-Milner algorithm[12], to evaluate the type and value of any legal expression and to check that expressions satisfy the proper data type in all function calls. Our compiler will flag any inconsistent data type in function calls as a type incompatibility error. Based on this type inference, the Superscript compiler implements error handling and prints the appropriate error messages to the user.

Superscript's standard library provides a set of functions to let user convert between different data types.

3.3.2. Atomic Data Types

The following are atomic (non-list) data types in Superscript.

3.3.2.1 Numerical types

We use type inference to avoid NaN errors common in Javascript, since non-numerical values cannot be used in arithmetic functions.

Integers

Superscript allows integers, which may be manipulated by integer arithmetic using the standard +, -, /, * operators. Integers are sequences of digits containing no decimals. Integers are considered accurate up to 15 digits.

```
> 42;;
>> 42

> (type 42);;
>> ‘int’
```

Floating Points

Superscript allows floating point values and requires the use of floating point specific operators to perform floating point arithmetic. These operators are the int operators followed by '.' (+., -., /., *.). Floating points must include at least one digit and a decimal, satisfying the regular expression: ['0'-'9']*'.'['0'-'9']+ | ['0'-'9']+'.'['0'-'9']*

```
> 42.0;;
42.0

> (type 42.0);;
'float'
```

3.3.2.3. String

Superscript supports UTF-8 strings as a way to represent textual data. Like Javascript, a single character is treated as a single character String. There is no type for chars. Print always prints to console and returns the unit datatype.

```
> "Hello, world!";;
>> "Hello, world!"

> (type "Hello, world!");;
>> 'string'
```

3.3.2.4. Symbols

Symbols, or identifiers, represent programmer-defined objects. They are containers for storing data values, and they return the value assigned to them. An identifier name must start with a letter, followed by any number of letters, digits, or underscores, and is defined by the regular expression: ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*

```
> (= a 42);;
42

> a;;
42
```

3.3.2.5. True/False/Nil

Superscript has a boolean true and false value.

```
> (type true);;
```

```
‘boolean’

> (type false);;
‘boolean’
```

Superscript also has a nil value which is the null datatype. It is equivalent to the empty list.

```
> nil;;
nil

> ‘();;
nil
```

3.3.3. Non-atomic Data Types

3.3.3.1. Lists

Multiple atoms enclosed parentheses are also called lists. An unquoted list is a function call, where the first element is the function name, and the other values are the parameters (See 3.3.2). For instance, *(a b c d e)* calls the function ***a*** with the arguments ***b***, ***c***, ***d***, and ***e***. It is a syntax error to write an expression that is an unquoted list, such as (5 4 8), where the first element is not a function name.

In order to use (a b c d e) as a list, rather than function call, you must quote the list: ‘(a b c d e) is a list of the elements a, b, c, d, and e.

```
> ‘(1 2 3);;
‘(1 2 3)

> (type ‘(1 2 3));;
‘(sometype) list ’
```

3.3.3.2. Functions

Call a function using an unquoted list, where the first argument is the function name, the following arguments are the parameters passed into the function call, and all the parameters are passed in by value, in the format of (function_name arg1 arg2 arg3…):

```
> (+ 1 2 3 4);;
10
```

Define an anonymous function in the following way, using the ‘fn’ keyword: (fn (optional_args) expression), where optional_args is 0 or more formal arguments separated by spaces, and enclosed by parentheses; the body of the function is a single expression enclosed by parentheses; and the entire expression is surrounded by parentheses. A function definition returns a function data type. In Superscript, functions are a data type much like lists and atoms.

```
> (fn (x y) (/(+ x y) 2));;
- : int -> int -> int
```

We can bind an anonymous function declaration to a name using the ‘=’ function, which evaluates right-to-left. Don’t be scared by the prefix notation, as the same may be expressed using infix notation, covered in Section 8, Syntax Modifications.

Anonymous function declaration, which is then bound to the name ‘average’:

```
> (= average (fn (x y) (/ (+ x y) 2)));;
val average : int -> int -> int
```

Function call:

```
> (average 20 10);;
15
```

3.4. Operators and Built-in Functions

3.4.1. Basic Assignment

The ‘=’ operator takes an even number of arguments, and assigns to the first of each pair the value of the second of each pair. This is a basic assignment that sets the value of a to 5, c to 6, and d to 7:

```
> (= a 5 c 6 d 7);;
val a : int
val c : int
val d : int

> (= x 5);;
val x : int
```

Basic assignment is applicable to all datatypes.

3.4.2. Arithmetic Operators

Before we go into arithmetic operators, note that we will be using prefix notation here. Superscript allows infix notation as well under the Infix Expression section.

Superscript offers several Standard Library arithmetic functions. These include both integral and floating addition, subtraction, multiplication, division; as well as the modulo of two ints. These functions are used as follows.

Addition, Subtraction, Multiplication, Division

You can add at least 2 arguments, ***a*** to ***b***, or add an unlimited amount of arguments together. The following examples show function calls to arithmetic operations.

```
(+ a b)
(+. a b c d e f g h i j k l m n o p …)
```

Both of the above expressions are unquoted lists where the first element is a function call to the addition function, done on the other elements in the list. The first example uses the integer operator (+), and the second, the floating operator (+.). Addition, Subtraction, Multiplication, and Division can be applied to two or more arguments.

All arguments to arithmetic functions must be numerical. They will be evaluated left to right.

Modulo

You can call a modulus function between TWO integers ***a*** and ***b,*** this will return the remainder of a / b.

```
> (mod 5 6)
5
```

3.4.3. Boolean Operators

Superscript's Standard Library contains logical functions to evaluate boolean expressions.

3.4.3.1. Logical NOT Function

'not' is a Standard-Library function that takes one boolean expression as its argument, and negates the value of that expression.

```
> (not true);;
false

> (not false);;
true
```

3.4.3.2 Logical AND / OR

Superscript's Standard Library supports logical AND/OR functions, using 'and' and 'or' keywords. AND/OR must be used with two boolean expressions as arguments.

```
> (and true false);;
false

(or true true);;
> true
```

3.4.3.3 Equality and Inequality

The 'is' function is an equality comparison that may be applied on atomic constants: ints, floats, and strings. The 'isnt' function compares two ints, floats, or strings for inequality. The arguments must be of the same type, for instance, string and string, or int and int.

```
> (is “a” “b”);;
false

> (is 1 1);;
true

> (is 1 2);;
false

> (isnt “a” “b”);;
true
```

3.4.3.5 Relational Comparison operators

These relational comparison operators can be used for ints, floats, and strings. They take two expressions, which must be of the same type, as arguments:

Examples:
```
(> a b);; (< a b);; (>= a b);; (<= a b);;

> (> 5 4);;
true
```

3.3.4.4 String concatenation

String concatenation is done using the “++”' function. It operates on a list of strings and concatenates them all from left to right.

```
> (++ “hello” “ ” “world” “ superscript” “ is” “ here”);;
“hello world superscript is here”
```

3.3.4.5. prn/pr Function

Print always prints to console. Its type is string -> unit. Hence you must pass it one or more strings as argument.

```
> (prn “Hello, world” “!”);;
Hello, world!

> (prn (string_of_int 5));;
5

> (type (prn 5));;
- : string -> unit
```

3.3.4.6 Infix Expressions

Infix expressions may be used in Superscript if you enclose them in curly braces: {expression};; Example usage is below:

1. Basic arithmetic in infix expressions will be evaluated in the standard order of operations. To parenthesize within infix expressions, use curly braces:

```
            { 1 * {2 - 4} / 4 };;
```

2. To call a function with an infix expression as argument, use parentheses around the entire function call, as per standard Superscript function calling syntax:

```
            ( foo {1 + 3 + 3 * 4} );;
```

3. To call functions from within an infix expression and manipulate its return values, call the functions as you would normally and remember to enclose all infix expression in curly braces.

```
            { 1 + 2 + (foo 3) + ( baz {3 + 4} ) };;
```

3.3.5. Lisp-Inspired Constructs

3.3.5.1. quote

Prefacing values with a quote, e.g., `‘(a),` creates a list with a as its element.

```
> ‘(1 2 3 “hello”);;
‘(1 2 3 “hello”)
```

3.3.5.3. is Function

`(is a b)` returns true if the value of **a** equals that of **b**. Returns false otherwise.

```
> (is ‘a ‘a);;
true
> (is ‘a ‘b);;
false
```

3.3.5.4. head Function

(head a) expects **a** to be a list, and returns its first element. Its return type is sometype. If you wish to pass the head of a list into any other function, you must use a type annotation to tell the compiler what the type of the head is.

NOTE: The below functions, which operate on lists, will have their output shown alongside the type inferred by our type inference system. This is to demonstrate the care that should be taken when manipulating lists in order to obey the compiler’s typing rules.

```
> (head ‘(a b c));;
- : sometype

> (int (head ‘(1 3 42)));;
   -  : int
   -  1
```

3.3.5.5. tail Function

(tail a) expects **a** to be a list. It returns a list that is the same as a, without the head of a.

```
> (tail ‘(a b c));;
   -  : (sometype) list
   -  : ‘(b c)
```

3.3.5.6. cons Function

(cons a b) expects the value of **`b`** to be a list, and returns a list comprised of **`a,`** as the first element, followed by the elements of list **`b.`**

```
> (cons ‘a ‘(b c));;
   -  : (sometype) list
   -  : ‘(a b c)

> (cons ‘(a) ‘(b c));;
   -  : (sometype) list
   -  : ‘(‘(a) b c)
```

3.3.5.7. if expression

(if a b c) where **a** is an S-expression which returns a boolean; equivalent to “if **a** then **b** else **c**”. The types of expressions b and c must be the same. The return type of “if” is the type of expression b and expression c.

```
> (if true 1 2);;
   -  : int
   -  : 1
> (if true 1 1.2);;
   -  Fatal error: exception Failure("Type Incompatible Error: The types
      int and float are incompatible in if.")
```

3.3.5.8. do and eval

(do a b c ...) is a function which takes several quoted lists as arguments. It runs, **a**, then **b**, then **c** sequentially. The arguments must be a quoted function call expression. It returns the return value of the last function call. Although I have used only three arguments here, ‘do’ can take unlimited arguments.

```
> (do ‘(prn "where") ‘(prn "are") ‘(prn "the") ‘(prn “spaces”));;
   -  : unit
   -  : wherearethespaces
```

(eval e) is a function which takes one quoted list as argument. It evaluates the quoted list as if it were an unquoted function call. The argument must be a quoted function call expression.

```
> (eval ‘(prn “hello”));;
    -: unit
    -: hello

> (eval ‘(+ 1 2 3))
    -: 6
```

3.3.5.9 call, dot, module

These functions provide a bridge between Superscript and existing Javascript code.

```
      (= express (module “express”));;
      (= app (call express));;
      (call app “get” “/:name”
            (fn (req res)
                  (call res “send”
                        (++ “Hello “ (dot req “params” “name”)))));;
```

In this example we are importing the express module using the module function, using the call function to instantiate the express module with no arguments and to register a callback for the /:name route with the app object. The dot function is used to access child elements of a javascript object, in this case to get req.params.name.

3.3.5.10 Type functions

Type conversion functions include string_of_float, float_of_string, int_of_string, string_of_boolean, string_of_int. These convert the rightmost type to the leftmost type.
Type annotation functions include int, float, string, list, boolean. These must be used with list operations, if the result of those list operations is going to be passed to any other built-in function. This is because operations like head return a sometype - if this ‘sometype’ is then operated upon, it must be annotated to its correct type, otherwise the compiler will throw a type incompatibility error.

```
> (+ 1 (head ‘(1 2 3)));;
Fatal error: exception Failure("Type Incompatible Error: The types
sometype and int are incompatible in +.")

>(+ 1 (int (head ‘(1 2 3))));;
   -  : int
   -  : 2
```

Finally, the type function returns the runtime type of its argument. It takes only 1 argument.

```
> (type 1);;
      -:  int
```

3.3.6. List Operations

3.3.6.1. Head/Tail Functions

Head takes a list and returns the first element of that list, which has type sometype.

```
> (head `(a b c));;
-: sometype
-: a
```

Tail takes a list and returns everything after its first element, the result has type list of sometype.

```
> (tail `(a b c));;
-: (sometype) list
-: ‘(b c)
```

Cons takes an expression and a list, and appends the expression onto the front of the list. The result has type list of sometype.

```
> (cons 1 ‘(1.2 2.2 3.3));;
-: (sometype) list
-: ‘(1 1.2 2.2 3.3)
```

3.3.3.7 Standard Library Functions

All standard library functions, written in Superscript, are contained in the “stdlib.ss” file. Included below are the function names and their types, for reference. Please view the stdlib.ss file for the internal implementation of these functions, and for further documentation with detailed comments about usage.

val identity : 'a -> 'a
val length : (sometype) list -> int
val nth : (sometype) list -> int -> sometype
val first : (sometype) list -> sometype
val second : (sometype) list -> sometype
val third : (sometype) list -> sometype
val fourth : (sometype) list -> sometype
val fifth : (sometype) list -> sometype
val sixth : (sometype) list -> sometype
val seventh : (sometype) list -> sometype
val eighth : (sometype) list -> sometype
val ninth : (sometype) list -> sometype
val tenth : (sometype) list -> sometype
val last : (sometype) list -> sometype
val map : (sometype) list -> (sometype -> 'a) -> (sometype) list
val fold_left : ((sometype) list) -> 'a -> 'b -> 'a
val fold_right : 'a -> (sometype) list -> 'b -> sometype
val filter : (sometype) list -> ('a -> bool) -> (sometype) list
val append : (sometype) list -> (sometype) list -> (sometype) list
val reverse : (sometype) list -> (sometype) list
val drop : (sometype) list -> int -> (sometype) list
val take : (sometype) list -> int -> (sometype) list

val intersperse : (sometype) list -> 'a -> (sometype) list
val member : (sometype) list -> 'a -> bool
val zipwith : ((sometype) list) -> (sometype) list -> sometype -> sometype -> 'a -> (sometype) list
val zipwith3 : ((sometype) list) -> ((sometype) list) -> (sometype) list -> (sometype) -> sometype -> sometype -> 'a -> (sometype) list
val zip : (sometype) list -> (sometype) list -> (sometype) list
val unzip : (sometype) list -> sometype
val format_boolean : 'a -> string
val format_int : 'a -> string
val format_string : 'a -> string
val format_float : 'a -> string
val stringify_list : (sometype) list -> (sometype -> 'a) -> string
val format_boolean2d : (sometype) list -> string
val format_int2d : (sometype) list -> string
val format_float2d : (sometype) list -> string
val format_string2d : (sometype) list -> string
val print_list : (sometype) list -> (sometype -> 'a) -> unit

3.3.8. Caramel, or "Syntax Modifications," and Preprocessing

"It's hot syntactic sugar for Superscript " - Author Unknown

Use the executable ./preprocessor, created automatically by ./make, to transform an input Caramel code file into an output file written in normal Superscript syntax. The preprocessor produces an output file with the same name, but with ".ss" file extension. You can then execute the output file by executing ./geb on the .ss file.

Throughout this manual, we have introduced you to the basics of Superscript. Superscript is obviously a Lisp, and for that reason programs in Superscript are composed of S-expressions. We have an example of a function call written below.

```
> (operator argument argument argument);;
```

A list of the same values looks like the following:

```
> ‘(operator argument argument argument);;
```

By simply quoting the first list, we do not look at it as a function call, but a list. This type of syntax, we believe, is both one of Superscript's greatest strengths but also one of its greatest shortcomings. Making programs look like data structures is something that is extremely attractive to not just the Lisp community, but to someone who understands how our language's programs are almost prewritten in their AST.

The shortcoming of this type of syntax is that without either the ability to separate parentheses or the user understanding the idea of constantly applying a function to a list of arguments, things like basic arithmetic seem less natural.

With Superscript attempting to be a language which focuses on simplicity allowing users to "do a lot with little", Superscript has an alternate syntax mode which prevents 'parenthetical hell' without losing the capabilities of Lisp.

Superscript already utilizes abbreviations to common Lisp ideas, such as using 'x instead of (quote x), so additional abbreviations have been added so that programs in Superscript using Caramel syntax can be efficiently parsed without losing the semantic power of Lisp.

You can write in Caramel syntax and preprocess your code into Superscript syntax. There are four key components which can must be used together if you are writing in Caramel syntax.

1. **Expression termination**
2. **Modern Expressions (m-expressions)**
3. **Curly infix expressions (c-expressions)**
4. **Indented Expressions for reserved keywords (i-expressions)**

3.3.8.1 Expression termination in Caramel vs. Superscript.
If you are using the preprocessor to process Caramel into Superscript, note the following changes to standard syntax: **Do not use ";;" ever.** Separate expressions with newlines, instead of ";;". Double newlines after each expression are allowed, as they are ignored by the preprocessor.

```
> "Hello"
"Hello"
```

3.3.8.1. Modern Expressions

Modern expressions can be built using the following rule:

Rule: Anything outside a bracket without a space gets slurped into an s-expression.
a. f(…) -> (f …)
b. f({…}) -> (f {… }) where {…} sees rule 1.

3.3.8.2. Curly infix expressions

If you are using the preprocessor to translate Caramel into Superscript, NO infix expression may be used as a top-level expression. In Caramel, infix expressions can only be used as subexpressions.

1. To call a function from within an infix expression, seeing m-expressions above.
```
            { 1 + 2 + foo(3) + baz({3 + 4}) }
```
2. To parenthesize within infix expressions, use curly braces. Function calls still use () around args, though the arg may be infix if wrapped in {}.
```
            {{ baz({3 + 4}) + 3} * 100 }
```
3. To call a function with infix expression as args:
```
            foo({1 + 3 + 3*4})
```

Note: Parens wrap all args - left parens after the function name, and right parens after the end of arguments. The arguments themselves may be wrapped in {} to be made infix.

