Pseudo: Final Report

Kristy Choi (eyc2120), Kevin Lin (kl2806),
Benjamin Low (bkl2115), Dennis Wei (dw2654), Raymond Xu (rx2125)

May 10, 2017

Contents

1 Introduction

Algorithmic thinking and analysis serve as the cornerstone of all application areas in computer science, equipping students and professionals alike with the ability to tackle challenging problems effectively and efficiently.

Our language will be perfect for rapidly prototyping algorithms, verifying the behavior of algorithms on given inputs, and for educational purposes, as the syntax is inspired by the easily-readable pseudocode from *Introduction to Algorithms* (also known as "CLRS"), the classic textbook for algorithmic analysis. Equipped with a concise syntax, powerful type inference system, rich built-in list functions, and on-demand objects of arbitrary structure, Pseudo will facilitate designing and implementing algorithms for all users.

2 Language Tutorial

2.1 Compiler Usage

To compile and run a program:

```
$ ./pseudo your_program_name.clrs
```

To just compile a program:

```
$ ./pseudo your_program_name.clrs your_program_name.ll
```

2.2 Basics

Pseudo has a extremely concise and beautiful syntax that closely resembles Python syntax. All Pseudo programs must have a `main()` function, the entry-point into any program.

Here is a simple program in Pseudo that prints "Hello world!".

```
def main():
    print "Hello world!"
```

We can compile and run this code by placing it into a file (let's call it "hello.clrs"), and running the following command:

```
$ ./pseudo hello.clrs
$ Hello world!
```

Here is a more complex program that utilizes user-defined functions and variables.

```
def main():
    my_list = [1, 2, 3, 4, 5]
    print sum(my_list)

def sum(list):
    s = 0
    for element in list:
        if element > 3:
            s = s + element
    print s
```

Running this example gives us:

```
$ ./pseudo sum.clrs
$ 9
```

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Comments

Comments are single-line and begin with //.

```
// this is a comment
```

3.1.2 Identifiers

Identifiers, or names, are used to describe the various components of Pseudo. They are composed of a sequence of alphanumeric characters and/or the character _, where the first character must be alphabetic. Identifiers are case sensitive - uppercase and lowercase characters are considered distinct.

```
// Valid identifiers
my_int = 10
flag7 = true

// Invalid identifiers
_int = 10
1thousand = 1000
```

3.1.3 Keywords

The list of identifiers reserved as keywords are below:

init	for	to
def	while	return
if	else	elseif
true	false	not
and	or	print

We will provide examples of usage for each keyword throughout the rest of the subsections.

3.1.4 Numerical Constants

Numerical constants consist of a sequence of digits from 0-9, including a hyphen (-) for negative numbers and a decimal point (.) for floating point numbers. Pseudo only supports decimal numbers - number systems in other bases (e.g. binary, hexadecimal) are not allowed.

```
// Valid numerical constants
42
35.76
-7
```

3.2 Types

3.2.1 Primitive Data Types

There are three primitives in pseudo: `num`, `bool`, and `string`.

3.2.1.1 `num`

`num` represents both integers and floating point numbers. `num` types are 32-bits and follow IEEE 754 standard. Because there is no distinguishing factor between integers and floating point numbers, it is acceptable to declare numerics in a variety of ways:

```
x = 3
y = 3.0
```

3.2.1.2 `bool`

`bool` represents a simple boolean value, either `true` or `false`. They can be declared as follows:

```
boolean_one = true
boolean_two = false
```

3.2.1.3 `string`

`string` is a primitive data type in Pseudo. They are denoted by enclosing the desired text in double quotes. The string datatype supports all ASCII characters. To insert the " character in a string, use \" to avoid ending the string.

```
str_one = "This is a \"string\"" // This is a "string"
str_two = "this Is %$# another %!@) \nstring"
```

3.2.2 Lists

`Lists` are used for storing sequences of same-type data. A `List` is represented by a sequence of comma-separated elements is enclosed in two square brackets `[]`. The `List` is a mutable data structure, which means that it supports functions to append, remove, or update its values.

Within a `List`, each element must be of the same type – for example, a `List` may not hold a collection of both `num` and `string` elements.

```
A = [1, 2, 3, 4, 5]
A.set(0, 7) = 4 // [7, 2, 3, 4, 5]

B = [5, "pseudo", true] // Not allowed
```

3.2.3 Type Inference

Pseudo contains a robust type inference system. Given an expression, it will be determined at compile time what type each variable is an instance of. For example, given the expression

```
num_example = 3
```

it will be inferred that `num_example` is of type `num`.

This principle extends to `Lists` as well. For example, in the expression

```
sample_list = [1, 2, 3, 4]
```

`sample_list` will be inferred to be of type `List<num>`.

If there is a contradiction with types, a compilation error will occur.

3.2.4 Objects

3.2.4.1 Introduction

`Objects` in Pseudo are containers of variables that permit mixed data types. `Objects` are extremely flexible and are defined on-demand based on their usage. Objects do not have in-built member functions. Functions on objects must be defined as a static function independent of the object itself.

The fields of an `Object` are accessible via the dot (`.`) operator. Consider an `Object` which has the fields `name` and `age`. One can create and set the fields of such an `Object` with the following code:

```
person.name = "John Doe"
person.age = 42
```

These values can be referenced later in the program as well.

```
person.name == "John Doe" // true
person.age + 5 // 47
```

Note that **there is no explicit declaration of what fields are in an** `Object` and **there is no explicit declaration that the variable** `person` **is an** `Object`. This is due to our robust type inference system that extends to the `Object` layer.

3.2.4.2 Object Type Inference

Firstly, the types of the fields of an `Object` are inferred similarly to how standalone variables are inferred.

Secondly, The set of fields that an `Object` contains is determined by adopting a coercive inference scheme that considers two `Objects` the same type if they are ever used in a manner that would require them to the be same type. Then, the set of fields for any `Object` can be obtained by taking the union of all the fields that are invoked by same-type variables.

The following example illustrates our coercive inference scheme:

```
def main():
    a.age = 42
    c = foo(a)

def foo(b):
    b.name = "Pseudo"
    return b
```

In the example, `a`, `b`, and `c` are all considered same-type variables: `a` is the same type as `b` because they are linked through the `foo` function call in `main`, and `b` and `c` are linked through the assignment of `c`. Thus, `a`, `b`, and `c` are all `Objects` of the same type with fields `age` and `name`.

The values of the fields of `Objects` that have not been assigned are set to default values according to the following table:

Type	Default Value
num	0
bool	false
string	""
List	[]

3.2.4.3 Object Initialization

Normally, `Objects` belong to the scope where their first field assignment appears. `Objects` can also be initialized without a field assignment using the `init` keyword.

```
init a
if true:
    a.age = 15
```

The above code initializes an object called `a` with the field `age` in the outermost scope.

```
init x, y, z
x.bar = "baz"
```

The above code initializes 3 objects of the same type called `x`, `y`, and `z`. They are of the same type because they are initialized in the same `init` statement. Therefore, `y` and `z` also have a `bar` field which is set to the default value of empty string.

3.3 Operators and Expressions

3.3.1 Operators

3.3.1.1 Assignment

The = operator is used to assign the value of an expression to an identifier.

```
A = [1, 2, 3]
```

Assignment is right associative, allowing for assignment chaining.

```
a = b = 10 // Set both a and b to 10
```

3.3.1.2 Arithmetic Operators

The arithmetic operators consist of +, - ,*, and /. The order of precedence from highest to lowest is the unary - followed by the binary * and / followed by the binary + and -.

3.3.1.3 Logical Operators

The logical operators consist of the keywords `and`, `or` and `not`. The negation operator `not` keyword inverts `true` to `false` and vice versa. The logical operators can only be applied to boolean operands. The `and` keyword joins two boolean expressions and evaluates to true when both are true. The `or` keyword joins two boolean expressions and evaluates to `true` when both are `true`.

```
a = true
b = false
not a // false
a and b // false
a or b // true
```

3.3.1.4 Relational Operators

Relational operators consist of `>`, `<`, `>=`, `<=`, `==` and `!=` which have the same precedence. The equality comparisons compare by value. `==` and `!=` can apply to `bool` types and `num` types, relational operators can only be applied to `num` types.

```
a = 1
b = 1.0
a == b // true
```

3.3.1.5 String Operators

String concatenation is denoted by the binary `^` operator.

```
a = "Hello"
b = " world!"
c = a ^ b // "Hello world!"
```

3.3.1.6 Print

The `print` keyword allows users to print out literals and variables. `nums` are printed with 2 decimal places.

```
a = 5
b = "Pseudo"
print a
print b
print true
```

produces the output

```
5.00
Pseudo
1
```

For `Lists`, `print` prints out the contents of the list.

```
l = [1, 2, 3]
print l
```

produces the output

```
[1.00, 2.00, 3.00]
```

`print` also works for `Objects`. When an `Object` is printed, the names and values of its fields are printed. `bools` are printed as 1 and 0.

```
obj.foo = 5
obj.bar = "Pseudo"
obj.baz = true
print obj
```

produces the output

```
obj.foo: 5.00
obj.bar: Pseudo
obj.baz: 1
```

The order the fields are printed is not specified.

3.3.1.7 List Operators

Lists support the following operations:

Length - returns the length of the list

```
a = [4, 5, 6]
a.length() // 3
```

Insertion - inserts an element at an index

```
a = [4, 5, 6]
a.insert(1, 8)
// a == [4, 8, 5, 6]
```

Removal - removes the element at an index

```
a = [4, 5, 6]
a.remove(0) // 4
// a == [5, 6]
```

Push/Enqueue - inserts an element at the end of the list and return it

```
a = [4, 5, 6]
a.push(7) // 7
// a == [4, 5, 6, 7]
a.enqueue(8) // 8
// a == [4, 5, 6, 7, 8]
```

Pop - removes the last element and returns it

```
a = [4, 5, 6]
a.pop() // 6
// a == [4, 5]
```

Dequeue - removes the first element and returns it

```
a = [4, 5, 6]
a.dequeue() // 4
// a == [5, 6]
```

For each `e` in `L` - iterate over a list

```
a = [4, 5, 6]
for elem in L:
    print a
// 4
// 5
// 6
```

3.4 Control Flow

All statements must inside of functions. The entry point into Pseudo programs is a mandatory `main` function that takes no parameters. From there, statements execute in sequence.

3.4.1 Conditionals

The `if` statement is used for conditional execution of a series of expressions. Each statement is separated by a colon (:) to signal the end of a clause. The `elseif` keyword catches conditions that are skipped by `if`, and the `else` keyword catches all other cases.

```
if boolean-expression:
    // code
```

```
if boolean-expression:
    // code
else:
    // code
```

```
if boolean-expression:
    // code
elseif boolean-expression:
    // code
else:
    // code
```

3.4.2 For

The `for` statement is used to iterate over the elements in a sequence, allowing the user to repeatedly execute statements nested inside the loop. The `to` keyword can be used to specify a range to iterate over a starting and ending `num` type. The starting `num` is inclusive and the ending `num` is exclusive.

```
for i = 0 to 5:
    print i
```

produces the following output

```
0.00
1.00
2.00
3.00
4.00
```

To iterate through each element in a `list`, `for in` can be used.

```
str_list = ["hello", "world"]
for str in str_list:
    print str
```

produces the following output

```
"hello"
"world"
```

3.4.3 While

The `while` statement is another way to continuously execute a statement so long as the value of the boolean expression evaluates to `true`. This expression is evaluated prior to execution of the nested statement.

```
while boolean-expression:
    // code
```

3.5 Scope

The lexical scope of variables follows from the structure of the program. Variables declared at the outermost level extends from their definition through the end of the file in which they appear. Function definitions, conditionals and loops create their own local scope. If a variable is a defined in a higher scope then assignment to that variable changes that variable.

```
a = 10
if true:
    a = 20
    b = 20
print a // 20.00
print b // error
```

3.6 Functions

3.6.1 Declarations

Functions are declared with the `def` keyword, the name of the function, and the parameters being passed into the function surrounded by parentheses, followed by a colon. The naming convention follows that of identifiers - it must begin with an alphabetic character and may consist of any combination of alphanumeric characters and _. If there are multiple parameters, they are separated by commas. Functions are not required to have `return` statements.

```
def foo():
print "This is a function with no parameters"

def add(a, b):
    return a + b
```

Return values and parameters of the function will determined based on type inference, whether from a previous initialization/usage of the variable, or from the operations performed on it in the function.

```
def increment(a):
    return a + 1 // return type is num
```

The scope of a function is determined by its indentation. All lines that are one tab greater than the function declaration's indentation are considered part of the function. Pseudo knows when a function's scope has ended once it finds a line with the same indentation as the function declaration or a line that has less tabs.

3.6.2 Usage

User-defined functions may be called simply by providing the function name and the required parameters.

```
a = 3
b = 2
print add(3, 2)
```

Parameters are passed into functions by reference. This means that any non-primitive type variable passed into a function (e.g Lists, Objects) can have its contents modified even outside the function's scope. For example:

```
x = [1, 2, 3, 4, 5]

def modify(x):
    x[0] = 0
    return x

modify(x) // [0, 2, 3, 4, 5]
```

Primitives cannot be passed by reference. A primitive variable declared outside a function will always retain its value unless reassigned within its scope. Any function that the variable is passed into will not modify the value of the original variable.

3.7 Additional Examples

Fibonacci

```
def main():
    for i = 0 to 10:
        print fib(i)

def fib(n):
    if n == 1 or n == 0:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Selection Sort

```
def main():
    list = [5, 3, 7, 1, 2, 6]

    for i = 0 to list.length() - 1:
        min = 99999
        min_index = i + 1
        for j = i to list.length():
            if list.get(j) < min:
                min = list.get(j)
                min_index = j
        temp = list.get(i)
        list.set(i, list.get(min_index))
        list.set(min_index, temp)

    print list
```

Quicksort

```
def main():
    arr = [5, 6, 2, 1, 4, 7]
    print arr
    quicksort(arr, 0, arr.length() - 1)
    print arr

def quicksort(a, start, end):
    if start >= end:
        return a
    pivot = start
    new_pivot = partition(a, start, end, pivot)
    quicksort(a, start, new_pivot - 1)
    quicksort(a, new_pivot + 1, end)

def partition(a, start, end, pivot):
    swap(a, pivot, end)
    ret = start

    for i = start to end:
        if a.get(i) < a.get(end):
            swap(a, i, ret)
            ret = ret + 1