3.3.8.3. Indented Expressions for If/Fn/Do/Eval

The reserved words **if**, **fn**, **do**, and **eval** must obey special indentation rules when you are writing in Caramel: first of all, the reserved word and its first argument go on the same line. The remaining arguments go on the following lines, one per line, and each must be indented more than the parent line. In addition, all of the remaining arguments must be indented at the same number of spaces and/or tabs as each other.

Example: if
```
if a
      b
      c
```

As shown above, a must be on the same line as ‘if’. b and c must be indented more than the first line, and b and c must be indented the same number of spaces or tabs.

Example: fn
```
fn (a_1 a_2 a_3 … a_n)
      body
```
For fn, the list of arguments, a_1, a_2, …, a_n must be on the same line as the keyword fn, and the entire list of arguments must be enclosed in parentheses. The second argument of the anonymous function declaration, the body expression(s), must be on the next line(s), and must be indented farther than the previous line.

Example: do
```
do a
      b
      c
```

Example: eval

```
eval a
      b
      c
```

Refer to ‘if’ above.

Example of Caramel syntax:

```
= (foo
    fn (x)
       {x + 1})

= (fib
   fn (x)
     if {x is 0}
        0
        if {x is 1}
           1
           {fib({x-1}) + fib({x-2})})
```

Result of Preprocessing:

src $ ./preprocessor caram.s **/* usage: ./preprocessor [file].[extension] produces [file].ss */**

```
(=  foo

(fn (x)
       {x + 1}));;
(=  fib

(fn (x)
(
     if {x is 0}
        0
(
        if {x is 1}
           1
           {(fib {x-1}) + (fib {x-2})}))));;
```

3.3.8.2 Operator Precedence and Associativity

The table below shows all Superscript operators, from lowest to highest precedence, along with their associativity.

Operator	Description	Associativity
=	basic assignment	right
or	logical OR	left
and	logical AND	left
is isnt	equality comparison inequality comparison	left
> < >= <=	relational comparison	left
++	string concatenation	left
+ - +. -. mod	integer addition, subtraction float addition, subtraction modulo	left
* / *. /.	integer multiplication, division float multiplication, division	left
not	logical NOT	right

Footnotes & References

[0]: Javascript is Assembly Langauge for the Web
http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebPart2MadnessOrJustInsanity.aspx
[1]: 7 Principles of Rich Web Applications. Guillermo Rauch.
http://rauchg.com/2014/7-principles-of-rich-web-applications
[2]: Wat, a lightning talk by Gary Bernhardt. https://www.destroyallsoftware.com/talks/wat
[3]: The Roots of Lisp. Paul Graham. http://languagelog.ldc.upenn.edu/myl/ldc/llog/jmc.pdf
[4]: Arc. Paul Graham. http://www.paulgraham.com/arc.html
[5]: Insertion Sort Lisp Implementation. Bob Dondero.
http://cs.princeton.edu/courses/archive/spr11/cos333/lectures/17paradigms/sort.lisp
[6]: Clojure for the Brave and True. Daniel Higginbotham http://www.braveclojure.com/
[7]: Arc Language. http://arclanguage.github.io/
[8]: Arc Forum. http://arclanguage.org/forum
[9]: Curly infix, Modern-expressions, and Sweet-expressions: A suite of readable formats for Lisp-like languages, David A. Wheeler:
http://www.dwheeler.com/readable/sweet-expressions.html

[10]: History of Lisp. John McCarthy. http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
[11]: Memo 8: Recursive Functions of Symbolic Expressions and Their Computation By Machine, John McCarthy. ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-008.pdf
[12]: Hindley Milner type inference. http://c2.com/cgi/wiki?HindleyMilnerTypeInference
[13]: Poly implementation of Algorithm W:
https://github.com/andrejbauer/plzoo/tree/master/src/poly

4. Project Plan

4.1 Identify process used for planning, specification, development, and testing

4.1.1 Planning

Throughout the term, we used a lean, iterative planning process. We planned roles immediately after the first class with Sam acting as manager, Tommy as tester, Uday as language guru, and Phoebe and Michelle acting as system architects. The initial language was developed over the first weekend by Uday who wrote an eval function with the first core functions. Our process of deciding things very early and quickly allowed us to keep making massive headway every week. We would meet every Monday from 5:25 PM where would work until at least 1 AM. Understanding that programming requires a certain level of onboarding time to solve problems, we committed to long periods of time during which we would be able to really knock down our goals for the next week in one sitting. We also were very focused on developing the entire language and deciding what to implement early on and committing to it preventing us from over promising and under delivering.

4.1.2 Specification Process

We had an initial set of features which were focused on keeping the Lisp homoiconic and dynamic. We were able to look at Arc and Clojure for ideas on how to develop a Lisp appropriately. We built a macroed version of Superscript on top of Arc which runs ontop of the MzScheme 372 compiler. This allowed us to determine what functions we could allow and what we couldn’t as we were able to work with the Arc or Scheme grammar. From this we were easily able to write our language reference manual which we followed and used in conjunction with the early level macroed Superscript. This was tough to continue using once we implemented type inference (where we turned to Poly, see reference 13).

4.1.3 Development Process

We would work every week in pairs. The system architects would focus and develop stuff like type inference on Michelle’s machine while more of the language generation and language issues would be handled on Sam’s machine. The emphasis on paired-programing paid off dividends. Everyone in the group can agree that over the course of the term, we developed specialization in our respective fields. Focusing on breaking and hacking around Javascript to find clever ways of executing closures, or playing with Poly to implement type inference, by breaking into groups we solved the classic programmer problem onboarding time issue.

Often times when faced with a problem in computer science, there is a certain amount of time required to understanding the problem and only then can you solve it. Usually, this is disrupted by things like meetings and often prevents programmers from seeking help. By pairing off and working together, we were able to prevent this as a minimum of two people would be able to understand and debug problems at the same time. If a problem became too difficult for a particular pair, only then would we reach across and add more people to the group to solve problems.

4.1.4 Testing Process

Every time a new aspect of the language was written, we wrote a test for it. Upon running these tests, we would also run the tests that were made for previously completed areas of the language. We kept this practice so that even in the case that a new change messes up previously behaving code (which we encountered), it would be easily detected and we would be better able to monitor the correctness of our language as it grew.

4.4 Table with roles

These roles were far from specific as everybody more or less contributing to everything resulting in a very t-shaped working experience where everybody performed as a specialist and a generalist at the same time.

Name	**Roles**
Michelle Zheng	**System Architect**
Samurdha Jayasinghe	**Manager**
Tommy Orok	**Tester**
Uday Singh	**Language Guru**
Yu Wang	**System Architect**

4.5 Software development tools

These were all tools used for the development of Superscript

Tools

OCaml
Javascript/Node.js Runtime
Node Debug
Git
Github
Circle CI
MzScheme 372
Poly
Slack
Unix

4.6 Project Log

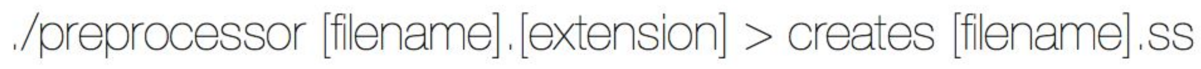
Date	**Milestone**
Sep 9	**Language idea developed**
Sep	**Language grammar complete**
Sep 21	***Macroed* version of Superscript complete**
Sep 26	**Proposal complete**
Sep 27	**Work on Superscript beings**
Sep 27	**Testing begins**
Sep 30	**Proposal Due**
Oct 1	**First version of AST complete**
Oct 2	**Language Reference Manual Started**
Oct 25	**Compiler (lexer and parser) complete**
Oct 26	**Language Reference Manual complete**
Nov 9	**Initial Javascript Generator complete**

Nov 14	**Initial Javascript Library**
Nov 15	**Preprocessor complete**
Nov 16	**Hello world and basic functions complete**
Nov 30	**Initial Superscript Library complete**
Dec 15	**Type checker complete**
Dec 21	**Language complete**

5. Architectural Design

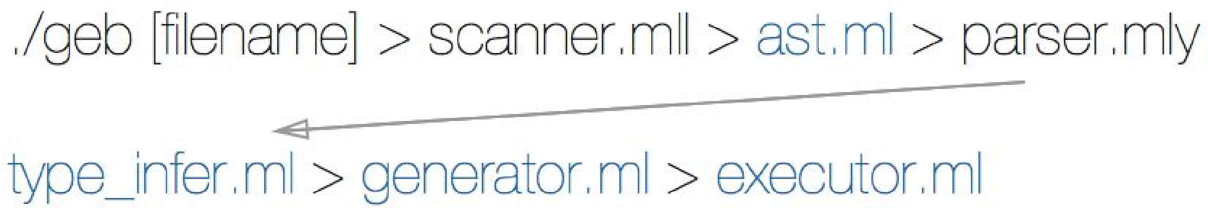
5.1 Diagram

5.1.1 Running Caramel preprocessor to create a .ss file



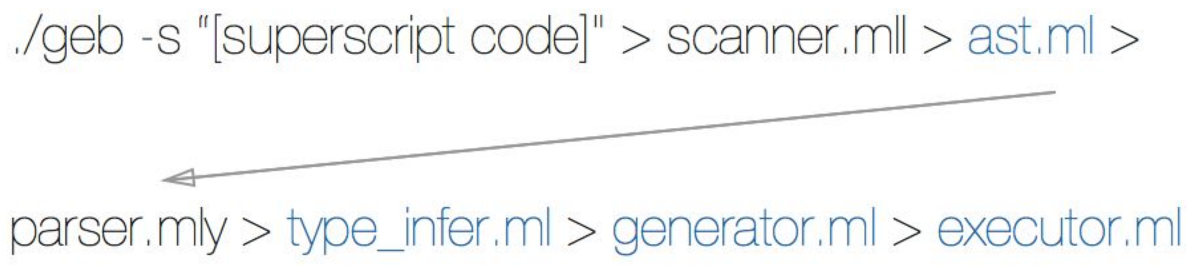
```
./preprocessor [filename].[extension] > creates [filename].ss
```

5.1.2 Running a .ss file



```
./geb [filename] > scanner.mll > ast.ml > parser.mly
type_infer.ml > generator.ml > executor.ml
```

5.1.3 Running an individual line of Superscript



```
./geb -s "[superscript code]" > scanner.mll > ast.ml >
parser.mly > type_infer.ml > generator.ml > executor.ml
```

5.2.1 Preprocessor (Michelle Zheng, Yu Wang)

The preprocessor is created automatically as an executable file, by the ./make command. Running **./preprocessor [file].[extension]** on a Caramel-style code text file will create an output text file, **[file].ss**, which contains the same code processed into Superscript. The output

Superscript code file can then be translated into JavaScript, as normal, by running **./geb [file].ss**.

The preprocessor is basically another scanner, that acts prior to the Superscript scanner, that scans in the Caramel input file. It uses specific lexing rules and a stack to track the indentations in the Caramel code, and it adds left and right parens according to the indentation levels. It also adds ;; to terminate expressions, and it transforms modern function calls (in the +(1 2 3) style) into Lisp-style (+ 1 2 3). The preprocessor makes these changes to the original text file and outputs the resulting lexbuf, into the output file, which is a text file containing the transformed code. This output file is now in Superscript syntax and may be executed with ./geb just like any other Superscript file.

See the section on Caramel to learn the syntax differences between Caramel and Superscript.

5.2.1 The Lexer (Michelle Zheng, Uday Singh, Yu Wang)

The scanner.mll lexer scans in tokens, which then go into the parser in the next step. Unique elements of the Superscript accepted tokens include:

/* C-style Comments */
;; terminates expressions
different int vs. float arithmetic operators (+ - * / , vs. +. -. *. /..)
identifiers must start with a letter, and can contain [0-9] and '_'
reserved keywords: fn, if, do, eval, =

The lexer reports the line and character where a lexer error occurs, for instance, if an invalid symbol such as & or ^ is read. For input at the command line, the line number will always be 1 (unless you use escaped newlines); but for code read from a file, the line count starts at line 1 and counts up for every newline or carriage return encountered.

5.2.2 The Parser (Michelle Zheng, Uday Singh, Yu Wang)

Grammar:

The structure of the Superscript grammar is as follows:

```
Program: expr list
      expr:
            atom
                  -int, float, boolean, string, empty list
                  -identifier
                        -built-in function identifier
            quoted list
                  example: '(1 2 3)
            s-expression
                  (if a b c)
```

```
                (fn (...) e)
                unquoted list – function call
            infix expression
                example: {1 * foo({2 – 3}) + {4 * 5}}
```

Error Recovery:
When the parser encounters an error, it attempts to recover so it continues parsing and then reports multiple errors. However, having a parsing error stops the execution of the compiler after the parsing stage; no JavaScript code is produced. All errors identified are then reported at the command line. Typical syntax errors include:

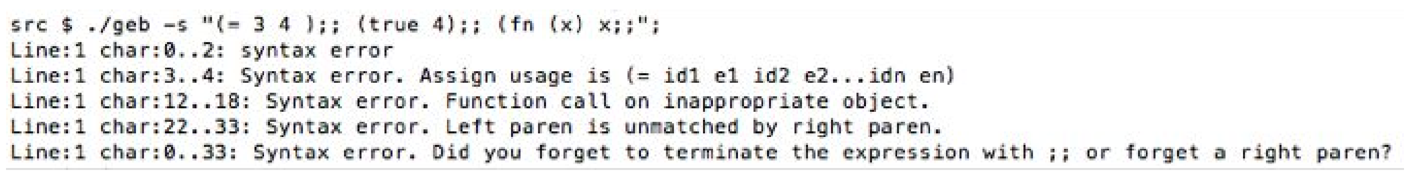
–Missing ";;"

–Unmatched "(" in s-expressions

–Function call on inappropriate object, e.g. (true 4);;

–Incorrect usage of assignment, e.g. (= 3 4);;

Example:



```
src $ ./geb -s "(= 3 4 );; (true 4);; (fn (x) x;;";
Line:1 char:0..2: syntax error
Line:1 char:3..4: Syntax error. Assign usage is (= id1 e1 id2 e2...idn en)
Line:1 char:12..18: Syntax error. Function call on inappropriate object.
Line:1 char:22..33: Syntax error. Left paren is unmatched by right paren.
Line:1 char:0..33: Syntax error. Did you forget to terminate the expression with ;; or forget a right paren?
```

5.2.3 Static Type Inference (Michelle Zheng, Yu Wang)

Relevant files: type_infer.ml

We implemented our own type inference system based on Hindley Milner type system with algorithm W, following the example of Poly's implementation (see reference 13). The following is the table of the type system of our language.

type in our type system	expression of this type
TInt	int, e.g. 2, 42
TFloat	float, e.g. 3.2, -55
TString	string, e.g. "hello world"
TBool	boolean, e.g. true, false
TUnit	nil, e.g. (prn "hello world")
TSome	some type that can't be inferred, e.g.

	(head ‘(1 2 3 4))
TSomeList	a list of elements of heterogenous types, e.g. ‘(1 true 42 “hello world”)
TJblob	JavaScript object, e.g. external JavaScript library
TException	exceptions, e.g. (exception “index out of bound”)
TParam	variables, e.g. a, b
TArrow	function, e.g. the type of (fn (x y) (+ x y)) is int -> int -> int; the type of (fn (x) x) is a’ -> a’

With the knowledge of constraints of input types and output types of our built-in functions, we can infer the constraints of usage of customized functions. For one expression, we calculate all the constraints on the tokens based on the current global environment, which contains the binding of variable names and its actual type, and check if there is any contradiction within these constraints, and report the compiling error if any contradiction exists.
Here is some examples:

```
>     (+ 1 1.1);;
   -  Fatal error: exception Failure("Type Incompatible Error: The types
      float and int are incompatible in +.")

>     (= my_func (fn (a) (prn a)));;(my_func 10);;
   -  Fatal error: exception Failure("Type Incompatible Error: The types
      string and int are incompatible in my_func.")
```

We also allow recursive functions defined with assign operators. For example:
(= fib

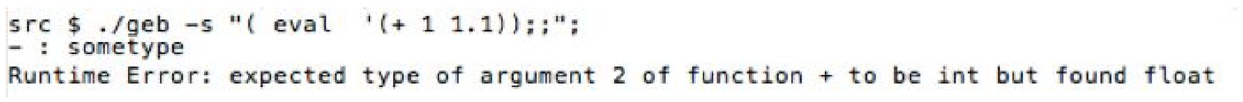
```
>         (fn (x)
      (
           if {x is 0}
              0
      (
              if {x is 1}
                 1
                 {(fib {x-1}) + (fib {x-2})}))));;
>          (prn (string_of_int {10 + {42 - 1} * 3 + (fib {1 + 3})}));;
-     int
-        136
```

5.2.5 Code Generator (Samurdha Jayasinghe, Yu Wang)

The generator performs a depth-first traversal of the abstract syntax tree to generate javascript code. Instead of using native javascript data types directly, we wrap all superscript data in special object wrappers because superscript has data types that are not natively supported by javascript (for example there are no int/float types in javascript). This allows us to check the type and number of the actual arguments passed into built-in functions in Javascript.

Runtime type checking:

Since our language has built-in function eval, which takes a quoted function call, executing this function with the following arguments, and it makes the function call get passed from the compile-time type inference checking, we will need to check the types at runtime. In order to implement this, all generated standard functions have argument type and number assertions so that we can throw runtime errors with instructive messages for cases not handled by the compile-time type inference system. Here's an example:



```
src $ ./geb -s "( eval  '(+ 1 1.1));;";
- : sometype
Runtime Error: expected type of argument 2 of function + to be int but found float
```

What's more, we also have type annotations in our language. We can do (int 1), (float 2.2), (string "hello world"), (boolean true). However, what this type annotation does in our compile-time type inference system is just to change the following argument's type to the target, while we need some actual type checking at runtime. For example, for "(int 1)", at runtime, we will make sure that 1 is actual int.

5.3 The Javascript Libraries (Samurdha Jayasinghe, Uday Singh)

The Javascript libraries consist of a set of low level functions that represent the fundamental language constructs in superscript. One of the difficult things about Javascript is that Javascript does not have the same type sensitivity that Superscript has. As such, it was necessary to develop a library for Javascript that could provide proper boxing for values handled. We developed a number of types within Javascript and used a library to do so which is all called everything a .ss file is compiled into the a.js file.

5.4 The Superscript Libraries (Michelle Zheng, Samurdha Jayasinghe, Tommy Orok, Uday Singh, Yu Wang)

In contrast to the built-in operations, which are written in JavaScript, the Superscript library (stdlib.ss) is written entirely in our own language. The standard library includes: the identity function, list manipulation functions,and stringify/printing functions.
Examples of list manipulation functions include length, nth, map, fold_left, reverse, drop, zipwith, member. The printing / stringify functions include format_[type], stringify_list, print_list. These enable the stringifying or printing of an entire homogeneous list of type [type] by passing in the format_[type] function and the list to either stringify_list or print_list.

Please see **page 20, Standard Library Functions**, for a complete listing of standard library functions.

5.5 Tests (Tommy Orok)

The test file manipulates the modules of the language from start to end. It first serves input into the Parser whose output is fed into the Lexer, whose output in turn is fed into the Generator. The Generator's output then is written to the file to be executed.

5.6 Writing Credits

5.6.1 Introduction (Uday Singh)
5.6.2 Tutorial (Uday Singh)
5.6.3 Language Manual (Michelle Zheng, Tommy Orok, Uday Singh, Yu Weng)
5.6.4 Project Plan (Uday Singh)
5.6.5 Architectural Design (Michelle Zheng, Samurdha Jayasinghe, Tommy Orok, Uday Singh, Yu Weng)
5.6.6 Test Plan (Tommy Orok, Uday Singh)
5.6.7 Lessons Learned (Michelle Zheng, Samurdha Jayasinghe, Tommy Orok, Uday Singh, Yu Weng)
5.6.8 Appendix (Uday Singh)

6. Test Plan

6.1 Testing Phases
6.1.1 Unit Testing (Lexer, parser, code generation)

We tested the Lexer and Parser in conjunction throughout the semester. We did this by feeding various input into the Lexer, piping the output into the Parser, and comparing the resulting Abstract Syntax Tree with the expected AST provided with the input.

The tests for the code generation were done subsequently. After the AST from the parser was confirmed to be equivalent to the expected AST, it was then fed into the generator storing the resulting code in a javascript file in the running directory. The Javascript file would then be executed and it's output compared with expected output also provided with the input.

6.1.2 Integration testing

We had an array of different base tests (that were later partitioned in order to test edge cases) that covered all areas of our language:

- Printing
  - Used print functions (prn/pr) with all basic types (string, boolean, float, int).
- Defining Functions
  - Used “=” operator to define functions and examined output to make sure behavior was correct.
- Retrieving Type of Expression
  - Created various expressions (some simple, some complex) and compared the result of our “type” function to the expected type.
- Comments
  - Repeated tests and added comments in random places in code to verify they were indeed being ignored.
- Operators
  - Used relational, boolean, and arithmetic operators on various expressions and compared expected result to actual.
- Postfix/Infix Expressions
  - Tested expressions that had either only Postfix expressions or infix expressions, then wrote tests that included a mix of both.
- boolean logic
  - Tested expressions that used raw booleans as well as booleans that are the result of a sub-expression.
- Standard Library Functions
  - Made a test for each of the StdLib functions demonstrating their behavior.
- Control Flow
  - Mostly tested in functions. Defined functions with nested if statements and compared actual behavior with expected behavior with different inputs.
- Variables and Scope
  - Defined variables and made called them in separate parts of the test to make sure they were visible when and only when they were supposed to be.
- type inference
  - Called functions and operators with incorrect types and manually inspected if the reason for failing aligned with the expected reason (incorrect types).

6.1.3 System testing

The system testing was completed as a result of our integration and unit testing. The different modules of our language were tested individually and subsequently, verifying the result each step of the way every time the tests were run. This process culminated in a system test being that the end result is examined and verified as well as the intermediate results.

6.2 Automation

All of our tests were kept inside a list of tuples in our test file. The tuples consisted of: a description of the test, the input code, the expected ast, and the expected output. The main engine for the test file is basically a function that takes one of these tuples and runs the input through the lexer, parser, and generator while comparing the intermediate AST and output. We use List.iter to call this function on every test in the list. In this way, we were able to automate the running of these tests for use anytime a change in our language occurred.

6.3 Test suites

All of our tests were put in one file. Each test included a description of its purpose so that we could differentiate its function from others'.

6.4 Testing Roles
Sam and Uday set up the initial Circle CI regression testing suite for OUnit, which was dropped for Tommy's custom build testing infrastructure. Tommy created the testing infrastructure including the automation of the regression tests, the testing scripts, and decided to have the test program itself execute the resulting code and examine results. Our shift from many individual tests in separate files handled by Circle CI to a number of tests in a single file actually made developing tests easier once the OCaml testing function was developed as tests were passed in as tuples. It was definitely an alternative method to standard regression testing, but ended up being more powerful for us in the long run.

7. Lessons Learned

Uday Singh:
This was an eye opening experience. I have this belief that you only like things if you like the process of doing them, and I never thought I would enjoy the process of building something like this as much as I have. I'm glad we were able to work with Lisp in a way which seemed difficult, and almost breaking the concept of Lisp. It was approaching problems like this that made this project exciting throughout the term. I think functional programming is a beautiful thing, and the ability to make something big out of something so small is what defines not just Superscript, but working in OCaml as well. Working in pairs, learning new things constantly from my group, and finding unconventional ways to make things "just work" in a world structured by OCaml was tough, but exhilarating. I have nothing but the utmost respect and appreciation for everything my group did. I could not have done anything without them. Every single person worked like mad to get this thing done, and I'm glad at the end, when we fired up that Express server that we were able to eat our own dog food and actually use what we had built.

This was the perfect mix of creativity and engineering, and really allowed us to flex our muscles as hackers in a way no other course at Columbia has. If I had any advice, despite Prof.

Edwards' dislike for Lisp (((or its parentheses))), try to build your project like a Lisp program: think of the smallest set of tools you can equip your user with enabling them to build anything. Think little, and prototype as much as you can. Don't be afraid to experiment, and don't be afraid to break things, because realistically you will break everything anyway.

Samurdha Jayasinghe

This experience allowed me to bring together an exceptional group of engineers and create something remarkable. What I found most insightful is that none of the individuals alone would've been able to achieve anything even remotely close to what we were able to achieve as a team. By letting different individuals take responsibility for different roles, allowing them to specialize in their respective roles, and giving them freedom to explore creative solutions instead of micromanaging their actions allowed the entire group to act cohesively in such a way that was not only enjoyable but also challenging at the same time.

Tommy Orok

I can't say that I didn't know this already, but I must still mention one of the most valuable things I've seen during this project is the value of starting early. I have witnessed many friends in previous semesters go through constant hell towards the end of it due to procrastination. I've also seen the value of functional programming in a way that I never have before. At the beginning of the semester, it was a bit annoying to work with OCaml, but all of us quickly realized throughout the course that OCaml was one of the best choices for such a project. The most valuable thing I've actually learned from this project, however, is probably the testing skills. I've always been a lazy tester in the past. Having a role where I would need to dedicate most of my focus on the testing and improvement of code rather than just the implementation as I usually do enabled me to really be patient and more responsible with testing even in my other classes.

Advice: Make sure the group members that have the smaller roles spill into the bigger roles *early* on in the semester or else the distribution of work can get skewed quickly. Once this happens, it may be difficult to balance it back out since it may not even be worth the time you'd need to sufficiently get up to speed on a separate area of the project in order to be capable of making significant contributions.

Michelle Zheng:

During this project, I learned the power of OCaml and recursion. These were especially useful in the implementation of type inference, where creating and solving the constraints became intuitive due to the recursive nature of OCaml.

In addition, since our type system includes heterogeneous lists, we overcame the challenge of implementing algorithm W to type-check these lists. Ultimately, we decided that all lists would be of type TSomeList of TSome (a list of sometype), and any element of a list is of type TSome (sometype). For the head operation, we implemented a combination of compile-time type annotation, and a runtime check to ensure that the type annotation was accurate. The work we did on type inference paid off, and I learned how valuable the type inference system was - whenever we turned it off, it became much harder to debug the Superscript code, since the resulting code in JavaScript has very lenient typing rules, and seldom alerted us of any type errors. When the type inference worked, it became a valuable tool for recognizing whether our Superscript programs were behaving correctly, especially for defining some of the more complicated functions we wrote in Superscript in our Standard Library.

In addition, I learned the value of paired programming and managing CVS; it was exciting to program with all my teammates after class, and to work off the progress that others had made. It was incredibly rewarding to see the final product - a result of multiple disparate pieces by different people, coming together in a pretty sweet language.

Yu Wang

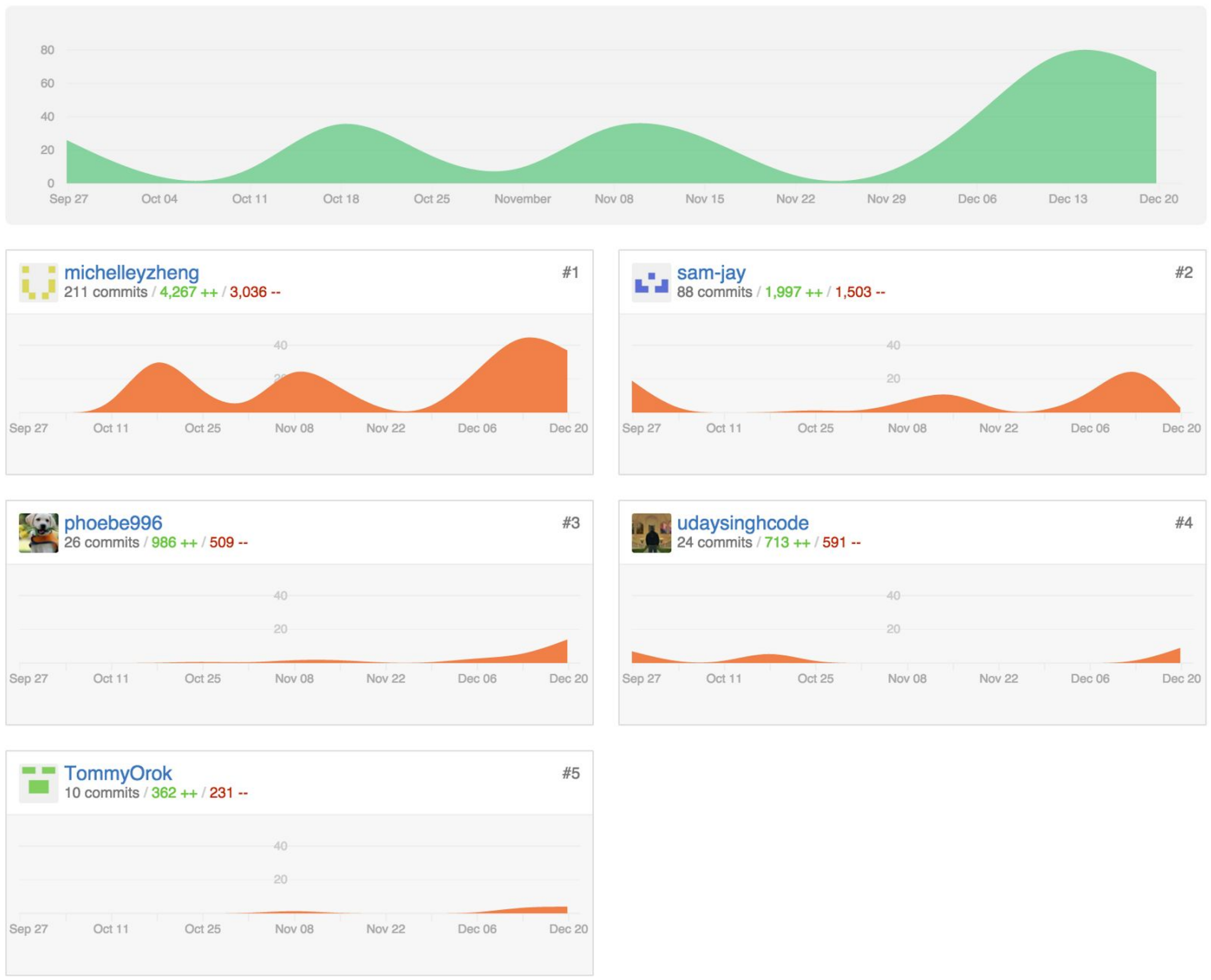
This has been a very unforgettable semester learning and coding with teams, and solving problems that we originally thought were impossible to fix. One of the hardest thing for me in this project is that we needed to get familiar with Ocaml, Lisp and Javascript in a short period of time. Coding in functional languages is totally different from coding in structural languages, and I needed to switch the way of thinking in solving problems, which was quite challenging for me at the beginning. On the other side, this also becomes one of biggest things that I've learned from this project. Although I hadn't had any chance to learn or code in functional languages until this class, the mechanism inside of functional language is quite efficient and it can be very productive compared to many other kinds of languages. As we knew more about Ocaml, we started to love it and enjoy how efficiency and neat it is.

Another thing that I’ve learned a lot from is implementing the compile-time type inference system and runtime type checking with my team. By looking into the Hindley Milner’s type system, algorithm W, and implementation of some other languages’ type systems, we figured out how to produce constraints for expressions recursively and how to solve these constraints to check contradictions. It’s hard to track any existing contradiction in these constraints and trying to locate in which functions these contradictions arise, and explain why to help user correct their code. Since algorithm W doesn't track the expected type and actual type in constraints, we only end up finishing reporting the name of the functions where contradiction occurs in error messages by now. I'll still be interested in figuring out how to report expected type and found type after this class is over.

I love the way my team collaborated in designing, coding and debugging during this project. Especially, pair programming is so efficiency and we can always pointed out critical issues that the other didn't notice when coding.

8. Commit Graph

Programming work was done collaboratively with paired programming allowing smaller subgroups handle different features and functions. We worked often times together and would, as noted below work on certain computers more than others. The parts of the project which were handled by respective people is covered in section 5, but we wanted to highlight the commit graph for a few reasons. The first being that you can see when work picked up and slowed, but that the checkpoint throughout the term proved to be extremely valuable. Our work definitely picked up around certain due dates such as for the "hello world" or the LRM. We would definitely recommend maybe hacking a bit more over Thanksgiving.



9. Appendix

ast.ml

```
type op = Add | Sub | Mult | Div | Addf | Subf | Multf | Divf
        | Equal | Neq | Less | Leq | Greater | Geq | And | Or | Assign | Concat

type htype =
  | TInt                  (** integers [int] *)
  | TFloat                    (** floats [floats] *)
  | TBool                  (** booleans [bool] *)
  | TString                   (** strings [string] *)
  | TParam of int          (** parameter *)
  | TArrow of htype list  (** Function type [s -> t] *)
  | TSomeList of htype        (** Lists *)
  | TUnit                       (* unit type for printing *)
  | TSome                       (* sometype - used only for lists *)
  | TJblob                      (* JavaScript object type *)
  | TException                          (* Exceptions *)

type expr =                             (* Expressions *)
  Int of int                            (* 4 *)
  | Float of float                      (* 4.444 *)
  | Boolean of bool                     (* true, false *)
  | String of string                    (* "hello world" *)
  | Id of string                (* caml_riders *)
  | Assign of expr list                 (* {x = 5} OR (= x 5 y 6 z 7) *)
  | Eval of expr * expr list            (* (foo 5 21) *)
  | Nil                                 (* empty list '() *)
  | List of expr list                   (* heterogeneous list '(1 true 2.4) *)
  | Fdecl of string list * expr  (* (fn (a b) {a + b}) *)
  | If of expr * expr * expr            (* (if a b c) *)

type program = expr list

(** [rename t] renames parameters in type [t] so that they count from
   [0] up. This is useful for pretty printing. *)
let rename (ty: htype) =
  let rec ren ((j,s) as c) = function
    | TInt -> TInt, c
    | TBool -> TBool, c
    | TFloat -> TFloat, c
    | TString -> TString, c
    | TSome -> TSome, c
    | TJblob -> TJblob, c
    | TException -> TException, c
    | TParam k ->
```

```
          (try
             TParam (List.assoc k s), c
           with
              Not_found -> TParam j, (j+1, (k, j)::s))
    | TArrow t_list ->
       let rec tarrow_ren ts us c' =
         match ts with
         | [] -> us, c'
         | hd::tl ->
           (let u1, c'' = ren c' hd in
           tarrow_ren tl (us@[u1]) c'')
       in let u_list, final_c = (tarrow_ren t_list [] c) in
       TArrow u_list, final_c
    | TSomeList t -> let u, c' = ren c t in TSomeList u, c'
    | TUnit -> TUnit, c
  in
    fst (ren (0,[]) ty)

(** [rename t1 t2] simultaneously renames types [t1] and [t2] so that
    parameters appearing in them are numbered from [0] on. *)

let rename2 t1 t2 =
match rename (TArrow [t1;t2]) with
     TArrow [u1;u2] -> u1, u2
   | _ -> assert false


(** [string_of_type] converts a Poly type to string. *)
let string_of_type ty =
  let a = [|"a";"b";"c";"d";"e";"f";"g";"h";"i";
              "j";"k";"l";"m";"n";"o";"p";"q";"r";
              "s";"t";"u";"v";"w";"x";"y";"z"|]
  in
  let rec to_str n ty =
    let (m, str) =
      match ty with
          | TSome -> (3, "sometype")
          | TSomeList ty -> (3, to_str 3 ty ^ " list")
          | TUnit -> (4, "unit")
          | TInt -> (4, "int")
          | TFloat -> (4, "float")
          | TString -> (4, "string")
          | TBool -> (4, "bool")
          | TJblob -> (4, "JavaScript object")
          | TParam k -> (4, (if k < Array.length a then "'" ^ a.(k) else "'ty" ^ string_of_int k))
```

```
        | TException -> (1, "exception")
          | TArrow t_list -> let len = (List.length t_list)-1 in
                    let rec tarrow_type ts s =
                      match ts with
                      | [] -> s
                      | hd::tl ->
                                   if (List.length tl > 0) then
                                   tarrow_type tl (s^(to_str ((List.length ts)-1) hd)^" -> ")
                           else (
                                   tarrow_type tl (s^(to_str ((List.length ts)-1) hd))
                                   )
                    in
                    (len, tarrow_type t_list "")
   in
     if m > n then str else "(" ^ str ^ ")"
  in
    to_str (-1) ty

(** [tsubst [(k1,t1); ...; (kn,tn)] t] replaces in type [t] parameters
    [TParam ki] with types [ti]. *)
let rec tsubst s = function
  | (TInt | TBool | TFloat | TString | TUnit | TSome | TJblob | TException ) as t -> t
  | TParam k -> (try List.assoc k s with Not_found -> TParam k)
  | TArrow t_list -> let u_list = List.map (fun t -> tsubst s t) t_list
                    in TArrow u_list
  | TSomeList t -> TSomeList (tsubst s t)
```