    swap(a, ret, end)
    return ret

def swap(a, i, j):
```

```
temp = a.get(i)
a.set(i, a.get(j))
a.set(j, temp)
```

4 Project Plan

4.1 Planning Process

Our team met three times a week: (1) on Monday evenings with our TA Alexandra Medway to discuss our progress from the week before; (2) on Tuesday nights to work on tasks that we had talked about with our TA; and (3) on Saturday afternoons to check in about the past couple days and to continue coding. We often had a number of milestones that we set to accomplish each week because different members of the team were involved in various aspects of the project. As we approached the project deadline, subsets of the team often met more frequently to make sure that pending tasks would be completed on time.

4.2 Specification Process

To make sure that all the software and programming conventions used to build our language were consistent across group members, we decided on the following specifications:

4.2.1 Software Development Environment

- **Github-Hosted Git Repo** - All features were developed on separate branches of our repo and then merged into our protected master branch after at least 1 other team member reviewed the pull request.
- **Travis CI** - We set up Travis to automatically run our test suite on every pull request, which added an additional level of integrity to our master branch.
- **OCaml 4.04.0** - The bulk of the compiler source code was written in OCaml.
- **LLVM 3.7** - Our Codegen module produces LLVM programs which was executed using LLI-3.7.
- **Python 2.7** - We used Python to develop our preprocessor that parses `Pseudo` code into an intermediate syntax representation that can be tokenized by an ocamllex-generated scanner.

4.2.2 Programming Style Guide

We established the following guidelines while programming our language:

- Indentation: we tried not to stray too far from the conventional OCaml indentation schemes to keep our code from looking too messy.
- Comments: We commented specific lines of code that were especially confusing for other group members to read.

4.3 Development Process

Our team implemented components in a front-to-back manner. First, we started with the preprocessor, scanner, and parser concurrently to ensure that we could generate `Pseudo` programs that could be parsed correctly. We set up a test suite for each of these components with plenty of unit tests before advancing further. Next, we worked on type inference using the Hindley-Milner algorithm to generate our SAST and augmented our test suite with a SAST pretty printer to verify the inference algorithm. We then worked on our object inference algorithm in parallel with our Codegen module, iterating adding tests as features were implemented.

4.4 Testing Process

We wrote extensive unit tests for the preprocessor, scanner, parser, and inference module that covered a multitude of features and edge cases. Tests for the scanner, parser, and inference module were made possible by our pretty printer, which deterministically printed all intermediate representations of our program so that we could match test outputs with expected outputs. All tests were run with the simple `make test` command. We also integrated Travis into our Github repo to automate test runs. As our Codegen module formed, we added end-to-end tests that compiled Pseudo files `.clrs` files to `.ll` files and ran them to check whether our output matched our expectations.

4.5 Team Responsibilities

As we progressed further along with the project and refined our initial ideas of what we wanted our language to look like, subsets of the team branched out to become more heavily involved in different components of the Compiler. We often pair-programmed and helped each other to make sure that any changes made by a single team member made would have no trouble being integrated with the rest of the code base. Because there was such significant overlap, there was a fluid division of responsibilities, although different members would oftentimes be more "specialized" in an aspect of the language than another.

Team Member	Responsibility
Kristy Choi	Codegen
Kevin Lin	Testing, Preprocessing, Codegen
Ben Low	Codegen
Dennis Wei	Scanner, Parser, AST, Type Inference, Codegen
Raymond Xu	Object Inference Algorithm, SAST

4.6 Project Timeline

Our project timeline is as follows:

Date	Task
February 8th	Project proposal submission
February 22nd	LRM submission
March 20th	Scanner and parser complete
March 27th	Completed "Hello world"
April 11th	Semantics + type inference complete
May 5th	Codegen complete
May 9th	Language complete; Project presentation
May 10th	Final Report submission

4.7 Project Log

Our project log is provided below:

Date	Task
TODO	First commit, creation of project repo
February 8th	Project proposal submission
February 22nd	LRM submission
March 27th	Completed "hello world"
April 10th	Finished general front-end
April 19th	Finished primitive type inference
April 26th	Finished general back-end
May 1st	Finished object front-end including object type inference
May 3rd	Finished list front-end including inference
May 7th	Finished object back-end
May 8th	Finished list back-end
May 9th	Project presentation
May 10th	Final Report submission

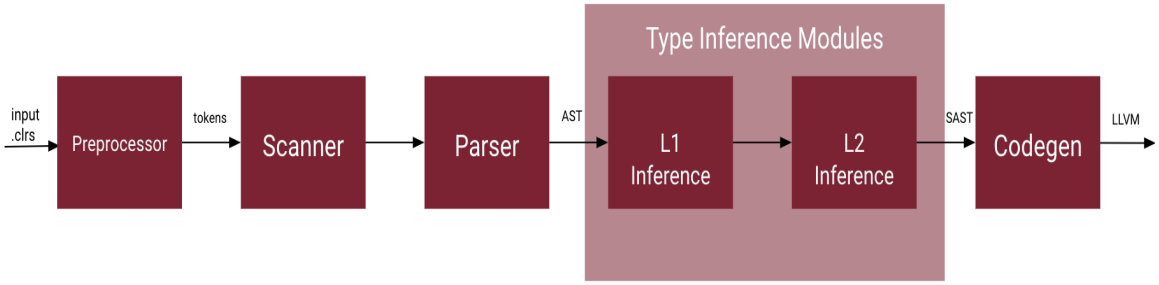

Figure 1: Architecture of Pseudo compiler

5 Architectural Design

5.1 Compiler Architecture

The architecture of the Pseudo compiler consists of the following key components: Preprocessor, Scanner, Parser, Typer Inference Module (L1 and L2), and Codegen, as shown in Figure 1. The Scanner and Parser comprise the compiler's front end, while the Type Inference module and Codegen make up the back end. With the exception of the Preprocessor, which was written in Python, all other components of the compiler were written in OCaml.

5.1.1 Preprocessor (Kevin)

The preprocessor reads a source program from standard input and writes a syntactically modified program to standard output. The main modifications are to strip comments and adds tokens to denote scoping based on indentation level. Because Pseudo's lightweight syntax poses challenges for tokenizing the input `.clrs` program, the preprocessor also adds in semicolons and braces to denote the end of statements and expressions.

5.1.2 Scanner (Dennis, Kevin)

The scanner accepts a pre-processed program as an argument and produces a sequence of tokens, as specified by Pseudo's syntax.

5.1.3 Parser (Dennis, Kevin)

The parser accepts sequence of tokens as an argument and parses it into an AST representation of the program.

5.1.4 Inference Module (Raymond, Dennis, Kevin)

The inference module accepts an AST as an argument and produces an SAST representation of the program as well as a Hashtable containing the object type definitions. The main distinction between the SAST and the AST is that the SAST contains type annotations for every expression, while the AST does not. The inference module is composed of two inference submodules, which we call L1 inference (primitives) and L2 inference (objects).

L1 inference The L1 layer of inference performs type inference on the AST expressions using the Hindley Milner algorithm to reduce expressions either to one of our three primitive types or to one of our aggregate types, including the underlying data types that constitute these aggregates. The Hindley Milner algorithm works by collecting a series of constraints related to an expression, recursively collecting the constraints in embedded expressions, then unifies these constraints to return one overall type for the expression.

For example, for the expression `a` = ("a" == "b"), constraints are generated that say `a` is of the same type as the right hand side of the assignment, the types of the two sides of the equality are the same, and the type of the equality is a boolean. Consequently, overall, the two strings are inferred as strings since they are literals, satisfying the type equality constraint. Then the equality is inferred as a boolean. Finally, the value `a` is transitively inferred as a boolean, and a variable name to type map records it as such.

The type inference module also handles are semantic checking, ensuring that expression types match the needed type in context. For example, as metioned above, the equality binary operation requires the two sides to be of the same type, an `if` statement requires its condition clause to be a boolean, and strong list typing is a asserted by checking to make sure all elements are of the same type at all times.

Within the overall AST structure, the type inference module is called on every statement and every expression using a depth first search on functions, beginning with the function `main`. Each function body is iterated through and inferred, and when a call is encountered, the called function is inferred before returning to the original function. Function types are also inferred from return statements within the function. However, to handle recursive functions, an additional pass is required to prevent recursive loops from continuously calling themselves. If a function call calls a function that has been "visited" already, then the inference module passes by the call and marks it as Undetermined before a second pass through the entirety of the program annotates these calls with the return type of the function.

L2 inference takes as input the intermediate SAST from L1 inference, in which all non-Object types are annotated. L2 inference uses a novel object inference algorithm to determine the types of all objects at compile-time. The algorithm breaks down into 5 main steps.

(1) Collect the fields for each object variable. This is done by examining all usages of the dot operator and maintaining a set of fields for each object variable.

(2) Collect equalities for object variables. Our L2 inference algorithm employs a *coercive object equality scheme*. This means that two objects are of the same type if they are used in a manner that would require them to be the same type. For example, if an object is placed into a list with other objects, they are all coerced to be the same type.

(3) Unify object variables to obtain object types. Using the object equalities and object field sets, we can now generate object definitions. Each object type is represented with an integer id. An object definition maps object ids to a set of fields. The set of fields is computed by unioning all fields of object variables that are equal.

(4) Annotate the SAST with object type ids. We traverse the program again to annotate every expression involving an object with the proper type id

(5) Pass to codegen the object definitions in addition to the SAST. Now that the SAST is annotated with object ids, codegen can utilize the object definitions to properly create and maintain each object.

5.1.5 Codegen (Ben, Kristy, Dennis, Kevin)

Codegen accepts an SAST and a Hashtable of object type definitions from the Inference Module. In order to support built-in functions for List operations and Objects, Codegen maintains a Hashtable for each of the following tasks: (1) mapping structs to the LLVM struct definitions; (2) mapping LLVM struct fields and types to their corresponding struct index; (3) mapping Object names to Object pointers; (4) mapping List names to their lengths. Additionally, it maintains a list of Hashtables to keep track of variable scoping. Using this information, Codegen produces the corresponding LLVM module as specified by the input `.clrs` program.

6 Test Plan

Our testing philosophy was to add tests concurrently while the compiler is being developed. The automated test suite was critical as our compiler had many parts with dependencies on each other– objects, inference and codegen. As these components all had to be worked on in parallel with one another and the test suite was crucial for ensuring that any changes to one part did not also break another part. We developed a JSON pretty print of our symantically checked abstract syntax tree. This allowed us to check our type inference algorithm.

6.1 Test suites

The tests are in the directory `tests` with subdirectories for the unit tests and integration tests respectively. We chose our tests to ensure the expected behavior the following functionality in priority.

- Primitive data types and operations: these are mainly checked in the semantic checking, which is part of the inference tests. We ensure that the operators for Strings, booleans etc. are operated on the correct types and output the correct output.
- Inference. Since our language relies heavily on our type inference algorithms, we chose many tests for the SAST to ensure that we cover inference cases for primitives, objects, lists.
- List operations. We test that our list support different kinds of outputs and the operations work as documented in the LRM.
- Recursion. We ensure that our programs work as expected for recursive functions as many algorithms rely on these.
- Control flow. We include tests for our while loops and different types of for loops as these are also essential for many algorithms.

6.1.1 Unit testing (preprocessing, scanner, parser, inference)

The preprocessing, scanner, parser depended on unit tests as they were worked on before the end to end program was set up. After that, they were run with integration tests to check that any modifications to the previous components for the end to end did not break their expected behvior. We also hand tested using menhir to debug our parser and AST.

Preprocessing. In our preprocessing step, the goals were to determine scoping, remove comments, and to add semicolons. Our unit test cases here focused on different scopes, nested scoping as well as adding comments in different places.

Scanner. The unit tests here concentrated on coverage of all our different tokens, we separated them to categories.

Parser. For the parser we implemented a pretty printer to print out the AST. This is done by the file parserize.ml.

Inference : For the inference algorithm, our tests focused on coverage of where we can get equality statements and infer the types of objects. We implemented printer.ml which takes the semantically checked AST and outputs the result of the inference so that we can check whether the types are what we expect them to be. The pretty printed file is in JSON output.

6.1.2 Integration testing

Integration testing involved running programs written in the Pseudo language and comparing it to the expected output. This runs through the entire pipline, from preprocessing to code generation. Our testing here focused on actual algorithms that the programmer may write such as quicksort or selection sort. In additon, we have fail tests that check that common programming errors do not result in programs accepted by the compiler.

6.2 Automation

The `make test` command in our Makefile in the main directly calls the shell scripts that runs each of the tests for preprocessing, scanning, parsing, inference and finally end to end. There is a shell script for each of these components that compares the `.in` files to `.out` files. The `.in` files contain the source code and the `.out` files contain the expected output.

6.3 Testing roles

While setting up the test suite (printer, parserize, shell scripts) was the tester (Kevin's) responsibility, the entire team contributed significantly to the tests. For each test, the member who developed the feature or component writes test cases. Members who review the pull request and the tester would add additional test cases when necessary.

We include the output of the test suite below:

```
demo-prep)  % make test

    ~/Documents/sp17/PLT/Pseudo
cd compiler; make
ocamlyacc parser.mly
ocamlc -c ast.mli
ocamlc -c parser.mli
ocamlc -c parser.ml
ocamllex scanner.mll
165 states, 9227 transitions, table size 37898 bytes
ocamlc -c scanner.ml
ocamlc -g -o pseudo parser.cmo scanner.cmo
cd tests; make test
cd scanner; make
ocamlc -I ../../compiler -o tokenize ../../compiler/scanner.cmo tokenize.cmo
cd parser; make
ocamlc -I ../../compiler -o parserize ../../compiler/parser.cmo ../../compiler/scanner.
    cmo parserize.cmo
```

```
./test_preprocessor.sh
Running preprocessor tests...
  - checking preprocessor/_call.in ...                    SUCCESS
  - checking preprocessor/_comments.in...                 SUCCESS
  - checking preprocessor/_eof.in ...                     SUCCESS
  - checking preprocessor/_hello.in ...                   SUCCESS
  - checking preprocessor/_hello_spaces.in ...            SUCCESS
  - checking preprocessor/_nesting.in ...                 SUCCESS
  - checking preprocessor/_nesting_spaces.in ...          SUCCESS
  - checking preprocessor/_rec.in ...                     SUCCESS
  - checking preprocessor/_selectionsort.in ...           SUCCESS
./test_scanner.sh
Running scanner tests...
  - checking scanner/_assign.in ...                       SUCCESS
  - checking scanner/_binops.in...                        SUCCESS
  - checking scanner/_brace.in...                         SUCCESS
  - checking scanner/_cond.in...                          SUCCESS
  - checking scanner/_control.in ...                      SUCCESS
  - checking scanner/_def.in ...                          SUCCESS
  - checking scanner/_eqneq.in...                         SUCCESS
  - checking scanner/_id.in ...                           SUCCESS
  - checking scanner/_key.in...                           SUCCESS
  - checking scanner/_literals.in ...                     SUCCESS
  - checking scanner/_logic.in ...                        SUCCESS
  - checking scanner/_ltgt.in ...                         SUCCESS
  - checking scanner/_obj.in...                           SUCCESS
  - checking scanner/_obj_init.in ...                     SUCCESS
  - checking scanner/_paren.in...                         SUCCESS
  - checking scanner/_utils.in ...                        SUCCESS
./test_parser.sh
Running parser tests...
  - checking parser/_combine.in...                        SUCCESS
  - checking parser/_combine_chain.in...                  SUCCESS
  - checking parser/_comments.in...                       SUCCESS
  - checking parser/_dict_decl.in ...                     SUCCESS
  - checking parser/_dict_of_lists.in ...                 SUCCESS
  - checking parser/_dict_ops.in ...                      SUCCESS
  - checking parser/_emb_undetermined_list.in...          SUCCESS
  - checking parser/_empty_list.in ...                    SUCCESS
  - checking parser/_eof.in ...                           SUCCESS
  - checking parser/_fdecl.in ...                         SUCCESS
  - checking parser/_for_range.in ...                     SUCCESS
  - checking parser/_hello.in ...                         SUCCESS
  - checking parser/_list_of_dicts.in ...                 SUCCESS
  - checking parser/_list_of_empty_lists.in ...           SUCCESS
  - checking parser/_list_of_list.in ...                  SUCCESS
  - checking parser/_list_ops.in ...                      SUCCESS
```

```
  - checking parser/_listdecl.in ...                    SUCCESS
  - checking parser/_no_return.in ...                   SUCCESS
  - checking parser/_noexpr_return.in ...               SUCCESS
  - checking parser/_obj.in ...                         SUCCESS
  - checking parser/_obj_init.in ...                    SUCCESS
  - checking parser/_objreassign.in ...                 SUCCESS
  - checking parser/_undetermined_dict.in ...           SUCCESS
  - checking parser/_undetermined_list.in ...           SUCCESS
cd ../compiler; make pseudo.native
make clean
ocamlbuild -clean
Finished, 0 targets (0 cached) in 00:00:00.
00:00:00 0      (0     ) STARTING

     --------- |rm -rf testall.log *.diff scanner.ml parser.ml parser.mli pseudo
rm -rf *.cmx *.cmi *.cmo *.cmx *.o
rm -rf *.err *.out *.ll
ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis,str,llvm.bitreader -cflags -w,+a
    -4 \
                pseudo.native
Finished, 26 targets (0 cached) in 00:00:02.
cd ../tests; ./test_inference.sh
Running inference tests ...
  - checking inference/_assign.in ...                   SUCCESS
  - checking inference/_call.in ...                     SUCCESS
  - checking inference/_combine.in ...                  SUCCESS
  - checking inference/_combine_chain.in ...            SUCCESS
  - checking inference/_emb_undetermined_list.in ...    SUCCESS
  - checking inference/ _list_of_empty_lists .in ...    SUCCESS
  - checking inference/ _list_of_list .in ...           SUCCESS
  - checking inference/_list_ops.in ...                 SUCCESS
  - checking inference/_listdecl.in ...                 SUCCESS
  - checking inference/_obj_assign.in ...               SUCCESS
  - checking inference/_obj_equality.in ...             SUCCESS
  - checking inference/ _obj_field_list_of_objs .in ... SUCCESS
  - checking inference/_obj_fields.in ...               SUCCESS
  - checking inference/_obj_func.in ...                 SUCCESS
  - checking inference/_obj_in_obj.in ...               SUCCESS
  - checking inference/_obj_inequality.in ...           SUCCESS
  - checking inference/_obj_init.in ...                 SUCCESS
  - checking inference/_obj_listdecl.in ...             SUCCESS
  - checking inference/_obj_lists.in ...                SUCCESS
  - checking inference/_obj_nesting.in ...              SUCCESS
  - checking inference/_obj_return.in ...               SUCCESS
  - checking inference/_obj_update.in ...               SUCCESS
  - checking inference/_undetermined_list.in ...        SUCCESS
  - checking inference/_while.in ...                    SUCCESS
```

./test_end_to_end.sh
Running end−to−end tests...
 − checking end_to_end/test−assignment.in... SUCCESS
 − checking end_to_end/test−binop.in... SUCCESS
 − checking end_to_end/test−default_assign.in ... SUCCESS
 − checking end_to_end/test−fibonacci.in ... SUCCESS
 − checking end_to_end/test−for−in−list.in... SUCCESS
 − checking end_to_end/test−for−range−by.in... SUCCESS
 − checking end_to_end/test−for.in... SUCCESS
 − checking end_to_end/test−func.in... SUCCESS
 − checking end_to_end/test−hello_world.in... SUCCESS
 − checking end_to_end/test−list−in−obj.in... SUCCESS
 − checking end_to_end/test−list−of−objects.in... SUCCESS
 − checking end_to_end/test−list−print.in ... SUCCESS
 − checking end_to_end/test−list−str.in ... SUCCESS
 − checking end_to_end/test−list.in ... SUCCESS
 − checking end_to_end/test−listops.in ... SUCCESS
 − checking end_to_end/test−obj−in−obj.in... SUCCESS
 − checking end_to_end/test−obj−return.in... SUCCESS
 − checking end_to_end/test−obj_init.in ... SUCCESS
 − checking end_to_end/test−obj_print.in... SUCCESS
 − checking end_to_end/test−objects.in... SUCCESS
 − checking end_to_end/test−print_string.in ... SUCCESS
 − checking end_to_end/test−quicksort.in... SUCCESS
 − checking end_to_end/test−scope.in... SUCCESS
 − checking end_to_end/test−selectionsort.in ... SUCCESS
 − checking end_to_end/test−unop.in... SUCCESS
 Running fail tests ...
 − checking end_to_end/fail/test_binop.in ... SUCCESS
 − checking end_to_end/fail/ test_call_notfunc .in ... SUCCESS
 − checking end_to_end/fail/ test_fnodecl .in ... SUCCESS
 − checking end_to_end/fail/test_nested_func.in ... SUCCESS
 − checking end_to_end/fail/test_nomain.in ... SUCCESS
 − checking end_to_end/fail/test_out_stmt.in ... SUCCESS
 − checking end_to_end/fail/test_scope.in ... SUCCESS
 − checking end_to_end/fail/test_type.in ... SUCCESS
 − checking end_to_end/fail/test_undefined_var.in ... SUCCESS
 − checking end_to_end/fail/test_while.in ... SUCCESS

6.4 Source language and target output

6.4.1 Quicksort