executor.ml

```
open Ast;;
open Type_infer;;
open Generator;;
open Scanner;;
open Unix;;
open Message;;
open Printf;;
open Array;;

exception Fatal_error of string
let fatal_error msg = raise (Fatal_error msg)

(** [exec_cmd ctx cmd] executes the toplevel command [cmd] in
    the given context [ctx]. It returns the new context. *)
let rec exec_cmd ctx = function
```

```
  | Assign(el) ->
        let rec gen_pairs l =
        match l with
                | [] -> []
                | h1::h2::tl -> (h1, h2)::(gen_pairs tl)
                | _::[] -> raise(Fatal_error("=operator used on odd numbered list!"))
        in
        let defs = (gen_pairs el) in
        let rec addCtx ctx = function
                | [] -> ctx
                | (x,e)::tl ->
                        (* convert x from Ast.htype into the actual identifier string *)
                        let x = match x with
                                | Id(s) -> s
                                | _ -> raise(Fatal_error("first operand of assignment must be
an identifier!"))

                        in let ty =
                                (* type check [e], and store it unevaluated! *)
                          try (Ast.rename (Type_infer.type_of ctx e))

                          with
                           (* Error: RHS of assignment contained an undeclared variable *)
                           | Type_infer.Unknown_variable (msg, id) -> (

                                (* Case: unknown var != LHS of assignment --> immediately
reject *)
                                if (String.compare x id) != 0 then
                                        (Type_infer.unknown_var_error("Unknown variable " ^
id) id)

                                (* Case: unknown var on RHS == LHS of assignment -->
allow if RHS is a recursive fdecl *)
                                else
                                (

                                 (* Assume RHS is recursive fn definition:
                                        store (x, TParam _) into the context to avoid unknown
var errors *)

                                 let tparam = Type_infer.fresh() in
                                 let ctx = (x,tparam)::ctx in
                                 let recfun = Ast.rename(Type_infer.type_of ctx e) in

                                 (* Check that type of RHS was indeed a function definition
*)
                                        let ty = match recfun with
                                            | TArrow _ -> recfun
                                            | _ -> Type_infer.unknown_var_error("Unknown
variable " ^ x) x
                                        in
                                            ty
                                      )
                         )
                           in
                            print_endline ("val " ^ x ^ " : " ^ string_of_type ty);
                      (x,ty)::(addCtx ctx tl)
            in
            (addCtx ctx defs)
    | _ as e ->
      (* type check [e], evaluate, and print result *)
      let ty =
        try Ast.rename (Type_infer.type_of ctx e) with
        Type_infer.Unknown_variable (msg, id) ->  raise (Failure (msg))

      in
                print_string ("- : " ^ string_of_type ty) ;
              print_newline ();
            ctx
;;

(** [exec_cmds ctx cmds] executes the list of
    expressions [cmds] in the given context [ctx].
    It returns the new context. *)
let exec_cmds ctx cmds =
   try
      List.fold_left (exec_cmd) ctx cmds
   with
      | Type_infer.Invalid_args msg -> raise (Failure (msg))
      | Type_infer.Type_error msg -> raise (Failure (msg))
;;

(** [load_file f] loads the file [f] and returns the contents as a string. *)
let load_file f =
  let ic = open_in f in
  let n = in_channel_length ic in
  let s = Bytes.create n in
  really_input ic s 0 n;
  close_in ic;
  (s) ;;

let rec funct foo =
  try
    let l = input_line foo in
    match l with
    | _ -> l::(funct foo)
  with End_of_file -> [] ;;

let write stuff =
  let oc = open_out "a.js" in
    output_string oc stuff;
    close_out oc;
;;

(** [exec_file ctx fn] executes the contents of file [fn] in
    the given context [ctx]. It returns the new context. *)
let exec_file ctx (sysargs) =
  let filename = sysargs.(1) in
  let filecontents =
            if filename = "-s" then
                        if Array.length sysargs != 3
                          then
                           raise(Failure "Usage: ./geb [file] or ./geb -s [input] ")
                          else
                                    sysargs.(2)
            else
                        if Array.length sysargs != 2
                          then
                           raise(Failure "Usage: ./geb [file] or ./geb -s [input] ")
                        else
                                    load_file filename
      in
         let lexbuf = Message.lexer_from_string
                        (let stdlib = load_file "stdlib.ss" in
                                    stdlib ^ filecontents) in
  let program =
    try
             Parser.program Scanner.token lexbuf
    with
     | Failure("lexing: empty token") -> print_position lexbuf "Lexing: empty token"; exit
(-1)
     | LexingErr msg -> print_position lexbuf msg; exit (-1)
     | Parsing.Parse_error -> print_position lexbuf "Syntax error occurs before"; exit (-1)
  in
  (* Perform type checking, by executing exec_cmds. Print all identifiers & types *)
     ignore(exec_cmds (
            (* Add types for built-in functions to the context *)
                  List.map (fun x -> (x, Generator.arrow_of(x)))
               (Generator.get_generatable_fnames program))
             program);

     let prog = Generator.generate_prog program in
       write prog;
          print_endline (String.concat "\n" (funct (Unix.open_process_in "node a.js")))

(** The main program. *)
let main =
         if(Array.length Sys.argv < 2) then
                  raise(Failure "Usage: ./geb [file] or ./geb -s [input] ")
         else
             exec_file [] (Sys.argv)
```

finaltest.ml

```
open Ast;;
open Generator;;
open Scanner;;
open Unix;;

let load_file f =
  let ic = open_in f in
  let n = in_channel_length ic in
  let s = Bytes.create n in
  really_input ic s 0 n;
  close_in ic;
  (s) ;;

 let rec funct foo =
  try
    let l = input_line foo in
    match l with
    | _ -> l::(funct foo)
  with End_of_file -> [] ;;


let write stuff =
  let oc = open_out "a.js" in
```

```
    output_string oc stuff;
    close_out oc;
;;

let tests = [
 ("prn function should log to stdout", "(prn \"Hello World!\");;", [], "Hello World!") ;
 ("string_of_int function should make string from int", "(prn (string_of_int 10));;", [], "10")
;
 ("string_of_boolean function should make string from boolean", "(pr (string_of_boolean
true));;(pr (string_of_boolean false));;", [], "truefalse") ;
 ("type function should return type of string", "(prn (type \"Hello\"));;", [], "string") ;
 ("type function should return type of int", "(prn (type 10));;", [], "int") ;
 ("type function should return type of float", "(prn (type 2.2));;", [], "float") ;
 ("type function should return type of boolean", "(prn (type true));;", [], "boolean") ;
 ("type function should return type of list", "(prn (type '(10 20)));;", [], "list") ;
 ("type function should return type of function", "(prn (type (fn (x) (prn x))));;", [],
"function") ;
 ("assignment operator", "(= foo \"Hello\");;(prn foo);;", [], "Hello") ;
 ("user defined functions", "(= foo (fn (x) (prn x)));;(foo \"Bar\");;", [], "Bar") ;
 ("curly infix arithmetic expression", "(prn (string_of_int {5 + 3}));;", [], "8") ;
 ("+ operator with no args should return 0", "(prn (string_of_int (+)));;", [], "0") ;
 ("* operator with no args should return 1", "(prn (string_of_int (*)));;", [], "1") ;
 ("prefix integer add", "(prn (string_of_int (+ 1 2 3 4)));;", [], "10") ;
 ("prefix integer sub", "(prn (string_of_int (- 10 2 3)));;", [], "5") ;
 ("prefix integer mult", "(prn (string_of_int (* 1 2 3 4)));;", [], "24") ;
 ("prefix integer div", "(prn (string_of_int (/ 10 2 (- 5))));;", [], "-1") ;
 ("prefix float add", "(prn (string_of_float (+. .1 .2 .3 .4)));;", [], "1") ;
 ("prefix float sub", "(prn (string_of_float (-. 5.0 .2 .3)));;", [], "4.5") ;
 ("prefix float mult", "(prn (string_of_float (*. 1. 2. 3. 4.)));;", [], "24") ;
 ("prefix float div", "(prn (string_of_float (/. 10. 2. (-.5.))));;", [], "-1") ;
 ("comparing ints with is func", "(pr (string_of_boolean (is 1 1)));;(pr (string_of_boolean
(is 1 2)));;", [], "truefalse") ;
 ("comparing floats with is func", "(pr (string_of_boolean (is 1.0 1.)));;(pr
(string_of_boolean (is .1 .2)));;", [], "truefalse") ;
 ("comparing bools with is func", "(pr (string_of_boolean (is true true)));;(pr
(string_of_boolean (is false true)));;", [], "truefalse") ;
 ("comparing strings with is func", "(pr (string_of_boolean (is \"hello\" \"world\")));;(pr
(string_of_boolean (is \"world\" \"world\")));;", [], "falsetrue") ;
 ("empty list should be nil", "(prn (string_of_boolean (is '() nil)));;", [], "true") ;
 ("head function should return head of list", "(prn (head '(\"foo\" \"bar\")));;", [], "foo") ;
 ("tail function should return tail of list", "(prn (head (tail '(\"foo\" \"bar\"))));;", [], "bar") ;
 ("++ operator concatenates strings", "(prn (++ \"foo\" \"bar\" \"orange\"));;", [],
"foobarorange") ;
 ("< operator should compare ints", "(pr (string_of_boolean (< 9 10)));;(pr
(string_of_boolean (< 11 10)));;", [], "truefalse") ;
```

```
 ("if statements", "(if true (pr \"foo\") (pr \"bar\"));;(if false (pr \"foos\") (pr \"bars\"));;", [],
"foobars") ;
 ("get first element (int) of list", "(prn (string_of_int (head \'(1 2 3 4 5 6 7 8 9 10))));;", [],
"1");
 ("multiple expressions", "( prn \"hello\" );; ( prn \"world\" );;  ( prn \"people\");;", [],
"hello\nworld\npeople");
 ("set add equal to anon func then call it", "(= add (fn (x y) (+ x y) )) ;;(prn (string_of_int
(add 1 3)));; ", [],  "4");
 ("print the 5 mod 6", "(prn (string_of_int (mod 5 6)));;", [], "5");
 ("logical AND of true and false","(prn (string_of_boolean (and true false)));;", [], "false");
 ("setting and testing inequality", "(= a \"a\");; (prn (string_of_boolean (is \"a\" a)));;", [],
"true");
 ("testing relational comparison operators", "(prn (string_of_boolean (> 2 4)));;", [],
"false");
 ("testing string concatenation", "(prn (++ \"hello\" \" \" \"world\" ));;", [], "hello world");
 ("testing cons function", "(prn (head (cons \"a\" \'(\"b\" \"c\"))));;", [], "a");
 ("testing if function", "(prn (if false \"b\" \"c\"));;", [], "c");
 ("print out float with string_of_float", "(prn (string_of_float 3.5));;", [], "3.5");
 ("print out result of infix expr","(prn (string_of_int {3+ 5}));;", [], "8");
 ("print out int with string_of_int","(prn (string_of_int 3));;", [], "3");
 ("print out sum of list ","(prn (string_of_int (+ 3 4 2 5 3 2 5)));;", [], "24");
 ("print out list using print_list","(print_list string '(\"a\" \"b\" \"c\" \"d\"));;", [], "[a,b,c,d]");
 ("should print 'letter a'", "(= a \"letter a\");; (prn a);;", [], "letter a");
 ("using curly infix expressions", "(prn (string_of_int {3 + 5}));;", [], "8");
 ("handle printing negative return value","(prn (string_of_int {7 - 9}));;", [], "-2");
 ("converting int returned from function to string", "(= square (fn (x) (* x x)));;  (prn
(string_of_int (square 5) ));;", [], "25");
 ("print function's return value, using infix", "(= square (fn (x) {x * x}));;  (prn (string_of_int
(square 5) ));;", [], "25");
 ("printing return of assignment (int)", "(prn (string_of_int (= x 3)));;", [], "3");
 ("printing return of assignment (float)", "(prn (string_of_float {x = 3.5}));;", [], "3.5");
 ("use string_of_boolean to print", "(pr (string_of_boolean true));;(pr (string_of_boolean
false));;", [], "truefalse");
 ("use string_of_boolean with prn", "(prn (string_of_boolean true));;(prn
(string_of_boolean false));;", [], "true\nfalse");  (************************)
 ("print the type 'string'", "(prn (type {x = \"a string\"}));;", [], "string");
 ("print the type of return value of assignment", "(prn (type {x = 50}));;", [], "int");
 ("print type of an list", "(prn (type '() ));;", [], "list");
 ("print type of var after assingment", "(= a 2);; (prn (type a));;", [], "int");
 ("print type of float", "(prn (type 3.5));;", [], "float");
 ("print type of function", "(prn (type (fn (x y) (/(+ x y) 2)) ));;", [], "function");
 ("use head and tail, basic case", "(= foo '(1 0));; (if true (prn (string_of_int (head foo)))
(prn (string_of_int (head (tail foo)))));;", [], "1");
 ("assing var to result of sum, then print", "(= foo (+ 1 2 3 4 5));; (prn (string_of_int
foo));;", [], "15");
```

```
 ("print boolean", "(= foo true);; (prn (string_of_boolean foo));;", [], "true");
 ("print second element using head and tail","(= foo '(\"a\" \"b\" \"c\"));; (prn (string(head
(tail foo))));;", [], "b");
 ("define fibonacci function", "(= fib (fn (x) (if (is x 0) 0 (if (is x 1) 1 {(fib {x - 1}) + (fib {x -2}
)}))));; (prn (string_of_int (fib 5)));;", [], "5");
 ("print infix mixed operations", "(prn (string_of_int {5 + 3 + 6 * 7 - 4 / 2}));;", [], "48");
 ("print mixed infix and prefix", "(prn (string_of_int (+ {5 + 3 + 6 * 7 - 4 / 2} (/ 100 5 20) (* {1
- 3} 3))));;", [], "43");
 ("mixed operations", "(= foo '(\"a\" \"b\"));; (prn (++ (string_of_int {5 + 3 + 6 * 7 - 4 / 2})
(string (head foo)) (string_of_int (* {1 - 3} 3))));;", [], "48a-6");
 ("print result of infix sum", "(prn (string_of_int {1 + 2 + 3 + 4}));;", [], "10");
 ("print result of infix subtraction", "(prn (string_of_int {10 - 2 - 3}));;", [], "5");
 ("print result of infix multiplication", "(prn (string_of_int {1 * 2 * 3 * 4}));;", [], "24");
 ("print result of infix float summation", "(prn (string_of_float {.1 +. .2 +. .3 +. .4 }));;", [],
"1");
 ("print float summation", "(prn (string_of_float {5.0 -. .2 -. .3 }));;", [], "4.5");
 ("print result of infix float multiplication", "(prn (string_of_float { 1. *. 2. *. 3. *. 4.}));;", [],
"24");
 ("using if", "(if  true (prn \"a\") (prn \"b\"));;", [], "a");
 ("assing var to list and print", "(= a '(1 2 3 4));; (prn (string_of_boolean (is a a)));;", [],
"true");
 ("check for empty list equality", "(= x '(1));; (prn (string_of_boolean (is (tail x) '())));;", [],
"true");
 ("appending object to head of list", "(= x '( \"1\"));; (prn (string (head (cons (head x) '(\"a\"
\"b\")))));;", [], "1");
 ("def roundabout concat function, use in roundabout way", "(= concat (fn (x y) (if (is x '())
y (if (is (concat (tail x) y) y) (cons (head x) y) (cons (head x) (concat (tail x) y))))));;  (prn
(string (head (tail (tail (tail (tail(concat '(\"1\" \"2\" \"3\") '(\"1\" \"LAST\")))))))));;", [],
"LAST") ;

 (**************************************STANDARD LIBRARY TESTS
***********************************************************************************************)

 ("Testing stdlib function: identity", "(= x '(1 2 3 4 ));; (prn (string_of_boolean (is x
(identity x))));;", [], "true") ;
 ("Testing stdlib function: length", "(= x '(1 2 3 4 ));; (prn (string_of_int (length x )));;", [],
"4") ;
 ("Testing stdlib function: nth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (nth 5
x)));;", [], "6") ;
 ("Testing stdlib function: first", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (first
x)));;", [], "1") ;
 ("Testing stdlib function: second", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int
(second x)));;", [], "2") ;
 ("Testing stdlib function: third", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (third
x)));;", [], "3") ;
```

```
 ("Testing stdlib function: fourth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (fourth
x)));;", [], "4") ;
 ("Testing stdlib function: fifth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (fifth
x)));;", [], "5") ;
 ("Testing stdlib function: sixth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (sixth
x)));;", [], "6") ;
 ("Testing stdlib function: seventh", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int
(seventh x)));;", [], "7") ;
 ("Testing stdlib function: eigth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (eighth
x)));;", [], "8") ;
 ("Testing stdlib function: ninth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (ninth
x)));;", [], "9") ;
 ("Testing stdlib function: tenth", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (tenth
x)));;", [], "10") ;
 ("Testing stdlib function: map", "(= newprn (fn (x) (prn (string_of_int (int x)))));; (= x '(1 2
3 4 5 6 7 8 9 10));; (map newprn x);;", [], "1\n2\n3\n4\n5\n6\n7\n8\n9\n10") ;
 ("Testing stdlib function: fold_left", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string_of_int (int
(fold_left + 0 x))));;", [], "55") ;
 ("Testing stdlib function: fold_right", "(= x '(1 2 3 4 5 6 7 8 9 10));; (prn (string_of_int (int
(fold_right + x 0 ))));;", [], "55") ;
 ("Testing stdlib function: filter", "(= fx (fn (x) (> x 3)));;  (= x '(1 2 3 4 5 6 7 8 9 10));;  (prn
(string_of_int (int (head (filter fx x)))));;", [], "4") ;
 ("Testing stdlib function: append", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (string (head (tail
(tail (tail (tail(append '(\"1\" \"2\" \"3\") '(\"1\" \"LAST\")))))))));;", [], "LAST") ;
 ("Testing stdlib function: take", "(= x '(1 2 3 4 5 6 7 8 9 10));; (print_list format_int (list
(take 5 x)));; ", [], "[1,2,3,4,5]") ;
 ("Testing stdlib function: drop", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (print_list format_int (list
(drop 5 x)));;", [], "[6,7,8,9,10]") ;
 ("Testing stdlib function: zipwith", "(print_list format_int (zipwith (fn (x y) (+ x y)) '(1 2 3
4) '(5 6 7 8)));;", [], "[6,8,10,12]") ;
 ("Testing stdlib function: zipwith3", "(print_list format_int (zipwith3 (fn (x y z) (+ x y z)) '(1
2 3 4) '(5 6 7 8) '(10 11 12 13)));;", [], "[16,19,22,25]") ;
 ("Testing stdlib function: reverse", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (print_list format_int
(reverse x));;", [], "[10,9,8,7,6,5,4,3,2,1]") ;
 ("Testing stdlib function: member", "(= x '(1 2 3 4 5 6 7 8 9 10));; (prn (string_of_boolean
(member 10 x)));; ", [], "true") ;
 ("Testing stdlib function: intersperse", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (print_list format_int
(intersperse 0 x));;", [], "[1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9,0,10]") ;
 ("Testing stdlib function: stringify_list", "(= x '(1 2 3 4 5 6 7 8 9 10));;  (prn (stringify_list
format_int x));;", [], "[1,2,3,4,5,6,7,8,9,10]")

] ;;

let unsuccess = ref 0 ;;
```

```
List.iter (fun (desc, input, ast, expout) ->
  let stdlib = load_file "stdlib.ss" in
  let lexbuf = Lexing.from_string (stdlib ^ input) in
  try
    let expression = Parser.program Scanner.token lexbuf in
    if (ast = expression || ast = []) then
      let prog = Generator.generate_prog expression in
      write prog;
      let actout = String.concat "\n" (funct (Unix.open_process_in "node a.js")) in
      if  expout = actout then print_string "" else
       print_endline (String.concat "" ["\027[38;5;1m"; desc; ": "; input; "...
UNSUCCESSFUL Compilation....\ninput: "; input; "\nexpected out: "; expout; "\nActual
out: "; actout; "\027[0m"]);
    else (print_endline (String.concat "" ["\027[38;5;1m"; desc; ": "; input; "\027[0m"]) ;
unsuccess := !unsuccess+1 ) ;
  with
    | _ -> print_endline (String.concat "" ["**START REPORT**\n" ; "Parse
Error:\ninput:";input ;"\n**END REPORT**"]) ; unsuccess := !unsuccess+1) tests ;;


if !unsuccess = 0 then print_endline "ALL SUCCESSFUL" else exit 1 ;;
```