```
def main():
    arr = [5, 6, 2, 1, 4, 7]
    print arr
```

```
    quicksort(arr, 0, arr.length() - 1)
    print arr

def quicksort(a, start, end):
    if start >= end:
        return a
    pivot = start
    new_pivot = partition(a, start, end, pivot)
    quicksort(a, start, new_pivot - 1)
    quicksort(a, new_pivot + 1, end)

def partition(a, start, end, pivot):
    swap(a, pivot, end)
    ret = start

    for i = start to end:
        if a.get(i) < a.get(end):
            swap(a, i, ret)
            ret = ret + 1

    swap(a, ret, end)
    return ret

def swap(a, i, j):
    temp = a.get(i)
    a.set(i, a.get(j))
    a.set(j, temp)
```

```
; ModuleID = 'Pseudo'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [7 x i8] c"%.02f\0A\00"
@fmt.2 = private unnamed_addr constant [5 x i8] c"%s, \00"
@fmt.3 = private unnamed_addr constant [3 x i8] c"%s\00"
@fmt.4 = private unnamed_addr constant [5 x i8] c"%d, \00"
@fmt.5 = private unnamed_addr constant [3 x i8] c"%d\00"
@fmt.6 = private unnamed_addr constant [8 x i8] c"%.02f, \00"
@fmt.7 = private unnamed_addr constant [6 x i8] c"%.02f\00"
@string = private unnamed_addr constant [2 x i8] c"[\00"
@string.8 = private unnamed_addr constant [3 x i8] c"]\0A\00"
@string.9 = private unnamed_addr constant [2 x i8] c"[\00"
@string.10 = private unnamed_addr constant [3 x i8] c"]\0A\00"
@fmt.11 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.12 = private unnamed_addr constant [7 x i8] c"%.02f\0A\00"
@fmt.13 = private unnamed_addr constant [5 x i8] c"%s, \00"
```

```
@fmt.14 = private unnamed_addr constant [3 x i8] c"%s\00"
@fmt.15 = private unnamed_addr constant [5 x i8] c"%d, \00"
@fmt.16 = private unnamed_addr constant [3 x i8] c"%d\00"
@fmt.17 = private unnamed_addr constant [8 x i8] c"%.02f, \00"
@fmt.18 = private unnamed_addr constant [6 x i8] c"%.02f\00"
@fmt.19 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.20 = private unnamed_addr constant [7 x i8] c"%.02f\0A\00"
@fmt.21 = private unnamed_addr constant [5 x i8] c"%s, \00"
@fmt.22 = private unnamed_addr constant [3 x i8] c"%s\00"
@fmt.23 = private unnamed_addr constant [5 x i8] c"%d, \00"
@fmt.24 = private unnamed_addr constant [3 x i8] c"%d\00"
@fmt.25 = private unnamed_addr constant [8 x i8] c"%.02f, \00"
@fmt.26 = private unnamed_addr constant [6 x i8] c"%.02f\00"
@fmt.27 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.28 = private unnamed_addr constant [7 x i8] c"%.02f\0A\00"
@fmt.29 = private unnamed_addr constant [5 x i8] c"%s, \00"
@fmt.30 = private unnamed_addr constant [3 x i8] c"%s\00"
@fmt.31 = private unnamed_addr constant [5 x i8] c"%d, \00"
@fmt.32 = private unnamed_addr constant [3 x i8] c"%d\00"
@fmt.33 = private unnamed_addr constant [8 x i8] c"%.02f, \00"
@fmt.34 = private unnamed_addr constant [6 x i8] c"%.02f\00"

declare i32 @printf(i8*, ...)

declare i8* @puts(i8*, ...)

define void @main() {
entry:
  %local = alloca double, i32 6
  %tmp = getelementptr double, double* %local, i32 0
  store double 5.000000e+00, double* %tmp
  %tmp1 = getelementptr double, double* %local, i32 1
  store double 6.000000e+00, double* %tmp1
  %tmp2 = getelementptr double, double* %local, i32 2
  store double 2.000000e+00, double* %tmp2
  %tmp3 = getelementptr double, double* %local, i32 3
  store double 1.000000e+00, double* %tmp3
  %tmp4 = getelementptr double, double* %local, i32 4
  store double 4.000000e+00, double* %tmp4
  %tmp5 = getelementptr double, double* %local, i32 5
  store double 7.000000e+00, double* %tmp5
  %tmp6 = getelementptr double, double* %local, i32 0
  %tmp7 = load double, double* %tmp6
  %tmp8 = getelementptr double, double* %local, i32 1
  %tmp9 = load double, double* %tmp8
  %tmp10 = getelementptr double, double* %local, i32 2
  %tmp11 = load double, double* %tmp10
```

```
%tmp12 = getelementptr double, double* %local, i32 3
%tmp13 = load double, double* %tmp12
%tmp14 = getelementptr double, double* %local, i32 4
%tmp15 = load double, double* %tmp14
%tmp16 = getelementptr double, double* %local, i32 5
%tmp17 = load double, double* %tmp16
%puts = call i8* (i8*, ...) bitcast (i32 (i8*, ...) * @printf to i8* (i8*, ...) *)(i8*
    getelementptr inbounds ([2 x i8], [2 x i8]* @string, i32 0, i32 0))
%puts18 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp7)
%puts19 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp9)
%puts20 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp11)
%puts21 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp13)
%puts22 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp15)
%puts23 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8], [6 x i8]*
    @fmt.7, i32 0, i32 0), double %tmp17)
%puts24 = call i8* (i8*, ...) bitcast (i32 (i8*, ...) * @printf to i8* (i8*, ...) *)(i8
    * getelementptr inbounds ([3 x i8], [3 x i8]* @string.8, i32 0, i32 0))
%quicksort_result = call double* @quicksort(double* %local, double 0.000000e+00,
    double 5.000000e+00)
%tmp25 = getelementptr double, double* %local, i32 0
%tmp26 = load double, double* %tmp25
%tmp27 = getelementptr double, double* %local, i32 1
%tmp28 = load double, double* %tmp27
%tmp29 = getelementptr double, double* %local, i32 2
%tmp30 = load double, double* %tmp29
%tmp31 = getelementptr double, double* %local, i32 3
%tmp32 = load double, double* %tmp31
%tmp33 = getelementptr double, double* %local, i32 4
%tmp34 = load double, double* %tmp33
%tmp35 = getelementptr double, double* %local, i32 5
%tmp36 = load double, double* %tmp35
%puts37 = call i8* (i8*, ...) bitcast (i32 (i8*, ...) * @printf to i8* (i8*, ...) *)(i8
    * getelementptr inbounds ([2 x i8], [2 x i8]* @string.9, i32 0, i32 0))
%puts38 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp26)
%puts39 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp28)
%puts40 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp30)
%puts41 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
    @fmt.6, i32 0, i32 0), double %tmp32)
%puts42 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([8 x i8], [8 x i8]*
```

```
    @fmt.6, i32 0, i32 0), double %tmp34)
  %puts43 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([6 x i8], [6 x i8]*
    @fmt.7, i32 0, i32 0), double %tmp36)
  %puts44 = call i8* (i8*, ...) bitcast (i32 (i8*, ...)* @printf to i8* (i8*, ...)*)(i8
    * getelementptr inbounds ([3 x i8], [3 x i8]* @string.10, i32 0, i32 0))
  ret void
}

define double* @quicksort(double* %a, double %start, double %end) {
entry:
  %start1 = alloca double
  store double %start, double* %start1
  %end2 = alloca double
  store double %end, double* %end2
  %start3 = load double, double* %start1
  %end4 = load double, double* %end2
  %tmp = fcmp oge double %start3, %end4
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else
  %start5 = load double, double* %start1
  %pivot = alloca double
  store double %start5, double* %pivot
  %pivot6 = load double, double* %pivot
  %end7 = load double, double* %end2
  %start8 = load double, double* %start1
  %partition_result = call double @partition(double* %a, double %start8, double %end7
    , double %pivot6)
  %new_pivot = alloca double
  store double %partition_result, double* %new_pivot
  %new_pivot9 = load double, double* %new_pivot
  %tmp10 = fsub double %new_pivot9, 1.000000e+00
  %start11 = load double, double* %start1
  %quicksort_result = call double* @quicksort(double* %a, double %start11, double %
    tmp10)
  %end12 = load double, double* %end2
  %new_pivot13 = load double, double* %new_pivot
  %tmp14 = fadd double %new_pivot13, 1.000000e+00
  %quicksort_result15 = call double* @quicksort(double* %a, double %tmp14, double %
    end12)
  %local = alloca double
  %tmp16 = getelementptr double, double* %local, i32 0
  store double 0.000000e+00, double* %tmp16
  ret double* %local

then:                                             ; preds = %entry
  ret double* %a
```

```
else:                                             ; preds = %entry
  br label %merge
}

define double @partition(double* %a, double %start, double %end, double %pivot) {
entry:
  %start1 = alloca double
  store double %start, double* %start1
  %end2 = alloca double
  store double %end, double* %end2
  %pivot3 = alloca double
  store double %pivot, double* %pivot3
  %end4 = load double, double* %end2
  %pivot5 = load double, double* %pivot3
  call void @swap(double* %a, double %pivot5, double %end4)
  %start6 = load double, double* %start1
  %ret = alloca double
  store double %start6, double* %ret
  %start7 = load double, double* %start1
  %i = alloca double
  store double %start7, double* %i
  %end8 = load double, double* %end2
  %lt = fcmp olt double %start7, %end8
  %incr = alloca double
  store double 1.000000e+00, double* %incr
  br i1 %lt, label %incr.incr, label %incr.decr

incr.incr:                                        ; preds = %entry
  %incr9 = load double, double* %incr
  %tmp = fsub double %start7, %incr9
  store double %tmp, double* %i
  br label %for.cond

incr.decr:                                        ; preds = %entry
  %incr10 = load double, double* %incr
  %tmp11 = fadd double %start7, %incr10
  store double %tmp11, double* %i
  br label %for.cond

for.cond:                                         ; preds = %merge, %incr.decr, %incr.
    incr
  %iter = load double, double* %i
  %incr12 = load double, double* %incr
  %tmp13 = fadd double %iter, %incr12
  store double %tmp13, double* %i
  %tmp1 = fcmp olt double %tmp13, %end8
```

```
  br i1 %tmp1, label %for.init, label %for.done

for.init:                                         ; preds = %for.cond
  %i14 = load double, double* %i
  %tmp15 = fptoui double %i14 to i32
  %tmp16 = getelementptr double, double* %a, i32 %tmp15
  %tmp17 = load double, double* %tmp16
  %end18 = load double, double* %end2
  %tmp19 = fptoui double %end18 to i32
  %tmp20 = getelementptr double, double* %a, i32 %tmp19
  %tmp21 = load double, double* %tmp20
  %tmp22 = fcmp olt double %tmp17, %tmp21
  br i1 %tmp22, label %then, label %else

for.done:                                         ; preds = %for.cond
  %end27 = load double, double* %end2
  %ret28 = load double, double* %ret
  call void @swap(double* %a, double %ret28, double %end27)
  %ret29 = load double, double* %ret
  ret double %ret29

merge:                                            ; preds = %else, %then
  br label %for.cond

then:                                             ; preds = %for.init
  %ret23 = load double, double* %ret
  %i24 = load double, double* %i
  call void @swap(double* %a, double %i24, double %ret23)
  %ret25 = load double, double* %ret
  %tmp26 = fadd double %ret25, 1.000000e+00
  store double %tmp26, double* %ret
  br label %merge

else:                                             ; preds = %for.init
  br label %merge
}

define void @swap(double* %a, double %i, double %j) {
entry:
  %i1 = alloca double
  store double %i, double* %i1
  %j2 = alloca double
  store double %j, double* %j2
  %i3 = load double, double* %i1
  %tmp = fptoui double %i3 to i32
  %tmp4 = getelementptr double, double* %a, i32 %tmp
  %tmp5 = load double, double* %tmp4
```

```
  %temp = alloca double
  store double %tmp5, double* %temp
  %i6 = load double, double* %i1
  %j7 = load double, double* %j2
  %tmp8 = fptoui double %j7 to i32
  %tmp9 = getelementptr double, double* %a, i32 %tmp8
  %tmp10 = load double, double* %tmp9
  %tmp11 = fptoui double %i6 to i32
  %tmp12 = getelementptr double, double* %a, i32 %tmp11
  store double %tmp10, double* %tmp12
  %tmp13 = load double, double* %tmp12
  %j14 = load double, double* %j2
  %temp15 = load double, double* %temp
  %tmp16 = fptoui double %j14 to i32
  %tmp17 = getelementptr double, double* %a, i32 %tmp16
  store double %temp15, double* %tmp17
  %tmp18 = load double, double* %tmp17
  ret void
}
```

6.4.2 Fibonacci

```
def main():
    for i = 0 to 10:
        print fib(i)

def fib(n):
    if n == 1 or n == 0:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
; ModuleID = 'Pseudo'

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [7 x i8] c"%.02f\0A\00"
@fmt.2 = private unnamed_addr constant [5 x i8] c"%s, \00"
@fmt.3 = private unnamed_addr constant [3 x i8] c"%s\00"
@fmt.4 = private unnamed_addr constant [5 x i8] c"%d, \00"
@fmt.5 = private unnamed_addr constant [3 x i8] c"%d\00"
@fmt.6 = private unnamed_addr constant [8 x i8] c"%.02f, \00"
@fmt.7 = private unnamed_addr constant [6 x i8] c"%.02f\00"
@fmt.8 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.9 = private unnamed_addr constant [7 x i8] c"%.02f\0A\00"
@fmt.10 = private unnamed_addr constant [5 x i8] c"%s, \00"
@fmt.11 = private unnamed_addr constant [3 x i8] c"%s\00"
```

```
@fmt.12 = private unnamed_addr constant [5 x i8] c"%d, \00"
@fmt.13 = private unnamed_addr constant [3 x i8] c"%d\00"
@fmt.14 = private unnamed_addr constant [8 x i8] c"%.02f, \00"
@fmt.15 = private unnamed_addr constant [6 x i8] c"%.02f\00"

declare i32 @printf(i8*, ...)

declare i8* @puts(i8*, ...)

define void @main() {
entry:
  %i = alloca double
  store double 0.000000e+00, double* %i
  %incr = alloca double
  store double 1.000000e+00, double* %incr
  br i1 true, label %incr.incr, label %incr.decr

incr.incr:                                        ; preds = %entry
  %incr1 = load double, double* %incr
  %tmp = fsub double 0.000000e+00, %incr1
  store double %tmp, double* %i
  br label %for.cond

incr.decr:                                        ; preds = %entry
  %incr2 = load double, double* %incr
  %tmp3 = fadd double 0.000000e+00, %incr2
  store double %tmp3, double* %i
  br label %for.cond

for.cond:                                         ; preds = %for.init, %incr.decr, %
    incr.incr
  %iter = load double, double* %i
  %incr4 = load double, double* %incr
  %tmp5 = fadd double %iter, %incr4
  store double %tmp5, double* %i
  %tmp1 = fcmp olt double %tmp5, 1.000000e+01
  br i1 %tmp1, label %for.init, label %for.done

for.init:                                         ; preds = %for.cond
  %i6 = load double, double* %i
  %fib_result = call double @fib(double %i6)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([7 x i8], [7 x i8]*
      @fmt.1, i32 0, i32 0), double %fib_result)
  br label %for.cond

for.done:                                         ; preds = %for.cond
  ret void
```

```
}

define double @fib(double %n) {
entry:
  %n1 = alloca double
  store double %n, double* %n1
  %n2 = load double, double* %n1
  %tmp = fcmp oeq double %n2, 1.000000e+00
  %n3 = load double, double* %n1
  %tmp4 = fcmp oeq double %n3, 0.000000e+00
  %tmp5 = or i1 %tmp, %tmp4
  br i1 %tmp5, label %then, label %else

merge:                                            ; No predecessors!
  ret double 0.000000e+00

then:                                             ; preds = %entry
  ret double 1.000000e+00

else:                                             ; preds = %entry
  %n6 = load double, double* %n1
  %tmp7 = fsub double %n6, 1.000000e+00
  %fib_result = call double @fib(double %tmp7)
  %n8 = load double, double* %n1
  %tmp9 = fsub double %n8, 2.000000e+00
  %fib_result10 = call double @fib(double %tmp9)
  %tmp11 = fadd double %fib_result, %fib_result10
  ret double %tmp11
}
```

7 Lessons Learned

7.1 Kristy

There will be times when your part of the project is put on hold while another branch of the compiler needs to be fleshed out or fixed. Take this time to sit with your teammates and understand what's going on in their portion of the code, especially towards the beginning stages of the project. This will really help when it's closer to the project deadline and the group has to interface all the moving parts together. Design decisions made earlier in the project timeline may affect several aspects of the language downstream, and it's critical to be aware of them. All in all, I learned a lot from this class and doing this project has helped me to develop an appreciation for how programming languages are implemented under the hood.

7.2 Kevin

Coming in the PLT with little to no knowledge of how the tools I used so much daily worked internally really motivated me to learn about how they are actually implemented. Going to the lectures and reading the dragon book definitely was fun and helpful in this regard. I also was really looking forward to learning functional programming– I did get my feet wet in this regard as previously I haven't really programmed in a functional paradigm except a few exercises in Haskell. I think that prioritizing getting the compiler working often got in the way of actually writing code that exposed the benefits of the functional paradigm. I hope to learn more about it in the future. I learned a lot from my teammates. Kristy, our manager was phenomenal showing us how to keep the project deadlines on track and leading the team. Raymond, as the language guru taught me a lot about paying attention to details and how to develop a large software project. Dennis was years beyond the rest of us in knowledge of OCaml and LLVM, he taught us a ton about that. Ben, really helped me with understanding debugging and how to approach new code I haven't touched before. I would advise future groups to really think through their language before implementing it. Although languages evolve, a lot of time can be saved by first figuring out why we make the decisions we do. Understanding MicroC is also very helpful, even if your language is not similar to it. It will at least give a baseline for understanding every component from scanning to LLVM.

7.3 Ben

For me, the biggest takeaway from PLT this semester was learning how to tackle a large project using unfamiliar concepts and technologies. Getting over the language barrier in learning OCaml was probably the easiest part of the project - though I was new to functional programming and OCaml's learning curve is relatively steep, I felt I learned the language quick enough to be efficient at working on the project for most of the semester. The development process of Pseudo was the hardest challenge of the project, as we too often prioritized getting features working instead of thinking through the implementation details and how this will affect other features down the road. If I were to do this project again, I would spend more time thinking about how objects should be implemented in codegen, for example, instead of worrying about meeting deadlines and trying to get something quick working. A piece of advice: don't underestimate the difficulty of writing a language, but also don't underestimate your own ability to do so. Adding simple features like function parameters and recursion aren't as easy as it may seem at first, but at the same time, don't be afraid to go for cool but difficult features - it'll be challenging but extremely rewarding in the end!

7.4 Dennis

First, I'm going to say something relatively generic in that we should have started earlier. However, this is not in any way due to a lack of a clear and strict timeline.

Rather, figure out the work you can parallelize. There were moments throughout the project when my other team members were blocked because I had to finish a portion of the project. Most notably, it was incredibly difficult for our codegen team to start generating code without a finished type inference module, as they needed the types of our expressions. Instead of waiting and doing everything sequentially, we should have adopted a data endpoint model where we specified what the output of a module would be so the proceeding team could work with that hypothetical output as input while other portions were being completed. Especially if you plan on building a language with many features, figuring out how to optimize each and every moment you meet together as a team so that people are always working on something is incredibly helpful. This can take shape in many forms, such as learning how different parts of the code work, or contributing to the final report. GitHub issues are a great way to document this, and I regret not extensively using them for minute tasks until the end. In general, planning ahead and figuring out out what work was feasible at each moment would have saved us from many long nights towards the end of the semester.

On the design side, definitely think about how you plan on implementing your features before you decide to pursue them. You can do this by trying to write C code – without unions – that accomplishes your feature, and if you cannot do this incredibly simply, then don't bother. We had to scrap a lot of features due to implementation issues, despite having already spent a lot of time making these features work in moduels preceeding code generation, such as the AST and SAST.

Lastly, on the coding side, learn the tools you have at your disposal early in the process. Learn that you don't have to use OCaml's immutable maps and pass them everywhere, and that you can instead use the Hashtbl type instead. Take the time to learn how OCaml references work, they're incredibly useful. When you get to codegen, dump_value, dump_type, and dump_module frequently. LLVM code is surprisingly readable, and these tools can help you debug a lot faster.

Overall, this was an incredibly fulfilling experience, and I can definitely see myself building another compiler in the future (though maybe not in OCaml). Additionally, being able to be a part of this group was fantastic. I've enjoyed having the opportunity to learn more about how to manage timelines, automate and streamline the development environment, and build atop hastily written code, including my own, and I'm sincerely thankful to everyone for that.

7.5 Raymond

LLVM is tricky, and it might be too challenging or take too long for you to implement some features that you thought were reasonable. Instead of implementing every feature in your front-end and then writing the back-end later, take a more conservative vertical approach for your ambitious features. For example, we believed that implementing variable sized lists that support arbitrary types as well as a strongly-typed dictionary collection would be reasonable, but LLVM so much more than we expected. As a result, we never finished implementing the back-end for the

dictionary, even though we had spent a considerable amount of time implementing the front-end for it.

Like any software project, adopt and maintain a system for software development. That is, establish guidelines, standards, and rules for how to push changes, the standard of code quality acceptable, what bug tracker to use, what communication medium to use, etc. And even though your teammates are probably your friends, enforce these guidelines to the very end to save your codebase. It doesn't really matter what your systems are, as long as they are sensible. What is really important is that everyone in the team agrees on them and abides by them.

Adopt the Unix philosophy of making your development infrastructure, compiler components, and testing architecture modular. In a multi-component project such as a compiler, the ease of programatically scripting different pipelines is directly correlated to how convenient and efficient your development and testing process is. Build up your modular components, and then compose them in every useful way possible. Then make these compositions accessible with simple commands. We didn't think ahead when developing the individual components of our front-end, and as a result we had to perform numerous refactorings of the pipeline and compiler script to accommodate everything we wanted.

8 Appendix

Pseudo/.travis-ocaml.sh

```
## basic OCaml and opam installation

 full_apt_version  () {
   package=$1
   version=$2
   case "${version}" in
        latest ) echo -n "${package}" ;;
        *) echo -n "${package}="
           apt-cache show "$package" \
                | sed -n "s/^Version: \(${version}\)/\1/p" \
                | head -1
   esac
}

set -uex

# the ocaml version to test
OCAML_VERSION=${OCAML_VERSION:-latest}
OPAM_VERSION=${OPAM_VERSION:-1.2.2}
OPAM_INIT=${OPAM_INIT:-true}
OPAM_SWITCH=${OPAM_SWITCH:-system}
```

```
# the base opam repository to use for bootstrapping and catch-all namespace
BASE_REMOTE=${BASE_REMOTE:-git://github.com/ocaml/opam-repository}

# whether we need a new gcc and binutils
UPDATE_GCC_BINUTILS=${UPDATE_GCC_BINUTILS:-"0"}

# Install Trusty remotes
UBUNTU_TRUSTY=${UBUNTU_TRUSTY:-"0"}

# Install XQuartz on OSX
INSTALL_XQUARTZ=${INSTALL_XQUARTZ:-"true"}

case "$OCAML_VERSION" in
    latest ) OCAML_VERSION=4.02;;
esac

install_on_linux () {
  case "$OCAML_VERSION,$OPAM_VERSION" in
    3.12,1.2.2)
      OCAML_VERSION=4.02; OPAM_SWITCH="3.12.1"
      ppa=avsm/ocaml42+opam12 ;;
    4.00,1.2.2)
      OCAML_VERSION=4.02; OPAM_SWITCH="4.00.1"
      ppa=avsm/ocaml42+opam12 ;;
    4.01,1.2.2)
      OCAML_VERSION=4.02; OPAM_SWITCH="4.01.0"
      ppa=avsm/ocaml42+opam12 ;;
    4.02,1.1.2) OPAM_SWITCH=4.02.3; ppa=avsm/ocaml42+opam11 ;;
    4.02,1.2.0) OPAM_SWITCH=4.02.3; ppa=avsm/ocaml42+opam120 ;;
    4.02,1.2.1) OPAM_SWITCH=4.02.3; ppa=avsm/ocaml42+opam121 ;;
    4.02,1.2.2) ppa=avsm/ocaml42+opam12 ;;
    4.03,1.2.2)
      OCAML_VERSION=4.02; OPAM_SWITCH="4.03.0";
      ppa=avsm/ocaml42+opam12 ;;
    4.04,1.2.2)
       OCAML_VERSION=4.02; OPAM_SWITCH="4.04.0"
       ppa=avsm/ocaml42+opam12 ;;
   *) echo "Unknown OCAML_VERSION=$OCAML_VERSION OPAM_VERSION=
       $OPAM_VERSION"
      exit 1 ;;
  esac

  sudo add-apt-repository --yes ppa:${ppa}
  sudo apt-get update -qq
  sudo apt-get install -y \
    "$( full_apt_version ocaml $OCAML_VERSION)" \
    "$( full_apt_version ocaml-base $OCAML_VERSION)" \
```

```
    "$( full_apt_version  ocaml-native-compilers $OCAML_VERSION)" \
    "$( full_apt_version  ocaml-compiler-libs $OCAML_VERSION)" \
    "$( full_apt_version  ocaml-interp $OCAML_VERSION)" \
    "$( full_apt_version  ocaml-base-nox $OCAML_VERSION)" \
    "$( full_apt_version  ocaml-nox $OCAML_VERSION)" \
    "$( full_apt_version  camlp4 $OCAML_VERSION)" \
    "$( full_apt_version  camlp4-extra $OCAML_VERSION)" \
    jq \
    opam

  TRUSTY="deb mirror://mirrors.ubuntu.com/mirrors.txt trusty main restricted
      universe"

  if [ "$UPDATE_GCC_BINUTILS" != "0" ] ; then
    echo " installing a recent gcc and binutils (mainly to get mirage-entropy-xen
        working!)"
    sudo add-apt-repository "${TRUSTY}"
    sudo add-apt-repository --yes ppa:ubuntu-toolchain-r/test
    sudo apt-get -qq update
    sudo apt-get install -y gcc-4.8
    sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 90
    sudo add-apt-repository -r "${TRUSTY}"
  fi

  if [ "$UBUNTU_TRUSTY" != "0" ] ; then
    echo "Adding Ubuntu Trusty mirrors"
    sudo add-apt-repository "${TRUSTY}"
    sudo apt-get -qq update
  fi

}

install_on_osx () {
  case $INSTALL_XQUARTZ in
      true)
        curl -OL "http://xquartz.macosforge.org/downloads/SL/XQuartz-2.7.6.dmg"
        sudo hdiutil attach XQuartz-2.7.6.dmg
        sudo installer -verbose -pkg /Volumes/XQuartz-2.7.6/XQuartz.pkg -target /
        ;;
  esac
  brew update &> /dev/null
  case "$OCAML_VERSION,$OPAM_VERSION" in
    3.12,1.2.2) OPAM_SWITCH=3.12.1; brew install opam ;;
    4.00,1.2.2) OPAM_SWITCH=4.00.1; brew install opam ;;
    4.01,1.2.2) OPAM_SWITCH=4.01.0; brew install opam ;;
    4.02,1.2.2) OPAM_SWITCH=4.02.3; brew install opam ;;
    4.02,1.3.0) OPAM_SWITCH=4.02.3; brew install opam --HEAD ;;
```

```
    4.03,1.2.2) OPAM_SWITCH=4.03.0; brew install opam ;;
    4.04,1.2.2) OPAM_SWITCH=system; brew install ocaml; brew install opam ;;
    *) echo "Unknown OCAML_VERSION=$OCAML_VERSION OPAM_VERSION=
        $OPAM_VERSION"
       exit 1 ;;
  esac
  brew install jq
}

case $TRAVIS_OS_NAME in
    osx) install_on_osx ;;
    linux) install_on_linux ;;
esac

export OPAMYES=1

case $OPAM_INIT in
  true)
      opam init -a "$BASE_REMOTE" --comp="$OPAM_SWITCH"
      eval $(opam config env)
      ;;
esac

echo OCAML_VERSION=$OCAML_VERSION > .travis-ocaml.env
echo OPAM_SWITCH=$OPAM_SWITCH >> .travis-ocaml.env

ocaml -version
opam --version
opam --git-version
```

Pseudo/Makefile

```
# Makefile
# - main entrypoint for building compiler and running tests

default: all

all: clean build

build:
        cd compiler; make

test: build
        cd tests; make test

pseudo:
        cd compiler; make pseudo.native;

.PHONY: clean
```

```
clean:
	cd compiler; make clean
	cd tests; make clean
```

Pseudo/compiler/ast.mli

```
(*
 * COMS4115: Pseudo Abstract Syntax Tree
 *
 * Authors:
 *  - Raymond Xu
 *  - Kevin Lin
 *  - Kristy Choi
 *  - Dennis Wei
 *  - Benjamin Low
 *)

(* Unary operators *)
type unop =
  | Neg       (* - *)
  | Not       (* !/not *)

(* Binary operators *)
type binop =
  (* Arithmetic *)
  | Add       (* + *)
  | Minus     (* - *)
  | Times     (* * *)
  | Divide    (* / *)
  (* Boolean *)
  | Or        (* || *)
  | And       (* && *)
  | Eq        (* == *)
  | Neq       (* != *)
  | Less      (* < *)
  | Leq       (* <= *)
  | Greater   (* > *)
  | Geq       (* >= *)
  (* String *)
  | Concat    (* ^ *)
  (* List *)
  | Combine (* :: *)

(* Expressions *)
(*type num =
  | Num_int of int        (* 42 *)
  | Num_float of float    (* 42.0 *)*)

type expr =
```

```
  | Num_lit of float                  (* 42 *)
  | String_lit of string             (* "Hello, world" *)
  | Bool_lit of bool                 (* true *)
  | Unop of unop * expr              (* -5 *)
  | Binop of expr * binop * expr     (* a + b *)
  | Id of string                     (* x *)
  | Assign of string * expr          (* x = 4 *)
  | Call of string * expr list       (* add(1, 2) *)
  | ListDecl of expr list            (* [1, 2, 3] *)
  | DictDecl of (expr * expr) list
  (* List Operations *)
  | ListInsert of expr * expr * expr
  | ListPush of expr * expr
  | ListRemove of expr * expr
  | ListPop of expr
  | ListDequeue of expr
  | ListLength of expr
  | ListGet of expr * expr
  | ListSet of expr * expr * expr
  (* Dict Operations *)
  | DictMem of expr * expr
  | DictFind of expr * expr
  | DictMap of expr * expr * expr
  | DictDelete of expr * expr
  | DictSize of expr
  | Noexpr
  | ObjectField of expr * string
  | ObjectAssign of expr * string * expr

(* Statements *)
type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | Break
  | Continue
  | While of expr * stmt
  | ForIn of expr * expr * stmt
  | ForRange of expr * expr * expr * stmt
  | If of expr * stmt * stmt
  | Print of expr
  | ObjectInit of string list

(* Function Declarations *)
type func_decl = {
  fname: string;
  formals: string list;  (* Parameters *)
```

```
  body: stmt list ;         (* Function Body *)
}

(* Program entry point *)
type program = func_decl list
```

Pseudo/compiler/codegen.ml

```
module L = Llvm
module StringMap = Map.Make(String)

let txt_of_type = function
      Sast.Num      -> "Num"
    | Sast.Bool     -> "Bool"
    | Sast.String   -> "String"
    | Sast.Void     -> "Void"
    | Sast.List(_)  -> "List"
    | _ -> "Other types"


let map_lst = ref [|Hashtbl.create 100|]
(* struct to llvm struct definition *)
let struct_types = Hashtbl.create 100
(* (struct ^ struct fields) fields to struct index *)
let struct_field_indexes = Hashtbl.create 100
(* global table mapping struct name -> struct pointer *)
let obj_ptr_map = Hashtbl.create 100
(* table for list len *)
let list_len_hash = Hashtbl.create 42

let rec is_new_var s i =
  if i = Array.length !map_lst then true
  else if Hashtbl.mem (Array.get !map_lst i) s then false
  else is_new_var s (i+1)

let rec lookup s i =
    if i = (Array.length !map_lst)
    then
      raise(Failure("Variable not found"))
    else if Hashtbl.mem (Array.get !map_lst i) s
    then
      Hashtbl.find (Array.get !map_lst i) s
    else lookup s (i+1)

let translate functions obj_hash _ =
  let context     = L.global_context () in
  let the_module = L.create_module context "Pseudo"
  and f_t         = L.double_type context
  and i32_t       = L.i32_type context
```

```
and i8_t          = L.i8_type    context
and i1_t          = L.i1_type    context
and str_t         = L.pointer_type (L.i8_type  context)
and void_t        = L.void_type context

in
let  rec  ltype_of_typ  = function
     Sast.Num  -> f_t
   | Sast.Bool  -> i1_t
   | Sast.String  -> str_t
   | Sast.List(t)  -> ltype_of_typ t
   | Sast.Void -> void_t
   | Sast.Object(id) -> Hashtbl.find struct_types ( string_of_int   id)
   | _ -> raise(Failure("Invalid  Data Type"))

and return_typ = function
     Sast.Num -> f_t
   | Sast.Bool -> i1_t
   | Sast.String  -> str_t
   | Sast.Void -> void_t
   | Sast.List(t)  -> L.pointer_type (ltype_of_typ t)
   | Sast.Object(id) -> L.pointer_type (Hashtbl.find struct_types  ( string_of_int   id))
   | _ -> raise(Failure("Invalid  Data Type"))
in