generator.ml

```
open Ast;;
open Printf;;

let sprintf = Printf.sprintf;;

let cc l = String.concat "" l;;
let box t v = sprintf "({ __t: '%s', __v: %s })" t v;;

let generate_js_func fname =
  let helper fname =
    match fname with
      "prn" ->
      ("function() {\
          for(var i=0; i < arguments.length; i++) {\
            __assert_type(\\'string\\', arguments[i], \\'prn\\', (i+1) + \\'\\');\
            console.log(__unbox(arguments[i]));\
          }\
          return __box(\\'unit\\', 0);\
        }", [TString], TUnit, [])
    | "exec" ->
      ("'function() {\
          var res;\
          for(var i = 0; i < arguments.length; i++) { \
            __assert_type(\\'list\\', arguments[i], \\'do\\', (i+1) + \\'\\'); \
            var str = JSON.stringify(__unbox(arguments[i]).slice(1)); \
            res = eval(\\'(\\' + __unbox(__unbox(arguments[i])[0]) + \\').apply(null, \\' + str +
  \\')\\'); \
          } \
          return res; \
        }'", [TSomeList(TSome)], TSome, ["evaluate"])

    | "pr" ->
      ("'function(s) { \
          for(var i=0; i < arguments.length; i++) {\
            __assert_type(\\'string\\', arguments[i], \\'pr\\', (i+1) + \\'\\'); \
            process.stdout.write(__unbox(arguments[i]));\
          } \
          return __box(\\'unit\\', 0); \
        }'", [TString], TUnit, [])

    | "type" ->
      ("'function(o) { \
          __assert_arguments_num(arguments.length, 1, \\'type\\'); \
          return __box(\\'string\\', o.__t); \
        }'", [TSome], TString, [])

    | "head" ->
      ("'function(l) { \
          __assert_arguments_num(arguments.length, 1, \\'head\\'); \
          __assert_type(\\'list\\', l, \\'head\\', \\'1\\'); \
          return __clone(__unbox(l)[0]); \
        }'", [TSomeList(TSome)], TSome, [])

    | "tail" ->
      ("'function(l) { \
          __assert_arguments_num(arguments.length, 1, \\'tail\\'); \
          __assert_type(\\'list\\', l, \\'tail\\', \\'1\\'); \
          return __box(\\'list\\', __unbox(l).slice(1)); \
        }'", [TSomeList(TSome)], TSomeList(TSome), [])

    | "cons" ->
      ("'function(i, l) { \
          __assert_arguments_num(arguments.length, 2, \\'cons\\'); \
          __assert_type(\\'list\\', l, \\'cons\\', \\'2\\'); \
          var __temp = __unbox(l); \
          __temp.unshift(__clone(i)); \
          return __box(\\'list\\', __temp); \
        }'", [TSome; TSomeList(TSome)], TSomeList(TSome), [])

    | "__add" ->
      ("'function() { \
          for(var i=0; i < arguments.length; i++) { \
            __assert_type(\\'int\\', arguments[i], \\'+\\', (i+1) + \\'\\');\
          } \
          return __box(\\'int\\', arguments.length === 0 ? 0 : \
            Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return
a+b;}));\
        }'", [TInt; TInt], TInt, [])

    | "__sub" ->
      ("'function(a1) { \
          for(var i=0; i < arguments.length; i++) { \
            __assert_type(\\'int\\', arguments[i], \\'-\\', (i+1) + \\'\\');\
          } \
          return __box(\\'int\\', arguments.length === 0 ? 0 : arguments.length === 1 ? -1 *
__unbox(a1) : \
            Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return
a-b;}));\
        }'", [TInt; TInt], TInt, [])

    | "__mult" ->
      ("'function() { \
          for(var i=0; i < arguments.length; i++) { \
            __assert_type(\\'int\\', arguments[i], \\'*\\', (i+1) + \\'\\');\
          } \
          return __box(\\'int\\', arguments.length === 0 ? 1 : \
            Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return
a*b;})); \
        }'", [TInt; TInt], TInt, [])

    | "__div" ->
      ("'function() { \
          for(var i=0; i < arguments.length; i++) { \
            __assert_type(\\'int\\', arguments[i], \\'/\\', (i+1) + \\'\\');\
          } \
          return __box(\\'int\\', \

Math.floor(Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){retu
rn a/b;}))); \
      }'", [TInt; TInt], TInt, [])

   | "mod" ->
     ("'function(a1, a2) { \
        for(var i=0; i < arguments.length; i++) { \
          __assert_arguments_num(arguments.length, 2, \\'mod\\'); \
          __assert_type(\\'int\\', arguments[i], \\'mod\\', (i+1) + \\'\\');\
        }  \
        return __box(\\'int\\', __unbox(a1) % __unbox(a2)); \
        }'", [TInt; TInt], TInt, [])

   | "__addf" ->
     ("'function(a1) { \
        for(var i=0; i < arguments.length; i++) { \
          __assert_type(\\'float\\', arguments[i], \\'+.\\', (i+1) + \\'\\');\
        }  \
        return __box(\\'float\\', arguments.length === 0 ? 0 : \
          Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return
a+b;})); \
      }'", [TFloat; TFloat], TFloat, [])

   | "__subf" ->
     ("'function(a1) { \
        for(var i=0; i < arguments.length; i++) { \
          __assert_type(\\'float\\', arguments[i], \\'-.\\', (i+1) + \\'\\');\
        } \
        return __box(\\'float\\', arguments.length === 0 ? 0 : arguments.length === 1 ? -1 *
__unbox(a1) : \
          Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return
a-b;})); \
      }'", [TFloat; TFloat], TFloat, [])

   | "__multf" ->
     ("'function(a1) { \
        for(var i=0; i < arguments.length; i++) { \
          __assert_type(\\'float\\', arguments[i], \\'*.\\', (i+1) + \\'\\');\
        } \
        return __box(\\'float\\',
Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return a*b;})); \
        }'", [TFloat; TFloat], TFloat, [])

   | "__divf" ->
     ("'function(a1, a2) { \
        for(var i=0; i < arguments.length; i++) { \
          __assert_type(\\'float\\', arguments[i], \\'/.\\', (i+1) + \\'\\');\
        } \
        return __box(\\'float\\',
Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return a/b;})); \
      }'", [TFloat; TFloat], TFloat, [])

    | "__equal" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'is\\'); \
        return __box(\\'boolean\\', JSON.stringify(__unbox(a1)) ===
JSON.stringify(__unbox(a2))); \
      }'", [TParam 1; TParam 1], TBool, [])

    | "__neq" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'isnt\\'); \
        if (a1.__t !== a2.__t) { \
          throw new TypeError(\\'expected arguments of function isnt to be the same, \
            but found \\' + a1.__t + \\' and \\' + a2.__t +\\'.\\'); \
        } \
        return __box(\\'boolean\\', __unbox(a1) !== __unbox(a2)); \
      }'", [TParam 1; TParam 1], TBool, [])

    | "__less" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'<\\'); \
        if (a1.__t !== a2.__t) { \
          throw new TypeError(\\'expected arguments of function < to be the same, \
            but found \\' + a1.__t + \\' and \\' + a2.__t +\\'.\\'); \
        } \
        return __box(\\'boolean\\', __unbox(a1) < __unbox(a2)); \
      }'", [TParam 1; TParam 1], TBool, [])

    | "__leq" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'<=\\'); \
        if (a1.__t !== a2.__t) { \
          throw new TypeError(\\'expected arguments of function <= to be the same, \
            but found \\' + a1.__t + \\' and \\' + a2.__t +\\'.\\'); \
        } \
        return __box(\\'boolean\\', __unbox(a1) <= __unbox(a2)); \
      }'",[TParam 1; TParam 1], TBool, [])

    | "__greater" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'>\\'); \
        return __box(\\'boolean\\', __unbox(a1) > __unbox(a2)); \
      }'", [TParam 1; TParam 1], TBool, [])

  | "__geq" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'<=\\'); \
        if (a1.__t !== a2.__t) { \
          throw new TypeError(\\'expected arguments of function >= to be the same, \
            but found \\' + a1.__t + \\' and \\' + a2.__t +\\'.\\'); \
        } \
        return __box(\\'boolean\\', __unbox(a1) >= __unbox(a2)); \
      }'", [TParam 1; TParam 1], TBool, [])

  | "__and" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'and\\'); \
        __assert_type(\\'boolean\\', a1, \\'and\\', \\'1\\'); \
        __assert_type(\\'boolean\\', a2, \\'and\\', \\'2\\'); \
        return __box(\\'boolean\\', __unbox(a1) && __unbox(a2)); \
      }'", [TBool; TBool], TBool, [])

  | "__or" ->
      ("'function(a1, a2) { \
        __assert_arguments_num(arguments.length, 2, \\'or\\'); \
        __assert_type(\\'boolean\\', a1, \\'or\\', \\'1\\'); \
        __assert_type(\\'boolean\\', a2, \\'or\\', \\'2\\'); \
        return __box(\\'boolean\\', __unbox(a1) || __unbox(a2)); \
      }'", [TBool; TBool], TBool, [])

  | "__not" ->
      ("'function(a) { \
        __assert_arguments_num(arguments.length, 1, \\'not\\'); \
        __assert_type(\\'boolean\\', a, \\'not\\', \\'1\\'); \
        return __box(\\'boolean\\', !__unbox(a)); \
      }'", [TBool], TBool, [])

    | "string_of_int" ->
      ("'function(i) { \
        __assert_arguments_num(arguments.length, 1, \\'string_of_int\\'); \
        __assert_type(\\'int\\', i, \\'string_of_int\\', \\'1\\'); \
        return __box(\\'string\\', \\'\\' + __unbox(i)); \
      }'", [TInt], TString, [])

    | "int_of_string" ->
    ("'function(s) { \
      __assert_arguments_num(arguments.length, 1, \\'int_of_string\\'); \
      __assert_type(\\'string\\', s, \\'int_of_string\\', \\'1\\'); \
      return __box(\\'int\\', parseInt(__unbox(s))); \
      }'", [TString], TInt, [])

  | "string_of_float" ->
    ("'function(f) { \
      __assert_arguments_num(arguments.length, 1, \\'string_of_float\\'); \
      __assert_type(\\'float\\', f, \\'string_of_float\\', \\' 1\\'); \
      return __box(\\'string\\', \\'\\' + __unbox(f)); \
      }'", [TFloat], TString, [])

  | "float_of_string" ->
    ("'function(s) { \
      __assert_arguments_num(arguments.length, 1, \\'float_of_string\\'); \
      __assert_type(\\'string\\', s, \\'float_of_string\\', \\'1\\'); \
      return __box(\\'float\\', parseFloat(__unbox(s))); \
      }'", [TString], TFloat, [])

  | "string_of_boolean" ->
    ("'function(b) { \
      __assert_arguments_num(arguments.length, 1, \\'string_of_boolean\\'); \
      __assert_type(\\'boolean\\', b, \\'string_of_boolean\\', \\'1\\'); \
      return __box(\\'string\\', \\'\\' + __unbox(b)); \
      }'", [TBool], TString, [])

  | "__concat" ->
    ("'function() { \
      for(var i=0; i < arguments.length; i++) { \
        __assert_type(\\'string\\', arguments[i], \\'++\\', (i+1) + \\'\\');\
      } \
      return __box(\\'string\\', arguments.length === 0 ? \\'\\' : \
        Array.prototype.slice.call(arguments).map(__unbox).reduce(function(a,b){return
a+b;})); \
      }'", [TString; TString], TString, [])

  | "evaluate" ->
    ("'function(l) { \
      __assert_arguments_num(arguments.length, 1, \\'eval\\'); \
      __assert_type(\\'list\\', l, \\'eval\\', \\'1\\');\
      return eval(\\'(\\' + __unbox(__unbox(l)[0]) + \\').apply(null, \\' +
JSON.stringify(__unbox(l).slice(1)) + \\')\\'); \
      }'", [TSomeList(TSome)], TSome, [])
  | "module" ->
    ("'function(n) { \
      __assert_arguments_num(arguments.length, 1, \\'module\\'); \
      __assert_type(\\'string\\', n, \\'module\\', \\'1\\'); \
      var __t = require(__unbox(n));\
      return __box(\\'module\\', __t, __t); \
      }'", [TString], TJblob, [])

  | "list" ->
    ("'function(i) { \
      __assert_arguments_num(arguments.length, 1, \\'list\\'); \
      if (__assert_type(\\'list\\', i, \\'list\\')) { \
        return i; \
      }\
      }'", [TSome], TSomeList(TSome), ["type"])

  | "int" ->
    ("'function(i) { \
      __assert_arguments_num(arguments.length, 1, \\'int\\'); \
      if (__assert_type(\\'int\\', i, \\'int\\')) { \
        return i;\
      }\
      }'", [TSome], TInt, ["type"])

  | "string" ->
    ("'function(i) { \
      __assert_arguments_num(arguments.length, 1, \\'string\\'); \
      if (__assert_type(\\'string\\', i, \\'string\\')) { \
        return i; \
      } \
      }'", [TSome], TString, ["type"])

  | "float" ->
    ("'function(i) { \
      __assert_arguments_num(arguments.length, 1, \\'float\\'); \
      if (__assert_type(\\'float\\', i, \\'float\\')) { \
        return i; \
      } \
      }'", [TSome], TFloat, ["type"])

  | "boolean" ->
    ("'function(i) { \
      __assert_arguments_num(arguments.length, 1, \\'boolean\\'); \
      if (__assert_type(\\'boolean\\', i, \\'boolean\\')) { \
        return i; \
        } \
      }''', [TSome], TBool, ["type"])

    | "exception" ->
      ('''function(i) { \
        __assert_arguments_num(arguments.length, 1, \\'exception\\'); \
        __assert_type(\\'string\\', i, \\'exception\\', \\'1\\'); \
        throw new Error(i); \
      }''', [TString], TUnit, [])

    | _ -> ("", [], TSome, [])
  in
    let (fstr, arg_types, ret_type, deps) = helper fname in
    (box "function" fstr, arg_types, ret_type, deps)

let is_generatable fname =
  let (_, argtypes, _, _) = generate_js_func fname in
  argtypes != []

let get_generatable_fnames prog =
  let rec get_deps fname =
    let (_, _, _, deps) = generate_js_func fname in
    deps @ (List.flatten (List.map get_deps deps)) in
  let rec get_fnames e = match e with
      Eval(f, el) -> (match f with
                        Id(x) -> [x]
                      | Fdecl(x, y) -> get_fnames y
                      | Eval(x, y) -> get_fnames (Eval(x, y))
                      | If(c, t, e) -> get_fnames (If(c, t, e))
                      | _ -> []) @ (get_fnames (List(el)))
    | Id(s) -> [s]
    | Assign(el) -> get_fnames (List(el))
    | List(el) -> List.flatten (List.map get_fnames el)
    | Fdecl(argl, exp) -> get_fnames exp
    | If(cond, thenb, elseb) -> (get_fnames cond) @ (get_fnames thenb) @ (get_fnames
elseb)
    | _ -> [] in
  let generatable = (List.filter is_generatable (List.flatten (List.map get_fnames prog))) in
  let dependencies = List.map get_deps generatable in
  List.sort_uniq compare (generatable @ (List.flatten dependencies))

let arrow_of fname =
  let (_, arg_types, ret_type, _) = generate_js_func fname in
  match arg_types, ret_type with
  | [t1], t2 -> TArrow([t1; t2])
```

```
 | [t1; t2], t3 -> TArrow([t1; t2; t3])
 | _ -> raise(Failure ("Unknown identifier: '" ^ fname ^ "' "))

let generate_prog p =
 let escape_quotes s =
   Str.global_replace (Str.regexp "\\([^\\\\]?\\)'") "\\1\\'" (Str.global_replace (Str.regexp
"\\([\\\\]+\\)'") "\\1\\1'" s) in
 let rec generate e =
   match e with
     Nil -> box "list" "[]"
   | List(el) -> box "list" (sprintf "[%s]" (String.concat ", " (List.map generate el)))
   | Int(i) -> box "int" (string_of_int (i))
   | Float(f) -> box "float" (string_of_float (f))
   | Boolean(b) -> box "boolean" (if b = true then "true" else "false")
   | String(s) -> box "string" (sprintf "'%s'" s)
   | Id(s) -> sprintf "eval('%s')" s

   | Assign(el) -> let rec gen_pairs l =
                    match l with
                      [] -> []
                    | h1::h2::tl -> (h1, h2)::(gen_pairs tl)
                    | _::[] -> raise (Failure("= operator used on odd numbered list!"))
                   in
                    sprintf "eval('%s')" (cc (List.map
                                   (function
                                    | (Id(s), e) -> sprintf "var %s = %s; %s;" s (escape_quotes
(generate e)) s
                                    | _ -> raise (Failure "can only assign to identifier"))
                                   (gen_pairs el)))

   | Eval(first, el) ->
       let argl = generate (List(el)) in
       (match first with
         Id(x) -> (match x with
                    "dot" -> sprintf "__dot(%s)" argl
                   | "call" -> sprintf "__call(%s)" argl
                   | _ -> sprintf "(function(_i, _a) { \
                              return _i.__t === 'module' ? __box('module', __unbox(_i).apply(null,
__unbox(_a).map(__unbox))) : \
                               eval('(' + __unbox(_i) + ').apply(null, ' +
JSON.stringify(__unbox(_a)) + ')'); \
                             })\
                             (eval('%s'), %s)" x argl)

       | x -> sprintf
```

```
            "eval('(' + __unbox(%s) + ').apply(null, ' + JSON.stringify(__unbox(%s)) + ')')"
            (match x with
              Fdecl(a, e) -> generate (Fdecl(a, e))
            | Eval(f, e) -> generate (Eval(f, e))
            | If(c, t, e) -> generate (If(c, t, e))
            | _ -> raise (Failure "foo"))
            argl)

   | Fdecl(argl, exp) -> box "function"
       (sprintf
         "(function(%s) { return %s; }).toString()"
         (String.concat ", " argl)
         (generate exp))

   | If(cond, thenb, elseb) ->
       sprintf
         "(function() { var __c = __unbox(%s); return !(Array.isArray(__c) && __c.length ===
0) && __c ? %s : %s; })()"
         (generate cond)
         (generate thenb)
         (generate elseb)

  in
  let generate_head p =
    let get_def fname =
      let (body, _, _, _) = generate_js_func fname in
      cc ["var "; fname; "="; body] in
    String.concat ";\n" (List.map get_def (get_generatable_fnames p)) in
  let wrap_exp e = cc [generate e; ";"] in
  cc (
      "
      try {
      function getOwnPropertyDescriptors(object) {\
        var keys = Object.getOwnPropertyNames(object), returnObj = {}; \
        keys.forEach(getPropertyDescriptor); \
        return returnObj; \
        function getPropertyDescriptor(key) { \
          var pd = Object.getOwnPropertyDescriptor(object, key); \
          returnObj[key] = pd; \
        } }"::

      "Object.getOwnPropertyDescriptors = getOwnPropertyDescriptors;"::

      "function __dup(o) { \
        return Object.create(Object.getPrototypeOf(o),
Object.getOwnPropertyDescriptors(o)); \
   }"::

   "function __flatten(o) { \
     var result = Object.create(o); \
     for(var key in result) { result[key] = result[key]; } \
     return result; \
   }"::

   "function __box(t,v,c){ \
     return ({ __t: t, __v: (t === 'module') ? v : __clone(v), _ctxt: c }); \
   };"::

   "function __clone(o){\
     return JSON.parse(JSON.stringify(o));\
   };"::

   "function __unbox(o){ \
     return (o.__t === 'module') ? o.__v : __clone(o.__v); \
   };"::

   "function __assert_type(t, o, f, nth) { \
     if (o.__t !== t && o.__t !== 'json') { \
       throw new TypeError('expected type of argument ' + nth + ' of function ' + f + \
         ' to be ' + t + ' but found ' + o.__t); \
     } else { \
       return true; \
     }\
   }"::

   "function __assert_arguments_num(actual, expected, f) { \
     if (actual !== expected) { \
       throw new TypeError('expected ' + expected + ' arguments to function ' + f + \
         ' but found ' + actual + ' arguments. '); \
     } else { \
       return true; \
     } \
   };"::

   "function __tojs(o) { \
     if (o.__t === 'function') { \
       return function() { \
         var __temp = Array.prototype.slice.call(arguments); \
         return eval('(' + o.__v + ').apply(null, __temp)'); \
       }; \
     } else { \
       return __unbox(o); \
     } \
   };"::

   "function __call(args) { if(args.__v.length === 1) { return __box('module',
args.__v[0].__v(), args.__v[0].__v()); }; var __temp = !args.__v[0].__t ? args.__v[0] :
__unbox(args.__v[0]); __temp[__unbox(args.__v[1])].apply(__temp,
args.__v.slice(2).map(__tojs)); };"::

   "function __dot(args) { var __temp = args.__v; var res; for(var i = 1; i < __temp.length;
i++) { res = (i === 1 ? __temp[0] : res)[__unbox(__temp[i])]; } return __box('json', res); };"::

   (generate_head p)::";\n"::(List.map wrap_exp p)@["}\
   catch (e) {\
     console.log('Runtime Error: ' + e.message);\
   };"])
```