(* each function  has  its  own StringMap, each scope needs one too *)
let   printf_t  = L.var_arg_function_type i32_t   [|  L.pointer_type  i8_t   |]  in
let   printf_func  = L.declare_function "printf"   printf_t   the_module in
let   prints_t  = L.var_arg_function_type  str_t   [|  L.pointer_type  i8_t   |]  in
let   prints_func  = L.declare_function "puts"  prints_t  the_module in
let   printss_t  = L.var_arg_function_type  str_t   [|  L.pointer_type  i8_t   |]  in
let   printss_func  = L.declare_function "printf"   printss_t   the_module in

let  codegen_struct c =
   let   struct_t  = L.named_struct_type context (string_of_int  c)  in
   let  _ = Hashtbl.add struct_types ( string_of_int   c)  struct_t  in
   let    field_list   = Hashtbl.find obj_hash c in
   let   type_list  = List.map(fun(_, t)  -> let x = (match t with
                                                  Sast.List(t)  -> L.pointer_type (
                                                       ltype_of_typ  t)
                                                | Sast.Object(_) -> L.pointer_type (
                                                    ltype_of_typ  t)
                                                | _ -> ltype_of_typ t)  in  x)   field_list   in
   let  name_list  = List.map (fun (s, _)  -> s)  field_list   in

   let  type_array  = (Array. of_list   type_list )  in
   List . iteri   (fun  i  f  ->
```

```
      let n = (string_of_int c) ^ "." ^ f in
      Hashtbl.add struct_field_indexes n i;
    ) name_list;
    L.struct_set_body struct_t type_array false
in
let _ = Hashtbl.iter (fun k _ -> codegen_struct k) obj_hash in
let function_decls =
  let function_decl m fdecl =
    let name = fdecl.Sast.afname
    and formal_types =
      Array.of_list (List.map (fun (t, _) ->
        (match t with
          | Sast.List(t') -> L.pointer_type (ltype_of_typ t')
          | Sast.Object(_) -> L.pointer_type (ltype_of_typ t)
          | _ -> ltype_of_typ t))
      fdecl.Sast.aformals) in
    let ftype = L.function_type (return_typ fdecl.Sast.return) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functions in

let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.Sast.afname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in
  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and float_format_str = L.build_global_stringptr "%.02f\n" "fmt" builder

  and string_list_iter = L.build_global_stringptr "%s, " "fmt" builder
  and string_list_ender = L.build_global_stringptr "%s" "fmt" builder

  and int_list_iter = L.build_global_stringptr "%d, " "fmt" builder
  and int_list_ender = L.build_global_stringptr "%d" "fmt" builder

  and float_list_iter = L.build_global_stringptr "%.02f, " "fmt" builder
  and float_list_ender = L.build_global_stringptr "%.02f" "fmt" builder in

  let add_formal (t,n) p = L.set_value_name n p;
    let ret = (match t with
      | Sast.Object(_) -> p
      | Sast.List(_) -> p
      | _ -> let local = L.build_alloca (ltype_of_typ t) n builder in
             let _ = L.build_store p local builder in
               local)
    in Hashtbl.add (Array.get !map_lst 0) n ret;
  in
  let _ = List.iter2 add_formal fdecl.Sast.aformals (Array.to_list (L.params
      the_function)) in
(* Return the value for a variable or formal argument *)
```

```
 let add_var s t builder =
   let found_var_arr = Array.map (fun hd -> Hashtbl.mem hd s) !map_lst in
   let find_var = Array.fold_left (fun x y -> x || y) false found_var_arr in
   if find_var = false then
     let local = L.build_alloca (ltype_of_typ t) s builder in
     Hashtbl.add (Array.get !map_lst 0) s local
in
let set_var s v =
  let hd = Array.get !map_lst 0 in
  Hashtbl.add hd s v
in
let string_from_expr = function
  Sast.ANumLit (f, _) -> string_of_float f
| Sast.ABoolLit (b, _) -> string_of_bool b
| Sast.AStringLit (s, _) -> s
| Sast.AVal (s, _) -> s
| _ -> raise (failwith "Expression cannot be converted to string") in
let get_type_from_list d = match d with
  | Sast.List(d) -> d
  | _ -> raise (failwith "not a list")

in let rec expr builder = function
    Sast.ANumLit (i, _) -> L.const_float f_t i
  | Sast.ABoolLit (b, _) -> L.const_int i1_t (if b then 1 else 0)
  | Sast.AStringLit (s, _) -> L.build_global_stringptr s "string" builder
  | Sast.AVal (s, t) -> (match t with
          Sast.List(_) -> lookup s 0
          | Sast.Object(_) -> lookup s 0
          | _ -> L.build_load (lookup s 0) s builder)
  | Sast.ABinop (e1, op, e2, t) ->
      let e1' = expr builder e1 in
      let e2' = expr builder e2
      and float_ops = (match op with
          Ast.Add     -> L.build_fadd
        | Ast.Minus   -> L.build_fsub
        | Ast.Times   -> L.build_fmul
        | Ast.Divide  -> L.build_fdiv
        | Ast.And     -> L.build_and
        | Ast.Or      -> L.build_or
        | Ast.Eq      -> L.build_fcmp L.Fcmp.Oeq
        | Ast.Neq     -> L.build_fcmp L.Fcmp.One
        | Ast.Less    -> L.build_fcmp L.Fcmp.Olt
        | Ast.Leq     -> L.build_fcmp L.Fcmp.Ole
        | Ast.Greater -> L.build_fcmp L.Fcmp.Ogt
        | Ast.Geq     -> L.build_fcmp L.Fcmp.Oge
        | _ -> L.build_fcmp L.Fcmp.Oeq
      )
```

```
    and int_ops = (match op with
        Ast.Add     -> L.build_add
      | Ast.Minus   -> L.build_sub
      | Ast.Times   -> L.build_mul
      | Ast.Divide  -> L.build_sdiv
      | Ast.And     -> L.build_and
      | Ast.Or      -> L.build_or
      | Ast.Eq      -> L.build_icmp L.Icmp.Eq
      | Ast.Neq     -> L.build_icmp L.Icmp.Ne
      | Ast.Less    -> L.build_icmp L.Icmp.Slt
      | Ast.Leq     -> L.build_icmp L.Icmp.Sle
      | Ast.Greater -> L.build_icmp L.Icmp.Sgt
      | Ast.Geq     -> L.build_icmp L.Icmp.Sge
      | _ -> L.build_icmp L.Icmp.Eq
    )
    and str_ops = (match op with
        Ast.Concat -> expr builder (Sast.AStringLit((string_from_expr e1) ^ (
            string_from_expr e2), t))
      | _ -> (L.const_int i32_t 0)
    )
    in
    if ((L.type_of e1' = str_t) && (L.type_of e2' = str_t)) then str_ops
    else if ((L.type_of e1' = f_t) && (L.type_of e2' = f_t)) then float_ops e1' e2'
        "tmp" builder
    else int_ops e1' e2' "tmp" builder
| Sast.AUnop(op, e, _) ->
    let e' = expr builder e in
    (match op with
        Ast.Neg     -> L.build_fneg
      | Ast.Not     -> L.build_not) e' "tmp" builder
| Sast.AAssign (s, e, t) ->
  let e' = expr builder e in
  let _ = (match e with
    | Sast.AListDecl(ael, _) ->
      let _ = Hashtbl.add list_len_hash s (List.length ael) in ()
    | _ -> ())
  in (match t with
    | Sast.Object(_) ->
          let _ = set_var s e' in
          let _ = Hashtbl.add obj_ptr_map s (lookup s 0) in e'
    | Sast.List(_) ->
          let _ = set_var s e' in e'
    | _ -> let _ = add_var s t builder in
          let _ = L.build_store e' (lookup s 0) builder in e')
| Sast.ACall (s, el, t) ->
    let (fdef, _) = StringMap.find s function_decls in
    let actuals = List.rev (List.map (expr builder) (List.rev el)) in
```

```
    let result = (match t with
      Sast.Void -> ""
    | _ -> s ^ "_result") in
    L.build_call fdef (Array.of_list actuals) result builder
| Sast.ANoexpr _ -> L.const_int i32_t 0
| Sast.AObjectAssign (e1, s, e2, t) ->
    let (e2', _) =
      (match t with
        | Sast.Object(_) ->
          (match e2 with
            | Sast.AVal(s, t) -> ((Hashtbl.find obj_ptr_map s), t)
            | _ -> ((expr builder e2), t)
          )
        | _ -> ((expr builder e2), t)
      )
    in
    let obj_name, obj_type = match e1 with
      Sast.AVal(s, t) -> (s, t)
    | _ -> raise (failwith "Not an object variable") in
    let obj_id = match obj_type with
      Sast.Object(id) -> id
    | _ -> -1 in
    let _ =
      if is_new_var obj_name 0 then
        let _ = add_var obj_name obj_type builder in
        let _ = Hashtbl.add obj_ptr_map obj_name (lookup obj_name 0) in
        let field_list = Hashtbl.find obj_hash obj_id in
        List.iter (fun (field_name, field_type) ->
          let e = (match field_type with
            | Sast.Num -> Sast.ANumLit(0.0, Sast.Num)
            | Sast.Bool -> Sast.ABoolLit(false, Sast.Bool)
            | Sast.String -> Sast.AStringLit("", Sast.String)
            | Sast.List(t) -> Sast.AListDecl([], Sast.List(t))
            | Sast.Object(_) -> Sast.ANoexpr(Sast.Num)
            | _ -> raise (failwith "Invalid object field type"))
          in
          if e = Sast.ANoexpr(Sast.Num)
          then ()
          else
            let e' = expr builder e in
            let var_name = (string_of_int obj_id) ^ "." ^ field_name in
            let field_idx = Hashtbl.find struct_field_indexes var_name in
            let _val = L.build_struct_gep (lookup obj_name 0) field_idx var_name
                builder in
            ignore(L.build_store e' _val builder)
        ) field_list
      else ()
```

```
    in
    let var_name = (string_of_int obj_id) ^ "." ^ s in
    let field_idx = Hashtbl.find struct_field_indexes var_name in
    let _ = add_var obj_name obj_type builder in
    let _ = Hashtbl.add obj_ptr_map obj_name (lookup obj_name 0) in
    let _val = L.build_struct_gep (lookup obj_name 0) field_idx var_name builder in
    let _ = L.build_store e2' _val builder in
    e2'
| Sast.AObjectField (e, s, _) ->
    let obj_ptr, obj_type = (match e with
      | Sast.AVal(s_2, t_2) ->
      if (Hashtbl.mem obj_ptr_map s_2) then ((Hashtbl.find obj_ptr_map s_2), t_2)
                                       else ((expr builder e), t_2)
      | Sast.AObjectField(_, _, t_1) ->
        let e' = (expr builder e) in (e', t_1)
      | Sast.AListGet(_, _, t) ->
        let e' = (expr builder e) in (e', t)
      | _ -> raise (failwith "not yet supported"))
    in
    let obj_id = (match obj_type with
      | Sast.Object(id) -> (string_of_int id)
      | _ -> raise (failwith "expected object")
    )
    in
    let search_term = (obj_id ^ "." ^ s) in
    let field_idx = Hashtbl.find struct_field_indexes search_term in
    let _val = L.build_struct_gep obj_ptr field_idx search_term builder in
    L.build_load _val s builder
| Sast.AListDecl(el, d) ->
    let t = get_type_from_list d in
    let initialize_arr p el =
        let map_build x o =
            let x' = expr builder x in
            let arr_ptr = L.build_gep p [| L.const_int i32_t o |]
                "tmp" builder in
            let _ = L.build_store x' arr_ptr builder
            in o + 1
        in List.fold_left (fun o e -> map_build e o) 0 el in

    let typ = match t with
      | Sast.Object(_) -> L.pointer_type (ltype_of_typ t)
      | Sast.List(u_t) -> L.pointer_type (ltype_of_typ u_t)
      | _ -> ltype_of_typ t
    in
    let local = L.build_array_alloca typ
        (L.const_int i32_t (List.length el)) "local" builder
    in let _ = initialize_arr local el in
```

```
    local
| Sast.AListGet(e1, e2, _) ->
        let e1' = expr builder e1 in
        let e2' = expr builder e2 in
        let e2'' = L.build_fptoui e2' i32_t "tmp" builder in
        let pointer = L.build_gep e1' [|e2''|] "tmp" builder in
        L.build_load pointer "tmp" builder
|  Sast.AListSet(e1, e2, e3, _) ->
        let e1' = expr builder e1 in
        let e2' = expr builder e2 in
        let e3' = expr builder e3 in
        let e2'' = L.build_fptoui e2' i32_t "tmp" builder in
        let pointer = L.build_gep e1' [|e2''|] "tmp" builder in
        let _ = L.build_store e3' pointer builder in
        L.build_load pointer "tmp" builder
| Sast.AListLength(e1, _) ->
    (match e1 with
      | Sast.AVal(s, _) -> L.const_float f_t ( float_of_int (Hashtbl.find
      list_len_hash s))
      | _ -> L.const_int i32_t 0)
| Sast.AListPush(e1, e2, _) ->
    let old_size = (match e1 with
      | Sast.AVal(s, _) -> L.const_int i32_t (Hashtbl.find list_len_hash s)
      | _ -> L.const_int i32_t 0)
    in
    let load_values old_arr new_arr final_val arr_len start_pos builder =
      let new_block label =
        let f = L.block_parent (L.insertion_block builder) in
        L.append_block context label f
      in
      let bbcurr = L.insertion_block builder in
      let bbcond = new_block "array.cond" in
      let bbbody = new_block "array.init" in
      let bbdone = new_block "array.done" in
      ignore (L.build_br bbcond builder);
      L.position_at_end bbcond builder;

      (* Counter into the length of the array *)
      let counter = L.build_phi [L.const_int i32_t start_pos, bbcurr] "counter"
          builder in
      L.add_incoming ((L.build_add counter (L.const_int i32_t 1) "tmp" builder),
          bbbody) counter;
      let cmp = L.build_icmp L.Icmp.Slt counter arr_len "tmp1" builder in
      ignore (L.build_cond_br cmp bbbody bbdone builder);
      L.position_at_end bbbody builder;

      (* Assign array position to init_val *)
```

```
      let new_arr_ptr = L.build_gep new_arr [| counter |] "tmp2" builder in
      let old_arr_ptr = L.build_gep old_arr [| counter |] "tmp3" builder in
      let old_val = L.build_load old_arr_ptr "tmp4" builder in
      ignore (L.build_store old_val new_arr_ptr builder);
      ignore (L.build_br bbcond builder);

      L.position_at_end bbdone builder;
      let new_arr_ptr = L.build_gep new_arr [| counter |] "tmp5" builder in
      ignore (L.build_store final_val new_arr_ptr builder);

    in
    let e1' = expr builder e1 and e2' = expr builder e2 in
    let new_size = L.build_add old_size (L.const_int i32_t 1) "new_size" builder in
    let new_arr = L.build_array_alloca (L.type_of e2') new_size "tmp8" builder in
    let _ = load_values e1' new_arr e2' old_size 0 builder in
    let _ = match e1 with
      | Sast.AVal(s, _) -> let _ = set_var s new_arr in
            let new_length = (Hashtbl.find list_len_hash s) in
            let _ = Hashtbl.replace list_len_hash s (new_length + 1) in
            ()
      | _ -> ()
    in e2'
| Sast.AListDequeue(e, _) ->
    let e' = expr builder e in
    let new_arr= L.build_gep e' [|L.const_int i32_t 1 |] "tmp" builder in
    let _ = match e with
        | Sast.AVal(s, _) -> let _ = set_var s new_arr in
              let new_length = (Hashtbl.find list_len_hash s) in
              let _ = Hashtbl.replace list_len_hash s (new_length -1) in ()
        | _ -> ()
    in L.build_load e' "tmp" builder
| Sast.AListPop(e, _) ->
        let list_name =
            (match e with
              | Sast.AVal(s, _) -> s
              | _ -> raise(Failure("pop on undefined list")))
        in
        let e' = expr builder e in
        let list_length = (Hashtbl.find list_len_hash list_name) in
        let _ = Hashtbl.replace list_len_hash list_name ( list_length - 1) in
        let lst_ptr = L.build_gep e' [| L.const_int i32_t ( list_length - 1) |] "
            tmp" builder
        in L.build_load lst_ptr "tmp" builder
| Sast.AListRemove(e1, e2, d) ->
    let list_name =
        (match e1 with
          | Sast.AVal(s, _) -> s
```

```
        | _ -> raise(Failure("Remove on undefined list")))
    in
    let e1' = expr builder e1 in
    let e2' =
      (match e2 with
        | Sast.ANumLit(n, _) -> int_of_float n
        | _ -> raise (failwith "expressions not yet supported")
      )
    in
    let rec copy_w_rm new_arr old_arr old_ctr new_ctr rm_index length =

    if (old_ctr >= length) then new_arr
    else
      (let new_arr_offset = L.build_gep new_arr [| L.const_int i32_t new_ctr |] "
          tmp3" builder in
       if (old_ctr == rm_index) then
          copy_w_rm new_arr old_arr (old_ctr + 1) (new_ctr) rm_index length
       else
       (
          let old_arr_offset = L.build_gep old_arr [| L.const_int i32_t old_ctr |]
              "tmp1" builder in
          let old_val = L.build_load old_arr_offset "tmp2" builder in
          let _ = L.build_store old_val new_arr_offset builder in
          copy_w_rm new_arr old_arr (old_ctr + 1) (new_ctr + 1) rm_index length
       ))
    in
    let new_list_length = ((Hashtbl.find list_len_hash list_name) - 1) in
    let local = L.build_array_alloca (ltype_of_typ d)
      (L.const_int i32_t (new_list_length)) "tmp" builder
    in let new_arr_addr = copy_w_rm local e1' 0 0 e2' (Hashtbl.find list_len_hash
      list_name) in
    let _ = Hashtbl.replace list_len_hash list_name new_list_length in
    let _ = match e1 with
      | Sast.AVal(s, _) -> let _ = set_var s new_arr_addr in ()
      | _ -> ()
    in local
| Sast.AListInsert(e1, e2, e3, d) ->
    let list_name =
      (match e1 with
        | Sast.AVal(s, _) -> s
        | _ -> raise(Failure("Insert into undefined list")))
    in
    let e1' = expr builder e1 in
    let e2' =
      (match e2 with
        | Sast.ANumLit(n, _) -> int_of_float n
        | _ -> raise (failwith "Expressions not yet supported"))
```