message.ml

```
(** Error messages for Lexing & Parsing *)

open Printf
open Lexing
open Parsing

exception LexingErr of string

let print_position lexbuf msg =
  let start = lexeme_start_p lexbuf in
   let finish = lexeme_end_p lexbuf in
    (fprintf stderr "Line %d: char %d..%d: %s: \"%s\" \n"
          start.pos_lnum
          (start.pos_cnum - start.pos_bol)
          (finish.pos_cnum - finish.pos_bol)
          msg
          (Lexing.lexeme lexbuf))

(** [lexer_from_channel fname ch] returns a lexer stream which takes
    input from channel [ch]. The input filename (for reporting errors) is
    set to [fname].
*)
let lexer_from_channel fname ch =
  let lex = Lexing.from_channel ch in
  let pos = lex.lex_curr_p in
```

```
    lex.lex_curr_p <- { pos with pos_fname = fname; pos_lnum = 1; } ;
    lex

(** [lexer_from_string str] returns a lexer stream which takes input
    from a string [str]. The input filename (for reporting errors) is set to
    [""]. *)
let lexer_from_string str =
  let lex = Lexing.from_string str in
  let pos = lex.lex_curr_p in
    lex.lex_curr_p <- { pos with pos_fname = ""; pos_lnum = 1; } ;
    lex
```

parser.mly

```
%{
  open Ast
  open Lexing
  open Parsing

  let num_errors = ref 0

  let parse_error msg = (* called by parser function on error *)
          let start = symbol_start_pos() in
          let final = symbol_end_pos() in
          Printf.fprintf stdout "Line:%d char:%d..%d: %s\n"
                      (start.pos_lnum) (start.pos_cnum - start.pos_bol) (final.pos_cnum -
final.pos_bol) msg;
        incr num_errors;
          flush stdout

%}

%token PLUS MINUS TIMES DIVIDE PLUSF MINUSF TIMESF DIVIDEF EOF
%token ASSIGN QUOTE AND OR NOT EQ NEQ LT LEQ GT GEQ CONCAT
%token SEMI LPAREN RPAREN LBRACE RBRACE
%token <int> INT
%token FUNC IF DO EVAL
%token <string> ID
%token <string> STRING
%token <float> FLOAT
%token <bool> BOOL
%token NIL

%right ASSIGN
```

```
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS PLUSF MINUSF
%left TIMES DIVIDE TIMESF DIVIDEF
%left CONCAT

%start program
%type <Ast.program> program

%%

program:
 expr_list EOF                     { if(!num_errors == 0) then (List.rev $1)
                             else (exit (-1)) }

expr_list:
/* nothing */              { [] }
| expr_list expr SEMI{ $2 :: $1 }
| expr_list expr          { (parse_error ("Syntax error. Did you forget to terminate the
expression with ;; " ^
                                 "or forget a right paren?")); $1 }

sexpr:
| IF expr expr expr              { If($2, $3, $4) }
| FUNC LPAREN formals_opt RPAREN expr { Fdecl(List.rev $3, $5) }
| call                           { $1 }

expr:
  atom                           { $1 }
| list                           { $1 }
| LBRACE infix_expr RBRACE       { $2 }
| unquoted_list                  { $1 }

list:
  QUOTE LPAREN args_opt RPAREN       { List(List.rev $3) }

unquoted_list:
  LPAREN sexpr RPAREN            { $2 }
| LPAREN sexpr SEMI              { (parse_error "Syntax error. Left paren is unmatched
by right paren."); $2 }

formals_opt:
/* nothing */   { [] }
```

```
| formal_list   { $1 }

 formal_list:
  ID                { [$1] }
| formal_list ID  { $2 :: $1 }

atom:
  constant                   { $1 }
| ID                         { Id($1) }
| NIL                        { Nil }
| operator                   { Id($1) }
| two_args_operators             { Id($1) }

operator:
| PLUS                       { "__add" }
| MINUS                          { "__sub" }
| TIMES                          { "__mult" }
| DIVIDE                     { "__div" }
| PLUSF                          { "__addf" }
| MINUSF                     { "__subf" }
| DIVIDEF                    { "__divf" }
| TIMESF                     { "__multf" }
| AND                        { "__and" }
| OR                         { "__or" }
| NOT                        { "__not" }
| CONCAT                     { "__concat" }
| DO                         { "exec" }
| EVAL                       { "evaluate" }

two_args_operators:
| EQ     { "__equal" }
| NEQ     { "__neq" }
| LT     { "__less" }
| LEQ     { "__leq" }
| GT     { "__greater" }
| GEQ     { "__geq" }

constant:
  INT                        { Int($1) }
| FLOAT                          { Float($1) }
| BOOL                           { Boolean($1) }
| STRING                     { String($1) }

call:
  ID args_opt                    { Eval(Id($1), List.rev $2) }
```

```
| unquoted_list args_opt     { Eval($1, List.rev $2) }
| operator args_opt          { Eval(Id($1), List.rev $2) }
| two_args_operators two_args     { Eval(Id($1), List.rev $2)}
| ASSIGN assign_args               { Assign($2) }
| constant args_opt          { (parse_error "Syntax error. Function call on inappropriate
object."); $1 }
;

args_opt:
/* nothing */           { [] }
| args                  { $1 }

two_args:
expr expr { [$2; $1] }

args:
  expr                  { [$1] }
| args expr             { $2 :: $1 }

assign_args:
  ID expr               { [Id($1); $2] }
| ID expr assign_args   { Id($1) :: $2 :: $3 }
| error                 { (parse_error "Syntax error. Assign usage is (= id1 e1 id2 e2...idn
en)"); [] }

infix_expr:
  constant                      { $1 }
| LPAREN ID args_opt RPAREN     { Eval(Id($2), List.rev $3) }
| ID                            { Id($1) }
| MINUS INT                     { Int(-1 * $2) }
| MINUS FLOAT                       { Float(-1.0 *. $2) }
| LBRACE infix_expr RBRACE      { $2 }
| ID ASSIGN infix_expr              { Assign([Id($1); $3]) }
| infix_expr CONCAT infix_expr  { Eval(Id("concat"), [$1; $3]) }
| infix_expr PLUS infix_expr        { Eval(Id("__add"), [$1; $3]) }
| infix_expr MINUS infix_expr       { Eval(Id("__sub"), [$1; $3])  }
| infix_expr TIMES infix_expr       { Eval(Id("__mult"), [$1; $3]) }
| infix_expr DIVIDE infix_expr      { Eval(Id("__div"), [$1; $3])  }
| infix_expr PLUSF infix_expr       { Eval(Id("__addf"), [$1; $3]) }
| infix_expr MINUSF infix_expr      { Eval(Id("__subf"), [$1; $3]) }
| infix_expr TIMESF infix_expr      { Eval(Id("__multf"), [$1; $3]) }
| infix_expr DIVIDEF infix_expr     { Eval(Id("__divf"), [$1; $3]) }
| infix_expr EQ infix_expr     { Eval(Id("__equal"), [$1; $3]) }
| infix_expr NEQ infix_expr { Eval(Id("__neq"), [$1; $3]) }
| infix_expr LT infix_expr     { Eval(Id("__less"), [$1; $3]) }
| infix_expr LEQ infix_expr  { Eval(Id("__leq"), [$1; $3]) }
| infix_expr GT infix_expr    { Eval(Id("__greater"), [$1; $3]) }
| infix_expr GEQ infix_expr { Eval(Id("__geq"), [$1; $3]) }
| infix_expr AND infix_expr { Eval(Id("__and"), [$1; $3]) }
| infix_expr OR infix_expr    { Eval(Id("__or"), [$1; $3]) }
```

preprocessor.mll

```
{

type token = EOF
             | String of string   (* "sss" *)
             | Fparen of string          (* foo(a) *)
             | Fdecl of string    (* fn(a b) a + b;; def foo(a) a + 1 *)
             | Word of string     (* other text in program *)
             | Comment                        (* comments look just like this *)
             | StdFn of string           (* if a
                                             b
                                             c *)
             | LineBreak                 (* \n *)

(* Store the indentation of added left parens using a stack *)
let curIndent = Stack.create();;
Stack.push (-1) curIndent;;  (* mark bottom of stack with -1 *)

(* Counts continuous whitespace in string s, starting at the given index and count *)
let rec countSp s count index =
          let len = (String.length s - 1) in
                  if index > len then count
                          else if String.get s index = ' ' then countSp s (count+1)(index+1)
                          else if String.get s index = '\t' then countSp s (count+8)(index+1)
                  else count

(* At the end of each expression (after seeing a newline):
   Add  one right paren for each element popped off the given stack,
   until the stop of stack is -1; Adding to the end of given string s *)
let rec closeExpression s stack =
          let top = Stack.top stack in
                  if (top == -1) then (s ^ ";;\n")  (* keep -1 as bottom-of-stack marker *)
                  else (ignore (Stack.pop stack); closeExpression (")" ^ s) stack)
}
let fn_name = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' '-']*
let binop =  "+" | "-" | "*" | "/" | "+." | "-." | "*." | "/."
             | "and" | "or" | "is" | "isnt" | ">" | "<" | ">=" | "<=" | "="
```

```
rule token = parse
         eof { EOF }

        (* Newline marks end of expression; Add R-parens & ";;"*)
        | "\n" { let parens =
                        if (Stack.top curIndent == -2) then "\n"
                        else (closeExpression "" curIndent)
                        in Word(parens) }

        | "/*" { comment lexbuf }

        | '\n'[' ' '\t']*("let" | "if" | "do" | "eval" ) as lxm       (* Standard library functions *)
                                { let spaces = countSp lxm 0 1 in
                                  ignore(Stack.push spaces curIndent);
                                 StdFn(lxm) }
        | '\n'[' ' '\t']* as ws {                  (* Indentation: whitespace at the beginning of a
line *)
                            let spaces = countSp ws 0 1 in
                            if(spaces > Stack.top curIndent) then ( Word(ws))
                            else (
                                (* Same or less indentation than previously added
lparen:
                                        pop stack until current indentation = top of
stack *)
                                let parens =
                                let rec closeParens s stack =
                                        let top = Stack.top curIndent in
                                        if (top == -1) then (s ^ ";;\n")
                                        else if (top >= spaces)
                                            then (ignore(Stack.pop stack); closeParens
(")" ^ s) stack)
                                        else (* top > spaces *) s
                                in closeParens ws curIndent
                                in Word(parens)
                            ) }

        | "fn" ' '* '('     as lxm { let spaces = countSp lxm 0 1 in   (* Anonymous function
declaration: "fn (" *)
                            ignore(Stack.push spaces curIndent); Fdecl(lxm) }

        | '\n'[' ' '\t']* "def" ' '* fn_name ' '* '('          (* Named function declaration: needs
own regexp
                                                so that fn_name(args) does not get
slurped into (fn_name args) *)
                            as lxm { let spaces = countSp lxm 0 1 in
```

```
                                ignore(Stack.push spaces curIndent); Fdecl(lxm) }

        | (fn_name | binop)' '* '(' as lxm { Fparen(lxm) }    (* Function call as "f(args)" *)

        | '\"'[^'\"']*'\"' as lxm { String(lxm) }          (* Quoted strings: quoted function calls
scan as strings,
                                                            not evaluated as fn calls *)

        | _ as lxm { Word(String.make 1 lxm) }              (* All characters other than the
above *)

and comment = parse
        "*/"     { token lexbuf }        (* Return to normal scanning *)
        | _      { comment lexbuf }  (* Ignore other characters *)
{
        let () =
        let infile = Sys.argv.(1) in
        let outfile =
                let endname = try (String.index infile '.')
                                                    with Not_found -> (String.length infile - 1) in
                String.sub infile 0 endname ^ ".ss"
        in
        let oc = open_out outfile in

        let load_file f =
                let ic = open_in f in
                let n = in_channel_length ic in
                let s = Bytes.create n in
                really_input ic s 0 n;
                close_in ic;
                (s)
        in
        let lexbuf = Lexing.from_string ("\n\n" ^ (load_file infile)) in

        let wordlist =
          let rec next l  = match token lexbuf with
                EOF -> l
                | String(s) -> next(s::l)
                | Comment -> next(l)
                | LineBreak -> next("\n" :: l)
                | Word(s) -> next(s::l)
                | Fdecl(s) -> next( ("\n(" ^ s) :: l)
                | StdFn(s) -> next( ("\n(" ^ s) :: l)
                | Fparen(s) ->
                        let args = String.index s '(' + 1 in
                              let s = "(" ^ (String.sub s 0 (args - 1)) ^ " " in
                                        next(s::l)
                    in next []
          in let program = String.concat "" (List.rev wordlist)
          in let lines = Str.split (Str.regexp "\n") program

          in ignore(List.iter (fun a -> if a <> ";;" then print_endline(a)) lines);
          (List.iter (fun a -> if a <> ";;" then (Printf.fprintf oc "%s\n" a)) lines);
          close_out oc;
}
```