```
        in
        let e3' = expr builder e3 in
        let rec copy_w_ins new_arr old_arr ins_val old_ctr new_ctr ins_index length =

        if (old_ctr >= length) then new_arr
        else
            (let new_arr_offset = L.build_gep new_arr [| L.const_int i32_t
            new_ctr |] "tmp3" builder in
            if (new_ctr == ins_index) then
                let _ = L.build_store ins_val new_arr_offset builder in
                copy_w_ins new_arr old_arr ins_val (old_ctr) (new_ctr + 1) ins_index
                    length
            else
            (
                let old_arr_offset = L.build_gep old_arr [| L.const_int i32_t old_ctr |]
                    "tmp1" builder in
                let old_val = L.build_load old_arr_offset "tmp2" builder in
                let _ = L.build_store old_val new_arr_offset builder in
                copy_w_ins new_arr old_arr ins_val (old_ctr + 1) (new_ctr + 1)
                    ins_index length
            ))
        in
        let new_list_length = ((Hashtbl.find list_len_hash list_name) + 1) in
        let local = L.build_array_alloca (ltype_of_typ d)
            (L.const_int i32_t (new_list_length)) "tmp" builder
        in let new_arr_addr = copy_w_ins local e1' e3' 0 0 e2' (Hashtbl.find
            list_len_hash list_name) in
        let _ = Hashtbl.replace list_len_hash list_name new_list_length in
        let _ = match e1 with
            | Sast.AVal(s,_) -> let _ = set_var s new_arr_addr in()
            | _ -> ()
        in e3'
    | _ -> L.const_float f_t 0.0
in
let add_terminal builder f =
  match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
    | None -> ignore (f builder);
in

let rec stmt builder = function
    Sast.ABlock sl -> List.fold_left stmt builder sl
  | Sast.AExpr e -> ignore (expr builder e); builder
  | Sast.AReturn e -> ignore (match fdecl.Sast.return with
      Sast.Void -> L.build_ret_void builder
    | _ -> L.build_ret (expr builder e) builder); builder
  | Sast.AIf (predicate, then_stmt, else_stmt) ->
```

```
  let bool_val = expr builder predicate in
  let merge_bb = L.append_block context "merge" the_function in

  map_lst := Array.append [|Hashtbl.create 100|] !map_lst;

  let then_bb = L.append_block context "then" the_function in
  add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
    (L.build_br merge_bb);

  map_lst := Array.sub !map_lst 1 ((Array.length !map_lst)-1);
  map_lst := Array.append [|Hashtbl.create 100|] !map_lst;

  let else_bb = L.append_block context "else" the_function in
  add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
    (L.build_br merge_bb);

  map_lst := Array.sub !map_lst 1 ((Array.length !map_lst)-1);

  ignore (L.build_cond_br bool_val then_bb else_bb builder);
  L.builder_at_end context merge_bb
| Sast.AWhile (predicate, body) ->
    let pred_bb = L.append_block context "while" the_function in
    ignore (L.build_br pred_bb builder);
    let body_bb = L.append_block context "while_body" the_function in

    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = expr pred_builder predicate in

    let merge_bb = L.append_block context "merge" the_function in
    map_lst := Array.append [|Hashtbl.create 100|] !map_lst;

    add_terminal (stmt (L.builder_at_end context body_bb) body)
      (L.build_br pred_bb);

    map_lst := Array.sub !map_lst 1 ((Array.length !map_lst)-1);

    ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
    L.builder_at_end context merge_bb
| Sast.AForIn (e1, e2, body) ->
    let vname = (match e1 with
      | Sast.AVal(id, _) -> id
      | _ -> raise (failwith "Not a variable"))
    in let length, sast_t = (match e2 with
      | Sast.AVal(id, t) -> Hashtbl.find list_len_hash id, get_type_from_list t
      | Sast.AListDecl(el, t) -> List.length el, get_type_from_list t
      | _ -> 0, Sast.Num)
    in
```

```
    let _ = add_var vname sast_t builder in
    let e2' = expr builder e2 in
    let rec map_iter_body cntr len =
      if cntr < len then
        let offset = L.const_int i32_t cntr in
        let val_ptr = L.build_gep e2' [| offset |] "val_ptr" builder in
        let new_val = L.build_load val_ptr "new_val" builder in
        let _ = L.build_store new_val (lookup vname 0) builder in

        map_lst := Array.append [|Hashtbl.create 100|] !map_lst;
        let _ = stmt builder body
        in map_lst := Array.sub !map_lst 1 ((Array.length !map_lst)-1);

        map_iter_body (cntr + 1) len
      else
        builder
    in map_iter_body 0 length;


| Sast.AForRange (e1, e2, e3, body) ->
    let varname = match e1 with
      | Sast.AAssign(id, _, _) -> id
      | _ -> raise (failwith "Not an assign")
    in let e1' = expr builder e1 and e2' = expr builder e2
    and e3' = expr builder e3 in
    let e1'_ptr = (lookup varname 0) in
    let bool_val = L.build_fcmp L.Fcmp.Olt e1' e2' "lt" builder in
    let incr_ptr = L.build_alloca f_t "incr" builder in
    let _ = L.build_store e3' incr_ptr builder in

    let new_block label =
      let f = L.block_parent (L.insertion_block builder) in
      L.append_block context label f
    in
    let _ = L.insertion_block builder in
    let incr_then = new_block "incr.incr" in
    let incr_else = new_block "incr.decr" in

    let bbcond = new_block "for.cond" in
    let bbbody = new_block "for.init" in
    let bbdone = new_block "for.done" in
    ignore (L.build_cond_br bool_val incr_then incr_else builder);

    (* increasing case*)
    L.position_at_end incr_then builder;
    let incr = L.build_load incr_ptr "incr" builder in
    let e1'' = L.build_fsub e1' incr "tmp" builder in
```

```
  let _ = L.build_store e1'' e1'_ptr builder in
  let _ = L.build_br bbcond builder in

  (* decreasing case *)
  L.position_at_end incr_else builder;
  let incr = L.build_load incr_ptr "incr" builder in
  let e1'' = L.build_fadd e1' incr "tmp" builder in
  let _ = L.build_store e1'' e1'_ptr builder in
  let _ = L.build_br bbcond builder in ();

  (* Iterate *)
  L.position_at_end bbcond builder;
  let load_itr = L.build_load e1'_ptr "iter" builder in
  let incr = L.build_load incr_ptr "incr" builder in
  let incr_itr = L.build_fadd load_itr incr "tmp" builder in
  let _ = L.build_store incr_itr e1'_ptr builder in
  let cmp = L.build_fcmp L.Fcmp.Olt incr_itr e2' "tmp1" builder in
  ignore (L.build_cond_br cmp bbbody bbdone builder);
  L.position_at_end bbbody builder;

  map_lst := Array.append [|Hashtbl.create 100|] !map_lst;

  add_terminal (stmt (L.builder_at_end context bbbody) body)
  (L.build_br bbcond);

  map_lst := Array.sub !map_lst 1 ((Array.length !map_lst)-1);

  L.position_at_end bbdone builder; builder

| Sast.APrint e ->
  let get_field_name obj_field_access =
    let split = Str.split (Str.regexp "\\.") in
      List.hd (List.rev ( split obj_field_access ))
  in
  let e' = (expr builder e) in
  let e_type = L.string_of_lltype (L.type_of e') in

  let list_print iter ender list_len lptr =
    let get_load n =
      let e'_ptr = L.build_gep lptr [| L.const_int i32_t n |] "tmp" builder in
      L.build_load e'_ptr "tmp" builder
    in
    let rec iter_print ctr max acc =
      if ctr = max then acc
      else let new_el = get_load ctr in iter_print (ctr + 1) max (acc @ [new_el])
    in
    let res = iter_print 0 list_len [] in
```

```
  let rec print_list l =
    match l with
      | [] -> L.build_call printss_func
          [| expr builder (Sast.AStringLit("", Sast.String)) |] "puts" builder
      | [el] -> L.build_call printf_func [| ender; el |] "puts" builder
      | hd :: tail ->
          let _ = L.build_call printf_func [| iter; hd|] "puts" builder in
          print_list tail
  in
  let lbrace = expr builder (Sast.AStringLit("[", Sast.String)) in
  let rbrace = expr builder (Sast.AStringLit("]\n", Sast.String)) in
  let _ = L.build_call printss_func [| lbrace |] "puts" builder in
  let _ = print_list res in
  let _ = L.build_call printss_func [| rbrace |] "puts" builder in
  builder
in
if e_type = L.string_of_lltype f_t then
  (ignore(L.build_call printf_func [| float_format_str; e' |]
    "printf" builder); builder)
else if e_type = L.string_of_lltype i1_t then
  (ignore(L.build_call printf_func [| int_format_str; e' |]
    "printf" builder); builder)
else if e_type = L.string_of_lltype str_t then
  (ignore(L.build_call prints_func [|e'|]
    "puts" builder); builder)

else if e_type = L.string_of_lltype (L.pointer_type str_t) then
  let length = match e with
    | Sast.AVal(s, _) -> Hashtbl.find list_len_hash s
    | Sast.AListDecl(el, _) -> List.length el
    | _ -> 0
  in
   list_print string_list_iter string_list_ender length e'

else if e_type = L.string_of_lltype (L.pointer_type f_t) then
  let length = match e with
    | Sast.AVal(s, _) -> Hashtbl.find list_len_hash s
    | Sast.AListDecl(el, _) -> List.length el
    | _ -> 0
  in
   list_print float_list_iter float_list_ender length e'

else if e_type = L.string_of_lltype (L.pointer_type i1_t) then
  let length = match e with
    | Sast.AVal(s, _) -> Hashtbl.find list_len_hash s
    | Sast.AListDecl(el, _) -> List.length el
    | _ -> 0
```

```
    in
     list_print    int_list_iter    int_list_ender  length e'

  else
    let  obj_name, obj_type = (match e with
      Sast.AVal(s,  t)  -> (s, t)
    | Sast.AObjectField(_, s,  t)  -> (s, t)
    | _ -> raise (failwith "Not a variable or an object variable") )in
    let  obj_id = (match obj_type with
      Sast.Object(id) -> id
    | _ -> -1) in
    let  split  = Str.split  (Str.regexp "\\.") in
    let  obj_ptr = Hashtbl.find obj_ptr_map obj_name in
    let  search_terms = Hashtbl.fold (fun k _ acc -> k :: acc)  struct_field_indexes
         []  in
    let  passed_terms = List. filter  (fun term -> (List.hd (split term)) = (
         string_of_int  obj_id)) search_terms in
    let  field_idxs  = List.map (fun k -> Hashtbl.find struct_field_indexes k)
         passed_terms in
    let  val_arr  = List.map2 (fun index field  ->
      L.build_struct_gep  obj_ptr  index  field  builder)  field_idxs  passed_terms in
    let  print_arr  = List.map2 (fun struct_val  field  ->
      L.build_load  struct_val  (get_field_name  field )  builder)  val_arr  passed_terms
    in
    List.iter2  (fun x y ->
      let  field_name = expr builder (Sast.AStringLit(obj_name ^ "." ^ (
           get_field_name y) ^ ": ", Sast.String)) in
      let  e_type = L. string_of_lltype  (L.type_of x) in
      if  e_type = L. string_of_lltype  f_t  then
        (ignore(L. build_call  printss_func  [|  field_name  |]  "puts" builder);
        ignore(L. build_call  printf_func  [|  float_format_str ;  x  |]  "printf"  builder))
      else  if  e_type = L. string_of_lltype  i1_t  then
        (ignore(L. build_call  printss_func  [|  field_name  |]  "puts" builder);
        ignore(L. build_call  printf_func  [|  int_format_str ;  x  |]  "printf"  builder))
      else  if  e_type = L. string_of_lltype  str_t  then
        (ignore(L. build_call  printss_func  [|  field_name  |]  "puts" builder);
        ignore(L. build_call  prints_func  [|x|]  "puts" builder))
    )  print_arr  passed_terms; builder
| Sast.AObjectInit sl ->
    let  _ = List. iter  (fun (obj,  obj_type) ->
      if  not (Hashtbl.mem obj_ptr_map obj) then
        let  obj_id  = (match obj_type with
          Sast.Object(id) -> id
        | _ -> -1
        ) in
        let  _ = add_var obj obj_type builder in
        let  _ = Hashtbl.add obj_ptr_map obj (lookup obj 0) in
```

```
            let  field_list  = Hashtbl.find obj_hash obj_id in
            List.iter (fun (field_name, field_type) ->
              let e = (match field_type with
                  Sast.Num    -> Sast.ANumLit(0.0, Sast.Num)
                | Sast.Bool   -> Sast.ABoolLit(false, Sast.Bool)
                | Sast.String -> Sast.AStringLit("", Sast.String)
                | Sast.List(t) -> Sast.AListDecl([], Sast.List(t))
                | Sast.Object(_) -> Sast.ANoexpr(Sast.Num)
                | _ -> raise (failwith "Invalid object field type"))
              in
              if e = Sast.ANoexpr(Sast.Num)
              then ()
              else
                let e' = expr builder e in
                let var_name = (string_of_int obj_id) ^ "." ^ field_name in
                let field_idx = Hashtbl.find struct_field_indexes var_name in
                let _val = L.build_struct_gep (lookup obj 0) field_idx var_name builder
                    in
                ignore(L.build_store e' _val builder)
            ) field_list
        ) sl in builder
    | _ -> builder
  in
  let builder = stmt builder (Sast.ABlock fdecl.Sast.abody) in
  add_terminal builder (match fdecl.Sast.return with
      Sast.String -> L.build_ret (L.build_global_stringptr "" "tmp" builder)
    | Sast.Num -> L.build_ret (L.const_float f_t 0.)
    | Sast.Bool -> L.build_ret (L.const_int i1_t 0)
    | Sast.Object(_) ->
      let arbitrary_obj = List.hd (Hashtbl.fold (fun _ v acc -> [v] @ acc) obj_ptr_map
          []) in
      L.build_ret arbitrary_obj
    | Sast.List(_) -> L.build_ret (expr builder (Sast.AListDecl([Sast.ANumLit(0., Sast.
        Num)], Sast.List(Sast.Num))))
    | _ -> L.build_ret_void)
in
List.iter build_function_body functions;
the_module
```

Pseudo/compiler/infer.ml

```
open Ast
open Sast

(* Map from variable name to its type *)
module NameMap = Map.Make(String)

(* Set of object names that are the same type *)
module ObjectTypeSet = Set.Make(String)
```

```
(* module ObjectFieldSet = Set.Make((String * Sast.data_type)) *)
module ObjectFieldSet = Set.Make(String)

module StringSet = Set.Make(String)

type environment = data_type NameMap.t
type id = string
type substitutions = (id * data_type) list

let rec infer_all (ast: Ast.program) =
  let sast_types = infer_types ast in
  let sast_objects, objs = infer_objs sast_types in
  (sast_objects, objs, obj_to_id_hash)

and logging = false

and log (msg: string) =
  if logging then print_endline msg else ()

and print_field_hash () =
  let _ = log "<field_hash>" in
  let _ = Hashtbl.iter (fun k v -> log (k ^ "->" ^ (string_of_data_type v))) field_hash
      in
  let _ = log "</field_hash>\n" in ()

and print_set (set: ObjectTypeSet.t) =
  let _ = log "<set>" in
  let list = ObjectTypeSet.elements set in
  let _ = List.iter log list in
  let _ = log "</set>" in ()

and print_set_list ( set_list : ObjectTypeSet.t list) =
  let _ = log "\n<set list>" in
  let _ = List.iter print_set set_list in
  let _ = log "</set list>\n" in ()

and print_obj_to_fields_hash () =
  let _ = log "\n<obj_to_fields_hash>" in
  let _ = Hashtbl.iter (fun k v -> let _ = log (k ^ "->") in let _ = List.iter log v in
      ()) obj_to_fields_hash in
  let _ = log "</obj_to_fields_hash>\n" in ()

and print_obj_to_id_hash () =
  let _ = log "\n<obj_to_id_hash>" in
  let _ = Hashtbl.iter (fun k v -> let _ = log (k ^ "->" ^ (string_of_int v)) in ())
      obj_to_id_hash in
```

```
  let _ = log "</obj_to_id_hash>\n" in ()

and print_id_to_fields_hash () =
  let _ = log "\n<id_to_fields_hash>" in
  let _ = Hashtbl.iter (fun k v -> let _ = log ((string_of_int k) ^ "->") in let _ =
      List.iter log v in ()) id_to_fields_hash in
  let _ = log "</id_to_fields_hash>\n" in ()

and print_id_to_fields_typified_hash () =
  let _ = log "\n< id_to_fields_typified_hash >" in
  let _ = Hashtbl.iter (fun k v -> let _ = log ((string_of_int k) ^ "->") in List.iter (
      fun x -> log ("(" ^ (fst x) ^ "," ^ string_of_data_type (snd x) ^ ")")) v)
       id_to_fields_typified_hash in
  let _ = log "</ id_to_fields_typified_hash >\n" in ()