scanner.mll

```
{
  open Parser
  open Message
  open Lexing
}

rule token = parse

| "###ENDSTDLIB###"        { ignore
                                  (lexbuf.lex_curr_p <- {(lexeme_start_p lexbuf)
                                      with  pos_lnum = 0 ; });
                                  token lexbuf }
| [' ' '\t' '\\'] { token lexbuf } (* Whitespace *)
| [ '\n' '\r' ]                   { ignore(Lexing.new_line lexbuf); token lexbuf }
| "/*"      { comment lexbuf }      (* Comments *)
| ";;"     { SEMI }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| "+."     { PLUSF }
| "-."     { MINUSF }
| "*."     { TIMESF }
| "/."     { DIVIDEF }
| "and"             { AND }
| "or"     { OR }
| "not"    { NOT }
| '\"'    { QUOTE }
| '='     { ASSIGN }
| "is"    { EQ }
| "isnt"   { NEQ }
| "true"   as lxm { BOOL(bool_of_string lxm) }
| "false"  as lxm { BOOL(bool_of_string lxm) }
| "nil"    { NIL }
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| "++"    { CONCAT }
| "fn"    { FUNC }
| "if"    { IF }
| "do"    { DO }
| "eval"   { EVAL }
| "evaluate"  { raise(Failure "Lexer error: evaluate is a reserved keyword and may not be
used. ") }
| "exec"     { raise(Failure "Lexer error: exec is a reserved keyword and may not be used.
") }
| '\"'[^'\"']*'\"' as lxm { STRING(String.sub lxm 1 (String.length lxm - 2)) }       (* String *)
| ['0'-'9']*'.'['0'-'9']+  as lxm { FLOAT(float_of_string lxm) }                  (* Float *)
| ['0'-'9']+'.'['0'-'9']*  as lxm { FLOAT(float_of_string lxm) }                  (* Float *)
| ['0'-'9']+ as lxm { INT(int_of_string lxm) }                                   (* Int *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }            (* Identifier *)
| eof { EOF }
| _       { raise (Message.LexingErr("Illegal input")) }

and comment = parse
  "*/"  { token lexbuf } (* comments *)
  | _   { comment lexbuf }
```

stringify.ml

```
open Ast

let rec stringify e =
 let stringify_op o = match o with
   Add -> "Add" | Sub -> "Sub" | Mult -> "Mult" | Div -> "Div"
   | Addf -> "Addf" | Subf -> "Subf" | Multf -> "Multf" | Divf -> "Divf"
   | Equal -> "Equal" | Neq -> "Neq" | Less -> "Less" | Leq -> "Leq"
   | Greater -> "Greater" | Geq -> "Geq" | And -> "And" | Or -> "Or"
   | Assign -> "Assign"
in
 let concat l = (String.concat "" l) in
```

```
  match e with
    Int(x) -> concat ["Int("; string_of_int x; ")"]
  | Float(x) -> concat ["Float("; string_of_float x; ")"]
  | Boolean(x) -> concat ["Boolean("; if x = true then "true" else "false"; ")"]
  | String(x) -> concat ["String(\""; x; "\")"]
  | Id(x) -> concat ["Id(\""; x; "\")"]
  | Assign(el) -> concat ["Assign("; stringify (List(el)); ")"]
  | Eval(str, el) -> concat ["Eval("; str; ", "; stringify (List(el)); ")"]
  | Nil -> "Nil"
  | List(expl) -> concat ["List("; (String.concat ", " (List.map (fun x -> stringify x) expl)); ")"]
  | Fdecl(args, e) -> concat ["Fdecl(["; String.concat ", " (List.map (fun x -> concat ["\""; x;
"\""]) args); "], "; stringify e; ")"]
  | If(cond, thenb, elseb) -> concat ["If("; stringify cond; ", "; stringify thenb; ", "; stringify
elseb; ")"]
  | For(init, cond, update, exp) -> concat ["For("; stringify init; ", "; stringify cond; ", ";
stringify update; ", "; stringify exp; ")"]
  | While(cond, exp) -> concat ["While("; stringify cond; ", "; stringify exp; ")"]
  | Let(str, exp1, exp2) -> concat ["Let(\""; str; "\", "; stringify exp1; ", "; stringify exp2; ")"]

let stringify_prog exp_list =
  String.concat "" ["["; (String.concat ", " (List.map stringify exp_list)); "]"]
```

test.ml

```
open OUnit

let empty_list = []
let singleton_list = [1]

let dummy_test _ =
  assert_equal 0 (List.length empty_list);
  assert_equal 1 (List.length singleton_list)

let suite = "OUnit Example" >::: ["dummy_test" >:: dummy_test]

let _ =
  run_test_tt_main suite
```

type_infer.ml

```
(* Type inference *)

open Ast;;
```

```
exception Type_error of string
exception Invalid_args of string
exception Unknown_variable of string * string

(** [type_error msg] reports a type error by raising [Type_error msg]. *)
let type_error msg = raise (Type_error msg)

(** [invalid_args_error msg] reports invalid argument exceptions by raising [Invalid_args
msg]. *)
let invalid_args_error msg = raise (Invalid_args msg)

(** [unknown_var_error msg] reports unknown variable exceptions by raising
[Unknown_variable msg]. *)
let unknown_var_error msg var = raise (Unknown_variable (msg, var))

let remove_underscores input =
  let uscore = Str.regexp_string "_" in
   Str.global_replace uscore "" input

(** [fresh ()] returns an unused type parameter. *)
let fresh =
  let k = ref 0 in
    fun () -> incr k; TParam !k

(** [refresh t] replaces all parameters appearing in [t] with unused ones. *)
let refresh ty =
  let rec refresh s = function
    | TInt -> TInt, s
    | TFloat -> TFloat, s
    | TBool -> TBool, s
    | TString -> TString, s
    | TUnit -> TUnit, s
    | TSome -> TSome, s
    | TJblob -> TJblob, s
    | TException -> TException, s
    | TParam k ->
        (try
           List.assoc k s, s
         with Not_found -> let t = fresh () in t, (k,t)::s)
    | TArrow t_list ->
        let rec tarrow_refresh ts us s' =
                match ts with
                | [] -> us, s'
                | hd::tl -> let u,s'' = refresh s' hd in
                                        tarrow_refresh tl (us@[u]) s'' in
```

```
          let u_list, s = tarrow_refresh t_list [] s in
          TArrow u_list, s
    | TSomeList t ->
          let u, s' = refresh s t in
            TSomeList u, s'
  in
    fst (refresh [] ty)

(** [occurs k t] returns [true] if parameter [TParam k] appears in type [t]. *)
let rec occurs k = function
  | TInt -> false
  | TFloat -> false
  | TString -> false
  | TBool -> false
  | TUnit -> false
  | TSome -> false
  | TJblob -> false
  | TException -> false
  | TParam j -> k = j
  | TArrow t_list ->
                  let rec tarrow_occurs ts o =
                            match ts with
                            | [] -> o
                            | hd::tl -> (tarrow_occurs tl (o||occurs k hd)) in
                  tarrow_occurs t_list false
  | TSomeList t1 -> occurs k t1

(* [get_name_of_functions fname] takes the intermediate function name inside of
          compiler and returns the actual function name users use. *)
let get_name_of_functions = function
          | "exec" -> "do"
          | "__add" -> "+"
          | "__sub" -> "-"
          | "__mult" -> "*"
          | "__div" -> "/"
          | "__addf" -> "+."
          | "__subf" -> "-."
          | "__multf" -> "*."
          | "__divf" -> "/."
          | "__equal" -> "is"
          | "__neq" -> "isnt"
          | "__less" -> "<"
          | "__leq" -> "<="
          | "__greater" -> ">"
          | "__geq" -> ">="
          | "__and" -> "and"
          | "__or" -> "or"
          | "__not" -> "not"
          | "__concat" -> "++"
          | "__evaluate" -> "eval"
          | _ as s -> s

(** [solve [(t1,u1); ...; (tn,un)] solves the system of equations
   [t1=u1], ..., [tn=un]. The solution is represented by a list of
   pairs [(k,t)], meaning that [TParam k] equals [t]. A type error is
   raised if there is no solution. The solution found is the most general
   one.
*)
let solve eq =
  let rec solve eq sbst =
    match eq with
     | [] -> sbst

     | (t1, t2 , f) :: eq when t1 = t2 -> solve eq sbst

     | ((TParam k, t, f) :: eq | (t, TParam k, f) :: eq) when (not (occurs k t)) ->
         let ts = Ast.tsubst [(k,t)] in
           solve
             (List.map (fun (ty1,ty2,fn) -> (ts ty1, ts ty2, fn)) eq)
             ((k,t)::(List.map (fun (n, u) -> (n, ts u)) sbst))

     | (TException, _, f) :: eq | (_, TException, f) :: eq -> solve eq sbst
     | (TArrow ty1, TArrow ty2, f) :: eq when (List.length ty1) = (List.length ty2) ->
        let rec get_eq l1 l2 eq =
                match l1 with
                | [] -> eq
                | hd::tl -> get_eq tl (List.tl l2) ((hd, (List.hd) l2, f)::eq) in
        solve (get_eq ty1 ty2 eq) sbst

     | (TSomeList t1, TSomeList t2, f) :: eq ->
        solve ((t1,t2,f) :: eq) sbst
     | (t1,t2,f)::_ ->
         let u1, u2 = rename2 t1 t2 in
           type_error ("Type Incompatible Error: The types " ^ string_of_type u1 ^ " and "
^
                                 string_of_type u2 ^ " are incompatible in " ^
(get_name_of_functions f) ^ "." )

in
   solve eq []
```

```
(** [constraints_of gctx e] infers the type of expression [e] and a set
    of constraints, where [gctx] is global context of values that [e]
    may refer to. *)
let rec constraints_of gctx =
  let rec cnstr ctx = function
    | Id x -> (
        (try
          List.assoc x ctx, []
         with Not_found ->
          (try
            (* we call [refresh] here to get let-polymorphism *)
            refresh (List.assoc x gctx), []
           with Not_found ->
                      unknown_var_error ("Unknown Variable Error: variable " ^x^ " is
unknown.") x
        )
    ))
    | Int _ ->  TInt, []
    | Float _ -> TFloat, []
    | String _ -> TString, []
    | Boolean _ -> TBool, []
    | Nil -> TSomeList (fresh ()), []
    | List thelist -> let rec addcnstr = function
        | [] -> []
        | hd::tl -> let ty1, eq1 = cnstr ctx hd in
              eq1 @ addcnstr tl
        in TSomeList(TSome), addcnstr thelist

    | If (e1, e2, e3) ->
        let ty1, eq1 = cnstr ctx e1 in
        let ty2, eq2 = cnstr ctx e2 in
        let ty3, eq3 = cnstr ctx e3 in
        let ty2 = match ty2 with
        | TException -> ty3
        | _ -> ty2

        in let ty3 =
        match ty3 with
        | TException -> ty2
        | _ -> ty3

        in
          ty2, ((ty1, TBool, "if")::(ty2, ty3, "if")::eq1@eq2@eq3)
```

```
   | Fdecl(x, e) ->
         let eqs = List.rev (List.map (fun v -> (v, fresh ())) x)in
         let ty1, eq = cnstr (eqs@ctx) e in
         (match eqs with
                 |[] -> TArrow [TUnit;ty1], eq
                 | _ -> let func_types = List.map (fun (a,b) -> b) eqs in
                                         TArrow (func_types@[ty1]), eq
         )

   | Assign(e) -> (* all ids/types were already added to ctx in the executor. just return the
type of the last assignment *)
         let last = List.hd (List.rev e) in
         let ty, eq = cnstr ctx last in
         ty, eq
   | Eval(e1, e2) -> (
       match e1 with
         | If(a,b,c)-> (
                 let ty1, eq1 = cnstr ctx e1 in
                         match ty1 with
                         | TArrow t_list -> (
                                 let ty2 = fresh() in
                                 match e2 with
                                 | [] -> ty2, (ty1, TArrow [TUnit; ty2], "function call")::eq1
                                 | _ -> let tys = List.map (fun v -> let (ty,eq) = cnstr ctx v in ty)
(List.rev e2) in
                                                 let rec get_eqs exp_list eq_list = (
                                                         match exp_list with
                                                         | [] -> eq_list
                                                         | hd::tl -> let (ty, eq) = cnstr ctx hd
in
                                                                         get_eqs tl
(eq_list@eq)
                                                 ) in
                                                 ty2, (ty1, TArrow (tys@[ty2]), "function
call")::eq1@(get_eqs e2 [])
                         )
                         | _ -> invalid_args_error ("Invalid Arguments Error: In function call
expression
                                         the first argument has type "^(string_of_type ty1)^",
but an expression was expected of type function")
         )
       | Fdecl(a,b) as e -> (
                 let ty1, eq1 = cnstr ctx e in
                 let ty2 = fresh () in
                         match e2 with
```

```
                        | [] -> ty2, (ty1, TArrow [TUnit; ty2], "function call")::eq1
                        | _ -> let tys = List.map (fun v -> let (ty,eq) = cnstr ctx v in ty) (List.rev
e2) in
                                        let rec get_eqs exp_list eq_list = (
                                                match exp_list with
                                                | [] -> eq_list
                                                | hd::tl -> let (ty, eq) = cnstr ctx hd in
                                                        get_eqs tl (eq_list@eq)
                                        ) in
                                        ty2, eq1@[(ty1, TArrow (tys@[ty2]), "function
call")]@(get_eqs e2 [])
                )
    | Eval(a, b) -> (
                let ty1, eq1 = cnstr ctx e1 in
                let ty2 = fresh() in
                match e2 with
                        | [] -> ty2, (ty1, TArrow [TUnit; ty2], "function call")::eq1
                        | _ -> let tys = List.map (fun v -> let (ty,eq) = cnstr ctx v in ty) (List.rev
e2) in
                                        let rec get_eqs exp_list eq_list = (
                                                match exp_list with
                                                | [] -> eq_list
                                                | hd::tl -> let (ty, eq) = cnstr ctx hd in
                                                        get_eqs tl (eq_list@eq)
                                        ) in
                                        ty2, eq1@[(ty1, TArrow (tys@[ty2]), "function
call")]@(get_eqs e2 [])
    )
    | Id(e1) ->
    (
        match e1 with

        | "__add" | "__sub" | "__mult"| "__div"
        | "__addf" | "__subf" | "__multf" | "__divf"
        | "__concat" -> (
                let tarrow = Generator.arrow_of(e1) in
                match tarrow with
                | TArrow(x) -> (let a = List.hd x in
                                let b = List.nth x 1 in
                                let c = List.nth x 2 in
                        match e2 with
                        | [] -> c, []
                        | hd::tl -> let ty1, eq1 = cnstr ctx hd in
                          let ty2, eq2 = cnstr ctx (Eval(Id(e1), tl)) in
                          c, (ty1,a,e1) :: (ty2,b,e1) :: eq1 @ eq2)
```

```
              | _ -> raise(Failure "Error: Generation of typing for built-in function failed. ")
            )
            | "call" -> TJblob, [] (* (let len = List.length e2 in
                            match len with
                              | 1 -> let ty1, eq1 = cnstr ctx (List.hd e2) in
                                        TJblob, (ty1, TJblob, e1) :: eq1
                              | 2 -> let ty1, eq1 = cnstr ctx (List.hd e2) in
                                    let ty2, eq2 = cnstr ctx (List.nth e2 1) in
                                   TJblob, (ty1, TJblob, e1) :: (ty2, TString, e1) :: eq1 @ eq2
                              | 3 -> let ty1, eq1 = cnstr ctx (List.hd e2) in
                                    let ty2, eq2 = cnstr ctx (List.nth e2 1) in
                                    let ty3, eq3 = cnstr ctx (List.nth e2 2) in
                                        TJblob, (ty1, TJblob, e1) :: (ty2, TString, e1) :: (ty3,
TSomeList(TSome), e1) :: eq1 @ eq2 @ eq3
                              | _ -> invalid_args_error ("Invalid arguments error: call takes at
most 3 arguments. ")
                 ) *)
            | "dot" -> TJblob, [] (* if (List.length e2 != 2) then (invalid_args_error("Invalid
arguments error: " ^
                                                "dot takes 1 JBlob and 1 list as
arguments. "))
                            else (
                             let jblob = List.hd e2 in
                               let ty1, eq1 = cnstr ctx jblob in
                               let ty2, eq2 = cnstr ctx (List.nth e2 1) in
                             TJblob, (ty1, TJblob, e1) :: (ty2, TSomeList(TSome), e1) :: eq1 @
eq2
                            ) *)
            | "module" ->
                            if (List.length e2 != 1) then (invalid_args_error ("Invalid arguments
error: " ^
                                                "module takes 1 String as argument. "))
                            else (
                                  let ty, eq = cnstr ctx (List.hd e2) in
                                  TJblob, (ty, TString, e1) :: eq
                            )
            | "exception" ->
                            let e = List.hd e2 in
                            let ty, eq = cnstr ctx e in
                            TException, (ty, TString, e1) :: eq

            | "mod" | "__and" | "__or" -> if (List.length e2 != 2) then
                   (invalid_args_error("Invalid arguments error: the ("
                       ^ (remove_underscores e1) ^ ") operation takes 2 arguments. "))
                   else (
                        let expected = match e1 with
                          | "mod" -> TInt
                          | _ -> TBool
                        in
                          let hd = List.hd e2 in
                          let tl = List.nth e2 1 in
                            let ty1, eq1 = cnstr ctx hd in
                            let ty2, eq2 = cnstr ctx tl in
                          expected, (ty1, expected, e1) :: (ty2, expected, e1) :: eq1 @ eq2
                )

        | "__not" -> if (List.length e2 != 1) then (invalid_args_error("Invalid Arguments
Error: " ^
                "not takes 1 boolean expression as argument. "))
                else (
                        let hd = List.hd e2 in
                        let ty, eq = cnstr ctx hd in
                        TBool, (ty, TBool, e1) :: eq
                )
        | "__equal" | "__neq" | "__less" | "__leq"
        | "__greater" | "__geq" ->
                if (List.length e2 != 2) then
                  (invalid_args_error("Invalid arguments error: the (" ^
                        (remove_underscores e1) ^ ") comparison takes 2 arguments. "))
                else (
                  let ty1, eq1 = cnstr ctx (List.hd e2) in
                  let ty2, eq2 = cnstr ctx (List.nth e2 1) in
                    TBool, (ty1, ty2, e1) :: eq1 @ eq2
                )

        | "cons" -> (
                if List.length e2 != 2 then (invalid_args_error("Invalid Arguments Error: " ^
"cons takes 2 arguments. "))
                else (
                        let newhd = List.hd e2
                        and thelist = List.hd (List.rev e2) in
                                let ty1, eq1 = cnstr ctx newhd in    (* ty1 can be anything *)
                                let ty2, eq2 = cnstr ctx thelist in
                                        TSomeList(TSome), (ty2,TSomeList(TSome), e1) :: eq1
@ eq2
                )
        )

        (* check that every arg to print is of type TString, and every arg to exec is
TSomeList(TSome) *)
```

```
        | "pr" | "prn" | "exec" -> (
            let tarrow = Generator.arrow_of e1 in
                match tarrow with
                | TArrow(x) -> let a = List.hd(x) in
                                let b = List.nth x 1 in
                        let rec addcnstr = function
                          | [] -> []
                          | hd::tl -> let ty1, eq1 = cnstr ctx hd in
                                (ty1,a,e1) :: eq1 @ addcnstr tl
                        in b, addcnstr e2
                | _ -> raise(Failure "Error: Generation of typing for built-in function
failed. ")
        )

        | "int_of_string"
        | "string_of_float"
        | "float_of_string"
        | "string_of_boolean"
        | "boolean_of_string"
        | "string_of_int" ->
        (
            if List.length e2 != 1 then (invalid_args_error("Invalid Arguments Error: " ^
                                                    e1 ^ " takes 1 argument. "))
            else (
                let arg = List.hd e2 in
                let ty, eq = cnstr ctx arg in
                let tarrow = (Generator.arrow_of e1) in
                    match tarrow with
                     | TArrow(x) -> (let a = List.hd(x) in
                                let b = List.hd(List.tl x) in
                                b, (ty, a, e1) :: eq)
                     | _ -> raise(Failure "Error: Generation of typing for built-in
function failed. ")
            )
        )

        | "tail"
        | "head"
        | "evaluate" ->
            if List.length e2 != 1 then (
                let fname =
                            if (String.compare e1 "evaluate") == 0 then "eval"
                            else e1
                        in
                        invalid_args_error("Invalid Arguments Error: " ^ fname ^ "
takes 1 list as argument. "))
                    else (
                              let thelist = (List.hd e2) in
                                let ty, eq = cnstr ctx thelist in
                                  let tarrow = (Generator.arrow_of e1) in
                                        match tarrow with
                                        | TArrow(x) -> (let a = List.hd(x) in
                                                            let b = List.hd(List.tl x) in
                                                            b, (ty, a, e1) :: eq)
                                        | _ -> raise(Failure "Error: Generation of typing for built-in
function failed. ")
                    )

          | "int" | "float" | "boolean" | "string" | "list"
          | "type" ->
          (
                    if List.length e2 != 1 then
                              (invalid_args_error("Invalid Arguments Error: " ^ e1 ^ " takes 1
argument." ))
                    else (
                              let x = List.hd e2 in
                              let ty1, eq1 = cnstr ctx x in
                              let tarrow = (Generator.arrow_of e1) in
                                        match tarrow with
                                                  | TArrow(x) -> List.hd(List.rev x), eq1
                                                  | _ -> raise(Failure "Error: Generation of typing for
built-in function failed. ")
                    )
          )
          | _ -> ( let ty1, eq1 = cnstr ctx (Id e1) in
                              let ty2 = fresh () in
                              match e2 with
                              | [] -> ty2, (ty1, TArrow [TUnit; ty2], e1)::eq1
                              | _ -> let tys = List.map (fun v -> let (ty,eq) = cnstr ctx v in ty) (List.rev
e2) in
                                                  let rec get_eqs exp_list eq_list = (
                                                            match exp_list with
                                                            | [] -> eq_list
                                                            | hd::tl -> let (ty, eq) = cnstr ctx hd in
                                                                      get_eqs tl (eq_list@eq)
                                                  ) in
                                                  ty2, eq1@[(ty1, TArrow (tys@[ty2]),e1)]@(get_eqs e2 [])
                   )
     ) (* end pattern matching for Id *)
```

```
          | _ -> invalid_args_error ("Invalid Arguments Error: In function call expression the
first argument is invalid.")

   ) (* end pattern matching for Eval *)
     in
       cnstr []

(** [type_of ctx e] computes the principal type of expression [e] in
   context [ctx]. *)
let type_of ctx e =
  let ty, eq = constraints_of ctx e in
   let ans = solve eq in
            let rec printType = function
         | TInt -> "TInt"
                     | TBool -> "TBool"
                     | TParam k -> "TypeVar" ^ (string_of_int k)
                     | TSome -> "SomeType"
                     | TSomeList _ -> "List of sometype"
                     | TString -> "TString"
                     | TUnit -> "TUnit"
                     | TFloat -> "TFloat"
                     | TArrow t_list -> (
                            let rec print_types_list types str =
                                  ( match types with
                                          |[] -> str
                                          | hd::tl when (List.length tl) = 0 -> str^(printType hd)
                                          | hd::tl when (List.length tl > 0) -> str^(printType hd)^"
-> "
          | _ -> raise(Failure "Exception: List.length is -1")
                                  ) in
                            print_types_list t_list ""
                     )
                     | _ -> "other case"
                   in let printpairs p =
                    ignore(printType (snd p));()
          in List.iter (printpairs) ans;
   tsubst (ans) ty
```