(*
infer_types: Driver for type inference - given a program's AST, turns it into an sast
Params:
  ast: AST for a Pseudo program (list of fdecls)
Returns:
  fdecls: SAST for a Pseudo program (list of afdecls)
*)
and infer_types (ast: Ast.program) =
  let _ = List.iter add_func_to_map ast
  and _ = if not (Hashtbl.mem func_decl_hash "main") then
    raise (failwith ("main function not found"))
  (* Begin with main() function *)

  and _ = infer_types_function (Hashtbl.find func_decl_hash "main") []
  and _ = add_afunc_decls()
  and _ = print_field_hash ()
  and fdecls = order_fdecls ast in
  fdecls


and order_fdecls (ast: Ast.program) =
  let _ = Hashtbl.add func_order_hash "main" 0
  and fdecl_tups = List. fold_left get_name_to_order [] ast
  in let sorted_tups = List.sort compare fdecl_tups
  in let fdecls = List.map drop_order sorted_tups in fdecls

and get_name_to_order acc fdecl =
    if Hashtbl.mem afunc_decl_hash fdecl.fname then
      (let tup = (Hashtbl.find afunc_decl_hash fdecl.fname, Hashtbl.find
                func_order_hash fdecl.fname)
      in tup :: acc)
```

```
    else acc

(* drop the order from the tuple *)
and drop_order x = match x with
    (fdecl, _) -> fdecl

(* compare by second key *)
and compare x y =
  snd x - snd y
(*
get_afunc_decl: Given a list of key (function name) value (function type) pairs,
    outputs an afdecl list
Looks through the global hash tables to collect information about functions
Params:
  k: Function Name
  v: Function Type
Returns:
  l: List of afdecls representing the sast
*)

and add_afunc_decl (k: string) (b: astmt list) =
  let ret = Hashtbl.find func_type_hash k
  and aforms = Hashtbl.find func_formals_hash k in
  let new_fdecl = {
    afname = k;
    aformals = aforms;
    abody = b;
    return = ret
  } in Hashtbl.add afunc_decl_hash new_fdecl.afname new_fdecl


(*
 collect_afunc_decls : Driver to collect a list of afdecls from the global hash maps
Returns:
  l: List of afdecls representing the sast
*)
and add_afunc_decls ()=
  Hashtbl.iter add_afunc_decl func_body_hash

(*
add_func_to_map: Adds a funcdecl to the global declaration hash
This hash map is isued to get parameters when a call is detected, mandating that that
    function be analyzed
Returns:
  unit: Never actually used
*)
and add_func_to_map (fd: func_decl) =
```

```
  Hashtbl.add func_decl_hash fd.fname fd

(*
gen_new_placeholder: Uses characters for placeholder types in inference for a single
    statement
Returns:
  c1: Character used as a placeholder
*)
and gen_new_placeholder () =
  let c1 = !type_variable in
  incr type_variable; T(Char.escaped (Char.chr c1))

and gen_new_index (): int =
  let n1 = !index in
  let _ = incr index in n1


(* Placeholder character *)
and type_variable = ref (Char.code 'a')

and index = ref 1

(* Hash Table for function declarations *)
and func_decl_hash = Hashtbl.create 10
(* Hash Table for function types after inference *)
and func_type_hash = Hashtbl.create 10
(* Hash Table for annotated formal parameters [eg add(a, b) -> int add(int a, int b)] *)
and func_formals_hash = Hashtbl.create 10
(* Hash Table for function bodies, in the form of list of annotated statements *)
and func_body_hash = Hashtbl.create 10
(* Hash Table for function order *)
and func_order_hash = Hashtbl.create 10
(* Hash Table for function decl *)
and afunc_decl_hash = Hashtbl.create 10
(* Hash Table for object fields *)
and field_hash = Hashtbl.create 30

and fun_vis_rec = Hashtbl.create 10
and func_env_hash = Hashtbl.create 10


and string_of_data_type (input: data_type) : string =
  match input with
  | Num -> "Num"
  | Bool -> "Bool"
  | String -> "String"
  | Void -> "Void"
```

```
  | Object(id) -> "Object of type " ^ string_of_int id
  | List(t) -> "List of " ^ string_of_data_type t
  | Dict(k, v) -> "Dict of (" ^ string_of_data_type k ^ ": " ^ string_of_data_type v ^
      ")"
  | Undetermined -> "Undetermined"
  | T(t) -> t

and string_of_aexpr (input: aexpr) : string =
  match input with
  | ANumLit(_, d) -> string_of_data_type d ^ " literal"
  | ABoolLit(_, d) -> string_of_data_type d ^ " literal "
  | AStringLit(_, d) -> string_of_data_type d ^ " literal "
  | AVal(_, d) -> "My AVal of type " ^ string_of_data_type d
  | AUnop(_, _, d) -> "My AUnop of type: " ^ string_of_data_type d
  | ABinop(_, _, _, d) -> "My ABinop of type: " ^ string_of_data_type d
  | AAssign(_, _, d) -> "My AAssign of type: " ^ string_of_data_type d
  | ACall(_, _, d) -> "My ACall of type: " ^ string_of_data_type d
  (* List Ops *)
  | AListDecl(_, d) -> "My AListDecl of type " ^ string_of_data_type d
  | AListInsert(_, _, _, d) -> "My AListInsert of type " ^ string_of_data_type d
  | AListPush(_, _, d) -> "My AListPush of type " ^ string_of_data_type d
  | AListRemove(_, _, d) -> "My AListRemove of type " ^ string_of_data_type d
  | AListPop(_, d) -> "My AListPop of type " ^ string_of_data_type d
  | AListDequeue(_, d) -> "My AListDequeue of type " ^ string_of_data_type d
  | AListLength(_, d) -> "My AListLength of type " ^ string_of_data_type d
  | AListGet(_, _, d) -> "My AListGet of type " ^ string_of_data_type d
  | AListSet(_, _, _, d) -> "My AListSet of type " ^ string_of_data_type d
  (* Dict Ops *)
  | ADictDecl(_, _, d) -> "My ADictDecl of type " ^ string_of_data_type d
  | ADictMem(_, _, d) -> "My ADictMem of type " ^ string_of_data_type d
  | ADictDelete (_, _, d) -> "My ADictDelete of type " ^ string_of_data_type d
  | ADictSize(_, d) -> "My ADictSize of type " ^ string_of_data_type d
  | ADictFind(_, _, d) -> "My ADictFind of type " ^ string_of_data_type d
  | ADictMap(_, _, _, d) -> "My ADictMap of type " ^ string_of_data_type d
  (* Obj Ops *)
  | AObjectAssign(_, _, _, d) -> "My AObjectAssign of type " ^ string_of_data_type d
  | AObjectField(_, _, d) -> "My AObjectField of type " ^ string_of_data_type d
  | _ -> raise (failwith("Could not get string of aexpr"))

and type_of_aexpr (input: aexpr) : data_type =
  match input with
  | ANumLit(_, d) -> d
  | ABoolLit(_, d) -> d
  | AStringLit(_, d) -> d
  | AVal(_, d) -> d
  | AUnop(_, _, d) -> d
  | ABinop(_, _, _, d) -> d
```

```
  | AAssign(_, _, d) -> d
  | ACall(_, _, d) -> d
  (* List Ops *)
  | AListDecl(_, d) -> d
  | AListInsert(_, _, _, d) -> d
  | AListPush(_, _, d) -> d
  | AListRemove(_, _, d) -> d
  | AListPop(_, d) -> d
  | AListDequeue(_, d) -> d
  | AListLength(_, d) -> d
  | AListGet(_, _, d) -> d
  | AListSet(_, _, _, d) -> d
  (* Dict Ops *)
  | ADictDecl(_, _, d) -> d
  | ADictMem(_, _, d) -> d
  | ADictDelete (_, _, d) -> d
  | ADictSize(_, d) -> d
  | ADictFind(_, _, d) -> d
  | ADictMap(_, _, _, d) -> d
  (* Obj Ops *)
  | AObjectAssign(_, _, _, d) -> d
  | AObjectField(_, _, d) -> d
  | _ -> raise (failwith("Could not determine type of aexpr"))

and is_object_list_decl (ae: aexpr) : bool =
  match ae with
    | AListDecl(_, typ) ->
      (match typ with
        | List(typ) ->
          (match typ with
            | Object(_) -> true
            | Undetermined -> true
            | _ -> false
          )
        | _ -> false
      )
    | _ -> false

(*
assign_add_map: After an assignment, add that variable to the local function (string->
    data type) environment
Params:
  env: Function environment with keys of variables and values of types corresponding to
      those variables
  aexpr (AAssign): Annotated assignment of id, annotated expression, data type of
      expression (and therefore variable)
Returns:
```

```
  env: The environment after the variable has been added to the map with its data type
*)
and assign_add_map (env: environment) (input: aexpr) =
  match input with
    | AAssign(id, _, d) ->
      if (NameMap.mem id env && (NameMap.find id env <> d)) then raise (failwith
          ("mismatched types for id " ^ id))
      else NameMap.add id d env
    | AObjectAssign(_, f, _, d) ->
      if (Hashtbl.mem field_hash f && (Hashtbl.find field_hash f <> d)) then raise (
          failwith ("mismatched types for field " ^ f))
      else let _ = Hashtbl.add field_hash f d in env
    | AVal(id, d) ->
      if (NameMap.mem id env && (NameMap.find id env <> d)) then raise (failwith
          ("mismatched types for object " ^ id))
    else NameMap.add id d env
    | _ -> raise (failwith ("Expected assign"))

(*
annotate_expr: 1st step in inference - annotate a variable, including the base cases of
    basic data types
Params:
  e: AST expression to be annotated
  env: Function environment with keys of variables and values of types corresponding to
      those variables
  aexpr (AAssign): Annotated assignment of id, annotated expression, data type of
      expression (and therefore variable)
Returns:
  aexpr: Annotated expression of the expression
*)
(* Step 1: Annotate Expressions *)
and annotate_expr (e: expr) (env: environment) : aexpr =
  match e with
  (* Base Cases *)
  | Num_lit(n) -> ANumLit(n, Num)
  | Bool_lit(b) -> ABoolLit(b, Bool)
  | String_lit (s) -> AStringLit(s, String)
  (* Variable -> Find variable in map, if it has not been declared yet throw an error *)
  | Id(x) -> if NameMap.mem x env
    then AVal(x, NameMap.find x env)
    else raise (failwith "Variable not defined")

  (* OBJECTS *)
  | ObjectField(o, f) -> if Hashtbl.mem field_hash f
    then let et = annotate_expr o env in AObjectField(et, f, Hashtbl.find field_hash f)
    else raise (failwith "Object field not defined")
  | ObjectAssign(o, f, e) ->
```

```
  let et1 = match o with
    | Id(x) -> if NameMap.mem x env
      then AVal(x, NameMap.find x env)
      else AVal(x, Object(-1))
    | _ -> annotate_expr o env
  and et2 = annotate_expr e env
  and new_type = gen_new_placeholder () in
  AObjectAssign(et1, f, et2, new_type)

(* Binop -> Annotate the left and right sides of the binop first before combining
   into an annotated Binop *)
| Binop(e1, op, e2) ->
  let et1 = annotate_expr e1 env
  and et2 = annotate_expr e2 env
  and new_type = gen_new_placeholder () in
  (match op with
    | Combine ->
      let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
      let t1 = type_of ae1 and t2 = type_of ae2 in
      let x = match t1 with
        | List(u) -> u
        | _ -> raise (failwith "Not a list")
      and y = match t2 with
        | List(v) -> v
        | _ -> raise (failwith "Not a list")
      in let x_base = (base_type_of x 0) and y_base = (base_type_of y 0) in
        if snd x_base <> snd y_base then raise (failwith "mixed embedded lists")
        else
          let x_type = fst x_base and y_type = fst y_base in
            if ((x_type = Undetermined) && (y_type = Undetermined)) then
              ABinop(ae1, op, ae2, List(x))
            else if x_type = Undetermined then ABinop(ae1, op, ae2, List(y))
            else ABinop(ae1, op, ae2, List(x))
    | _ -> ABinop(et1, op, et2, new_type))
(* Unop -> Annotate the right side before combining into an annotated Unop *)
| Unop(op, e) ->
  let et = annotate_expr e env
  and new_type = gen_new_placeholder () in
  AUnop(op, et, new_type)
(* Assign -> Annotate the right side before combining into an annotated Assign *)
| Assign(id, e) ->
  let et = annotate_expr e env
  and new_type = gen_new_placeholder () in
  AAssign(id, et, new_type)

(* LIST OPS *)
| ListDecl(el) ->
```

```
    (match el with
      | [] -> AListDecl([], List(Undetermined))
      | _ -> let ael = List.map (fun x -> fst (infer env x)) el in
        let t = assert_list_type ael in
        AListDecl(ael, List(t))
    )
  | ListInsert(e1, e2, e3) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) and ae3 = fst (infer env
        e3) in
    AListInsert(ae1, ae2, ae3, gen_new_placeholder())
  | ListPush(e1, e2) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
    AListPush(ae1, ae2, gen_new_placeholder())
  | ListRemove(e1, e2) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
    AListRemove(ae1, ae2, gen_new_placeholder())
  | ListPop(e) ->
    let ae = fst (infer env e) in
    AListPop(ae, gen_new_placeholder())
  | ListDequeue(e) ->
    let ae = fst (infer env e) in
    AListDequeue(ae, gen_new_placeholder())
  | ListLength(e) ->
    let ae = fst (infer env e) in
    AListLength(ae, gen_new_placeholder())
  | ListGet(e1, e2) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
    AListGet(ae1, ae2, gen_new_placeholder())
  | ListSet(e1, e2, e3) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) and ae3 = fst (infer env
        e3) in
    AListSet(ae1, ae2, ae3, gen_new_placeholder())

  (* DICT OPS *)
  | DictDecl(tl) ->
    (match tl with
      | [] -> ADictDecl([], [], Dict(Undetermined, Undetermined))
      | _ ->
        let kl = List.map (fun x -> fst x) tl
        and vl = List.map (fun x -> snd x) tl
        in let akl = List.map (fun x -> fst (infer env x)) kl
        and avl = List.map (fun x -> fst (infer env x)) vl in
        let kt = assert_basic_list_type akl
        and vt = assert_list_type avl
        in ADictDecl(akl, avl, Dict(kt, vt))
    )
  | DictMem(e1, e2) ->
```

```
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
    ADictMem(ae1, ae2, Bool)
  | DictDelete(e1, e2) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
    ADictDelete(ae1, ae2, Bool)
  | DictSize(e) ->
    let ae = fst (infer env e) in ADictSize(ae, Num)
  | DictFind(e1, e2) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) in
    ADictFind(ae1, ae2, gen_new_placeholder())
  | DictMap(e1, e2, e3) ->
    let ae1 = fst (infer env e1) and ae2 = fst (infer env e2) and ae3 = fst (infer env
        e3) in
    ADictMap(ae1, ae2, ae3, Bool)

  (* Call -> search to see if the function has already been annotated, if not call
      inference on it *)
  | Call(id, args) ->
    let _ = if not (Hashtbl.mem func_order_hash id) then
      let new_id = gen_new_index() in
        (Hashtbl.add func_order_hash id new_id)
    in
    (* annotate the arguments *)
    let new_args = List.map (fun x -> fst(infer env x)) args in
    (* Function already inferred *)
    if Hashtbl.mem func_type_hash id then
      ACall(id, new_args, Hashtbl.find func_type_hash id)
    (* Function not inferred yet *)
    else if not (Hashtbl.mem fun_vis_rec id) then
      let _ = Hashtbl.add fun_vis_rec id 0 in
      (* Infer the argument types to pass into the environment of the function abbout
          to be inferred *)
      let params_list = List.map (fun x -> type_of (fst (infer env x))) args in
      (* Call inference on the function *)
      let _ = infer_types_function(Hashtbl.find func_decl_hash id) params_list in
      ACall(id, new_args, Hashtbl.find func_type_hash id)
    else
      ACall(id, new_args, Undetermined)
  | _ -> raise (failwith "Could not annotate expr")


(*
type_of: Get the type of an annotated expression
Params:
  ae: annotated expression to get the type of
Returns:
  t: the type of the annotated expression
```

```
*)
and type_of (ae: aexpr): data_type =
  match ae with
  | ANumLit(_, t) | ABoolLit(_, t) | AStringLit(_, t) -> t
  | AListDecl(_, t) -> t
  | ADictDecl(_, _, t) -> t
  | AVal(_, t) -> t
  | ABinop(_, _, _, t) -> t
  | AUnop(_, _, t) -> t
  | AAssign(_, _, t) -> t
  | ACall(_, _, t) -> t
  (* LIST OPS *)
  | AListInsert(_, _, _, t) -> t
  | AListPush(_, _, t) -> t
  | AListRemove(_, _, t) -> t
  | AListPop(_, t) -> t
  | AListDequeue(_, t) -> t
  | AListLength(_, t) -> t
  | AListGet(_, _, t) -> t
  | AListSet(_, _, _, t) -> t
  (* DICT OPS *)
  | ADictMem(_, _, t) -> t
  | ADictDelete(_, _, t) -> t
  | ADictSize(_, t) -> t
  | ADictFind(_, _, t) -> t
  | ADictMap(_, _, _, t) -> t
  (* OBJECT OPS *)
  | AObjectField(_, _, t) -> t
  | AObjectAssign(_, _, _, t) -> t
  | _ -> Undetermined

and base_type_of (d: data_type) (acc: int): (data_type * int) =
  match d with
  | Num -> Num, acc
  | Bool -> Bool, acc
  | String -> String, acc
  | Void -> Void, acc
  | Object(id) -> Object(id), acc
  | List(t) -> base_type_of t (acc + 1)
  | Dict(t1, t2) -> Dict(t1, t2), acc
  | Undetermined -> Undetermined, acc
  | T(t) -> T(t), acc

and get_list_type (d: data_type): data_type =
  match d with
  | List(t) -> t
  | _ -> raise (failwith "Expected list")
```