stdlib.ss

```
/*
**      *~*~ Superscript Standard Library ~*~*
**
**                 ______ ______
**                /  ___// ___/
**                \__  \\__  \
**            ___/  /___/  /
**           /_____//_____/
*/


/*
 *  identity takes an expression e and returns it.
 */
        (= identity (fn (e) e));;

/* * * * * * * * * * * * * * * * * * * * * * *
**
**   ~*~* List Manipulation Functions *~*~
**
** * * * * * * * * * * * * * * * * * * * * * * */

/*
 *  length takes a list l and returns its length.
 */
        (= length (fn (l)
          (if (is l '()) 0
            {1 + (length (tail l))})));;

/*
 *  nth takes an int n and a list l and returns the nth element of the list.
 */
            (= nth (fn (n l)
          (if (or (is l '()) {n < 0}) (exception "nth: index out of bounds.")
            (if (is n 0) (head l)
              (nth {n - 1} (tail l))))));;

/*
 *  first takes a list l and returns the first element.
 */
            (= first (fn (l) (nth 0 l)));;

/*
 *  second takes a list l and returns the second element.
 */
            (= second (fn (l) (nth 1 l)));;

/*
 *  third takes a list l and returns the third element.
 */
            (= third (fn (l) (nth 2 l)));;
```

```
/*
 *  fourth takes a list l and returns the fourth element.
 */
        (= fourth (fn (l) (nth 3 l)));;

/*
 *  fifth takes a list [l] and returns the fifth element.
 */
        (= fifth (fn (l) (nth 4 l)));;

/*
 *  sixth takes a list l and returns the sixth element.
 */
        (= sixth (fn (l) (nth 5 l)));;

/*
 *  seventh takes a list and returns the seventh element.
 */
        (= seventh (fn (l) (nth 6 l)));;

/*
 *  eighth takes a list and returns the eighth element.
 */
        (= eighth (fn (l) (nth 7 l)));;

/*
 *  ninth takes a list and returns the ninth element.
 */
        (= ninth (fn (l) (nth 8 l)));;

/*
 *  tenth takes a list and returns the tenth element.
 */
        (= tenth (fn (l) (nth 9 l)));;

/*
 *  last takes a list and returns the last element.
 */
        (= last (fn (l) (nth (- (length l) 1) l)));;

/*
 *  map takes a function f and a HOMOGENEOUS list l. It evaluates f
 *  on each element of the list, and returns a list of the results.
 */
```

```
        (= map (fn (f l)
          (if (is l '()) '()
            (cons (f (head l)) (map f (tail l))))));;

/*
 *  fold_left takes a function f, HOMOGENEOUS list a, and list l.
 *  It evaluates f on each element of list l, starting from the left,
 *  and stores the results in the accumulator list a.  It returns a.
 */
        (= fold_left (fn (f a l)
          (if (is l '()) a
            (fold_left f (eval '(f a (head l))) (tail l)))));;

/*
 *  fold_right takes a function f, HOMOGENEOUS list l, and list a.
 *  It evaluates f on each element of list l, starting from the right,
 *  and stores the results in the accumulator list a. It returns a.
 */
        (= fold_right (fn (f l a)
          (if (is l '()) (eval '(identity a))
            (eval '(f (head l) (fold_right f (tail l) a))))));;

/*
 *  filter takes a function p and HOMOGENOUS list l. The function must
 *  take 1 argument and return a BOOLEAN.  Filter returns a list of
 *  those elements of l, on which the predicate evaluates to true.
 */
        (= filter (fn (p l)
          (list (fold_right (fn (x y) (if (p x) (cons x y) y)) l '()))));;

/*
 *  append takes an element a, and list b. It returns a list
 *  where a is appended onto the front of b.
 */
        (= append (fn (a b)
          (if (is a '())
            b
            (cons (head a) (append (tail a) b)))));;

/*
 *  reverse takes a list l. It returns the list reversed.
 */
        (= reverse (fn (l)
          (if (is l '())
            l
            (append (reverse (tail l)) (cons (head l) '())))));;

/*
 *  drop takes an int i and list l. It drops the first i elements
 *  from the front of the list, and returns the resulting list.
 */
        (= drop (fn (i l)
          (if (< i 1)
            l
            (drop (- i 1) (tail l)))));;

/*
 *  take takes an int i and list l. It returns a list containing
 *  the first i elements of the list.
 */
        (= take (fn (i l)
          (if (or (not (> i 0)) (is l '()))
            '()
            (cons (head l) (take (- i 1) (tail l))))));;

/*
 *  intersperse takes an expression e and list l. It inserts the
 *  expression between each pair of elements in the list,
 *  and returns the resulting list.
 */
        (= intersperse (fn (e l)
          (if (or (is l '()) (is (tail l) '()))
            l
            (cons (head l) (cons e (intersperse e (tail l)))))));;

/*
 *  member takes an expression e and list l. It returns true if
 *  e is an element of l, or false if e is not an element of l.
 */
        (= member
          (fn (e l)
            (if (is l '())
              false
              (if (boolean (eval '(is (head l) e)))
            true
            (member e (tail l))
          )
        )
          )
        );;
```

```
/*
 *  zipwith takes function f, list a, and list b. The function should
 *  take 2 arguments. zipwith returns a list where each element is the
 *  result of evaluating (f a b).
 */
          (= zipwith (fn (f a b)
            (if (or (is a '()) (is b '()))
              '()
              (cons (f (head a) (head b)) (zipwith f (tail a) (tail b))))));;

/*
 *  zipwith3 takes a function f, list a, list b, and list c. The function
 *  should take 3 arguments. zipwith3 returns a list where each element
 *  is the result of evaluating (f a b c).
 */
          (= zipwith3 (fn (f a b c)
                    (if (or (or (is a '()) (is b '())) (is c '()))
                      '()
                      (cons (f (head a) (head b) (head c))
                        (zipwith3 f (tail a) (tail b) (tail c))))));;

/*
 *  zip takes list a and list b.  It iterates through both lists and
 *  returns  a new list of 'pairs' (2-element lists)
 *  where each pair has 1 element from a, and 1 from b.
 *  Example: (zip '(1 2 3) '(a b c)) ---> '( '(1 a) '(2 b) '(3 c) )
 */
          (= zip (fn (a b)
            (zipwith
              (fn (x y) (cons x (cons y '())))
              a b)));;

/*
 *  unzip takes list l, which should be a list of pairs (2-element lists).
 *  It returns a list of 2 lists, where the first sublist contains all the
 *  first elements, and the second sublist contains all of the
 *  second elements, from each pair of the input list l.
 *  Example: (unzip '( '(1 a) '(2 b) '(3 c) '(4 d) ))
 *                  --> '( '(1 2 3 4) '(a b c d) )
 */
          (= unzip (fn (l)
            (fold_right
              (fn (x y) '((append (first x) (cons (first y) '())) (append (second x) (cons(second y)
'())))))
```

```
             l
             '(   '() '()   ))));;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
**
**
**       *~*~  Printing / Stringify Functions ~*~*
**
**
**   USAGE NOTE: stringify_list and print_list take as an argument
**   a function that, when mapped onto a homogeneous list, stringifies
**   the elements of that list.
**
**   For ease of use, several formatting functions are provided below:
**
**                   format_[t] will stringify a list of atoms of type t.
**                   format_[t]2d will stringify a list of lists of type t.
**
**   These formatting functions may be passed as argument to
**   stringify_list or print_list.
**
**   Example: (print_list format_int '(1 2 3 4));;
**   >>>>>> prints [1,2,3,4]
**
** * * * * * * * * * * * * * * * * * * * * * * * * * * * /

/*
 *  format_boolean returns a function which stringifies a boolean value.
 */
          (= format_boolean (fn (x) (string_of_boolean (boolean x))));;

/*
 *  format_int returns a function which stringifies an int value.
 */
          (= format_int (fn (x) (string_of_int (int x))));;

/*
 *  format_string returns a function which takes a string and returns it.
 */
          (= format_string (fn (x) (string x)));;

/*
 *  format_float returns a function which stringifies a float.
 */
          (= format_float (fn (x) (string_of_float (float x))));;
```

```
/*
 *  stringify_list takes a function f and list l. The function f
 *  should return a function that can transform each element of the
 *  list into a string. The stringified elements are concatenated,
 *  with "," as delimiter, and returned altogether as a string.
 */
        (= stringify_list (fn (f l)
          (++
            "["
            (fold_left ++ "" (intersperse "," (map f l)))
            "]")));;

/*
 *  format_boolean2d returns a function that takes a HOMOGENEOUS list
 *  of BOOLEANS and stringifies it.
 */
        (= format_boolean2d (fn (x) (stringify_list format_boolean x)));;

/*
 *  format_int2d returns a function that takes a HOMOGENEOUS list
 *  of INTS and stringifies it.
 */
        (= format_int2d (fn (x) (stringify_list format_int x)));;

/*
 *  format_float2d returns a function that takes a HOMOGENEOUS list
 *  of FLOATS and stringifies it.
 */
        (= format_float2d (fn (x) (stringify_list format_float x)));;

/*
 *  format_string2d returns a function that takes a HOMOGENEOUS list
 *  of STRINGS and stringifies it.
 */
        (= format_string2d (fn (x) (stringify_list format_string x)));;

/*
 *  print_list takes a function f and list l. The function f should return
 *  a function that can transform each element of the list into a string.
 *  The stringified list is printed to console.
 */
        (= print_list (fn (f l)
          (prn (stringify_list f l))));;
```

```
###ENDSTDLIB###
```

examples/fib.c

```
= (fib
   fn (x)
     if {x is 0}
        0
        if {x is 1}
           1
           {fib({x-1}) + fib({x-2})}})
= (fib
   fn (x)
     if {x is 0}
        0
        if {x is 1}
           1
           +(fib({x-1}) fib({x-2})))
prn(string_of_int( fib({1 + 3})))
```