```
and get_basic_type (ael: aexpr list) =
  match ael with
    | [] -> List(Undetermined)
    | [l] -> type_of l
    | hd :: tail ->
      (match type_of hd with
        | Num -> Num
        | Bool -> Bool
        | String -> String
        | Void -> Void
        | Object(id) -> Object(id)
        | Dict(k, v) ->
          if k = Undetermined then get_basic_type tail else Dict(k, v)
        | List(t) ->
          if (fst (base_type_of t 0)) = Undetermined then get_basic_type tail else List(
              t)
        | Undetermined -> get_basic_type tail
        | T(t) -> T(t)
      )

and compare_list_type (x: aexpr) (t: data_type): int =
  let x_base = base_type_of (type_of x) 0
  and t_base = base_type_of t 0 in
  if snd x_base <> snd t_base then raise (failwith "mixed embedded lists: 1st check")
      else
    if (fst x_base = fst t_base || fst x_base = Undetermined) then 1 else 0

and assert_list_type (ael: aexpr list) =
  let t = get_basic_type ael in
  let same_type = List.map (fun x-> compare_list_type x t) ael in
  let not_same = List.mem 0 same_type in
  if not_same then raise (failwith "Mismatched list or dict key types") else t

and assert_basic_list_type (ael: aexpr list) =
  let t = type_of (List.hd ael) in
  let same_type = List.map (fun x-> (type_of x) = t) ael in
  let not_same = List.mem false same_type in
  if not_same then raise (failwith "Mismatched key types") else t

(*
collect_expr: collect constraints (ie 'a = 'b, 'b = Num) within an annotated
    expression
Also conveniently handles semantic analysis for binary operations down the line when
    types are mismatched
Params:
  ae: annotated expression to form constraints for
```

```
Returns:
    constraint_list : List of constraints, where each element is a data type equality of
        two types
*)
(* Step 2: Collect Constraints *)
and collect_expr (ae: aexpr) : (data_type * data_type) list =
  match ae with
  (* Literals are already fine *)
  | ANumLit(_) | ABoolLit(_) | AStringLit(_) | AListDecl(_) | ADictDecl(_) | AVal(_) |
      ACall(_) -> []
  | AObjectField(_) -> []
  | ABinop(ae1, op, ae2, t) ->
    let et1 = type_of ae1 and et2 = type_of ae2 in
    let opc = match op with
      (* Left and right side must be num, final return is a num *)
      | Add | Minus | Times | Divide -> [(et1, Num); (et2, Num); (t, Num)]
      (* Left and right side must be same type, final return is a boolean *)
      | Greater | Less | Geq | Leq | Eq | Neq -> [(et1, et2); (t, Bool)]
      (* Left and right side must be booleans, final return is a boolean *)
      | And | Or -> [(et1, Bool); (et2, Bool); (t, Bool)]
      (* Left and right side must be strings, final return is a string *)
      | Concat -> [(et1, String); (et2, String); (t, String)]
      (* Lists must be of same type *)
      | Combine -> []
    in
    (* Combine constraints of left side, right side, and binop constraints themselves
       *)
    (collect_expr ae1) @ (collect_expr ae2) @ opc
  | AUnop(op, ae, t) ->
    let et = type_of ae in
    let opc = match op with
      (* Right side must be a num, final return is a num *)
      | Neg -> [(et, Num); (t, Num)]
      | Not -> [(et, Bool); (t, Bool)]
    (* Combine constraints of right side and unop constraints themselves *)
    in (collect_expr ae) @ opc

  (* LIST OPS *)
  | AListInsert(ae1, ae2, ae3, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 and t3 = type_of ae3 in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in if x = Undetermined
      then [(t2, Num); (t, t3)]
      else [(x, t3); (t2, Num); (t, t3)]
  | AListPush(ae1, ae2, t) ->
```

```
    let t1 = type_of ae1 and t2 = type_of ae2 in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in if x = Undetermined
      then [(t, t2)] else [(x, t2); (t, t2)]
  | AListRemove(ae1, ae2, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in [(t2, Num); (t, x)]
  | AListPop(ae, t) ->
    let t1 = type_of ae in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in [(t, x)]
  | AListDequeue(ae, t) ->
    let t1 = type_of ae in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in [(t, x)]
  | AListLength(ae, t) ->
    let t1 = type_of ae in
    let _ = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in [(t, Num)]
  | AListGet(ae1, ae2, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
    in [(t2, Num); (t, x)]
  | AListSet(ae1, ae2, ae3, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 and t3 = type_of ae3 in
    let x = match t1 with
      | List(u) -> u
      | _ -> raise (failwith "Not a list")
     in [(x, t3); (t2, Num); (t, t3)]
  (* DICT OPS *)
  | ADictMem(ae1, ae2, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 in
    let x = match t1 with
      | Dict(t, _) -> t
```

```
      | _ -> raise (failwith "Not a map")
    in [(x, t2); (t, Bool)]
  | ADictDelete(ae1, ae2, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 in
    let x = match t1 with
      | Dict(k, _) -> k
      | _ -> raise (failwith "Not a map")
    in [(x, t2); (t, Bool)]
  | ADictSize(ae, t) ->
    let aet = type_of ae in
    let _ = match aet with
      | Dict(_, _) -> ()
      | _ -> raise (failwith "Not a map")
    in [(t, Num)]
  | ADictFind(ae1, ae2, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 in
    let x, y = match t1 with
      | Dict(k, v) -> k, v
      | _ -> raise (failwith "Not a map")
    in [(x, t2); (t, y)]
  | ADictMap(ae1, ae2, ae3, t) ->
    let t1 = type_of ae1 and t2 = type_of ae2 and t3 = type_of ae3 in
    let x, y = match t1 with
      | Dict(k, v) -> k, v
      | _ -> raise (failwith "Not a map")
    in if x = Undetermined && y = Undetermined then
      [(t, Bool)] else [(x, t2); (y, t3); (t, Bool)]

  | AAssign(_, ae, lhs_type) ->
    (* lhs = variable name type (if reusing variable, it must be reused as same type, ie
       static inference) *)
    (* collect constraints of the right hand side, then type of left and right sides
       must be equal *)
    let rhs_type = type_of ae in
    (collect_expr ae) @ [(rhs_type, lhs_type)]
  | AObjectAssign(_, _, ae, lhs_type) ->
    let rhs_type = type_of ae in
    (collect_expr ae) @ [(rhs_type, lhs_type)]
  | _ -> raise (failwith "Not an aexpr in collect")

(* Step 3: Unification *)
and substitute (u: data_type) (x: id) (t: data_type) : data_type =
  match t with
  | Num | Bool | String | Void| Undetermined | Object(_) | List(_) | Dict(_) -> t
  | T(c) -> if c = x then u else t

and apply (subs: substitutions) (t: data_type) : data_type =
```

```
  List.fold_right (fun (x, u) t -> substitute u x t) subs t

and unify (constraints: (data_type * data_type) list) : substitutions =
  match constraints with
  | [] -> []
  | (x, y) :: xs ->
    (* generate substitutions of the rest of the list *)
    let t2 = unify xs in
    (* resolve the LHS and RHS of the constraints from the previous substitutions *)
    let t1 = unify_one (apply t2 x) (apply t2 y) in
    t1 @ t2

and unify_one (t1: data_type) (t2: data_type) : substitutions =
  match t1, t2 with
  | Num, Num | Bool, Bool | String, String | Undetermined, _ | _, Undetermined -> []
  | T(x), z | z, T(x) -> [(x, z)]
  | _ -> if t1 = t2 then []
    else raise (failwith "mismatched types")

and apply_expr (subs: substitutions) (ae: aexpr) (env: environment): (aexpr *
    environment) =
  match ae with
  | ABoolLit(b, t) -> ABoolLit(b, apply subs t), env
  | ANumLit(n, t) -> ANumLit(n, apply subs t), env
  | AStringLit(s, t) -> AStringLit(s, apply subs t), env
  | AVal(s, t) -> AVal(s, apply subs t), env
  | ABinop(e1, op, e2, t) ->
      ABinop(fst (apply_expr subs e1 env), op, fst (apply_expr subs e2 env), apply subs
          t), env
  | AUnop(op, e, t) -> AUnop(op, fst (apply_expr subs e env), apply subs t), env
  | AAssign(id, e, t) ->
    let apply_expr_ret = apply_expr subs e env in
    let ret_aexpr = AAssign(id, fst apply_expr_ret, apply subs t) in
    let _ = if type_of ret_aexpr = Void then
      raise (failwith "Cannot assign to a void function call") in
    let ret_env = assign_add_map (snd apply_expr_ret) ret_aexpr in
    ret_aexpr, ret_env
  (* OBJECT OPS *)
  | AObjectAssign(o, f, e, t) ->
    let apply_expr_ret = apply_expr subs e env in
    let ret_env1 = match o with
      | AVal(_, Object(_)) -> assign_add_map (snd apply_expr_ret) o
      | _ -> snd apply_expr_ret
    in let ret_aexpr = AObjectAssign(o, f, fst apply_expr_ret, apply subs t) in
    let ret_env2 = assign_add_map ret_env1 ret_aexpr in
    ret_aexpr, ret_env2
  | AObjectField(o, f, t) -> AObjectField(o, f, apply subs t), env
```

```
| ACall(id, args, t) -> ACall(id, args, apply subs t), env
| AListDecl(el, t) -> AListDecl(el, apply subs t), env
| ADictDecl(kl, vl, t) -> ADictDecl(kl, vl, apply subs t), env
(* LIST OPS *)
| AListInsert(e1, e2, e3, t) ->
  let app = apply subs t in
  (* If the list is a variable, update the variable type after the insert *)
  let up_env = match e1 with
    | AVal(s, vt) ->
      let res = base_type_of vt 0 in
      if fst res = Undetermined then (set_iter_type s (List(app)) env) else env
    | AListGet(_, _, vt) ->
      let res = base_type_of vt 0 in
      if fst res = Undetermined then (handle_emb_list e1 (List(app)) env) else env
    | ADictFind(_, _, vt) ->
      let kt, vt = match vt with
        | Dict(k, v) -> k, v
        | _ -> raise (failwith "Expected map")
      in let res = base_type_of vt 0 in
      if fst res = Undetermined then (handle_emb_list e1 (Dict(kt, app)) env) else
        env
    | AObjectField(_, _, t) ->
      let res = base_type_of t 0 in
      if fst res = Undetermined then (handle_emb_list e1 (List(app)) env) else env
    | _ -> env
  in AListInsert(e1, e2, e3, app), up_env
| AListPush(e1, e2, t) ->
  let app = apply subs t in
  (* If the list is a variable, update the variable type after the push *)
  let up_env = match e1 with
    | AVal(s, vt) ->
      let res = base_type_of vt 0 in
      if fst res = Undetermined then (set_iter_type s (List(app)) env) else env
    | AListGet(_, _, vt) ->
      let res = base_type_of vt 0 in
      if fst res = Undetermined then (handle_emb_list e1 (List(app)) env) else env
    | ADictFind(e, _, _) ->
      let vt = match type_of e with
        | Dict(_, v) -> v
        | _ -> raise (failwith "Expected map")
      in let res = base_type_of vt 0 in
      if fst res = Undetermined then (handle_emb_list e1 (List(app)) env) else env
    | AObjectField(_, _, t) ->
      let res = base_type_of t 0 in
      if fst res = Undetermined then (handle_emb_list e1 (List(app)) env) else env
    | _ -> env
  in AListPush(e1, e2, apply subs t), up_env
```

```
  | AListRemove(e1, e2, t) -> AListRemove(e1, e2, apply subs t), env
  | AListGet(e1, e2, t) -> AListGet(e1, e2, apply subs t), env
  | AListPop(e, t) -> AListPop(e, apply subs t), env
  | AListDequeue(e, t) -> AListDequeue(e, apply subs t), env
  | AListLength(e, t) -> AListLength(e, apply subs t), env
  | AListSet(e1, e2, e3, t) -> AListSet(e1, e2, e3, apply subs t), env
  (* DICT OPS *)
  | ADictMem(e1, e2, t) -> ADictMem(e1, e2, apply subs t), env
  | ADictDelete(e1, e2, t) -> ADictDelete(e1, e2, apply subs t), env
  | ADictSize(e, t) -> ADictSize(e, apply subs t), env
  | ADictFind(e1, e2, t) -> ADictFind(e1, e2, apply subs t), env
  | ADictMap(e1, e2, e3, t) ->
    let app = apply subs t
    and t2 = type_of e2 and t3 = type_of e3 in
    let up_env = match e1 with
      | AVal(s, vt) ->
        let k = match vt with
          | Dict(x, _) -> x
          | _ -> raise (failwith "Expected a dict")
        in if k = Undetermined then set_iter_type s (Dict(t2, t3)) env else env
      | AListGet(e1, _, t) ->
        let k = match t with
          | Dict(x, _) -> x
          | _ -> raise (failwith "Expected a dict")
        in if k = Undetermined then (handle_emb_list e1 (List(Dict(t2, t3))) env) else
            env
      | AObjectField(_, s, t) ->
        let k = match t with
          | Dict(x, _)-> x
          | _ -> raise(failwith "Expected a dict")
        in if k = Undetermined then set_iter_type s (Dict(t2, t3)) env else env
      | _ -> env
    in ADictMap(e1, e2, e3, app), up_env
  | _ -> raise (failwith("Invalid aexpr in apply_aexpr"))

(*
handle_emb_list: Changes the type of an undetermined embedded list (ie [[], []]) after
    a push or insert
Params:
  aexpr: The current layer of expression. The total form should be a number of
      AListGet's with an AVal at the center
  t: The data_type thus far of the list , embeds more lists recursively
  env: the local function environment where the type of the list will be altered
Returns:
  env: The updated environment
*)
and handle_emb_list (ae: aexpr) (t: data_type) (env: environment): environment =
```

```
  match ae with
    | AVal(s, _) ->
      let temp = NameMap.remove s env in
      NameMap.add s t temp
    | AListGet(e1, _, _) ->
      handle_emb_list e1 (List(t)) env
    | ADictFind(e1, _, _) ->
      let res = type_of e1 in
      let kt = match res with
        | Dict(k, _) -> k
        | _ -> raise (failwith "Expected dict")
      in handle_emb_list e1 (Dict(kt, t)) env
    | AObjectField(_, s, _) ->
      let _ = Hashtbl.remove field_hash s in
      let _ = Hashtbl.add field_hash s t
      and temp = NameMap.remove s env in
      NameMap.add s t temp
    | _ -> raise (failwith "Expected list or map")

(*
 set_iter_type : general helper function to remove then re-add a type for a variable
Params:
  s: The variable name
  t: The data_type to be inserted
  env: the local function environment where the type of the list will be altered
Returns:
  env: The updated environment after s has been set to type t
*)
and set_iter_type (s: string) (t: data_type) (env: environment): environment =
  let _ = match Hashtbl.mem field_hash s with
    | true ->
      let _ = Hashtbl.remove field_hash s in
      let _ = Hashtbl.add field_hash s t in ()
    | _ -> ()
  in let temp = NameMap.remove s env in
  NameMap.add s t temp

(*
 infer : Infer the annotated expression form of an expression by calling HM type
     inference
Params:
  env: the local function environment, used in case e is an assignment or contains an
       Id
  e: The expression to perform type inference on
Returns:
  aexpr: annotated version of the expression
  env: local function environment
```

```
*)
and infer (env: environment) (e: expr) : (aexpr * environment) =
  let annotated_expr = annotate_expr e env in
  let constraints = collect_expr annotated_expr in
  let subs = unify constraints in
  (* reset the type counter after completing inference *)
  type_variable := (Char.code 'a');
  let ret = apply_expr subs annotated_expr env in
  if logging then
   let _ = print_endline (string_of_aexpr (fst ret)) in ret else ret

(*
infer: Infer the annotated expression form of an statement
Params:
  env: the local function environment, used in case e is an assignment or contains an
      Id
  stmt: The statement to be annotated
Returns:
  astmt: annotated version of the statement
  env: local function environment, updated to include possible changes due to
      assignments
*)
and infer_stmt (env: environment) (stmt: stmt) (func_name: string): (astmt *
    environment) =
  match stmt with
  (* Expr -> infer the expression *)
  | Expr(e) -> let ret = infer env e in AExpr(fst ret), snd ret
  (* Return -> infer the expression being returned *)
  | Print(e) -> let ret = infer env e in APrint(fst ret), snd ret
  | Return(e) ->
    (match e with
      | Ast.Noexpr -> AReturn(Sast.ANoexpr(Void)), env
      | _ ->
        let ret = infer env e in
        let d = type_of (fst ret) in
        if (Hashtbl.mem func_type_hash func_name) then
          (* Functions must return same type *)
          let _ = log ("[infer_stmt] looking up " ^ func_name ^ " in func_type_hash")
              in
          if (Hashtbl.find func_type_hash func_name <> d) then
            raise (failwith ("[infer_stmt] mismatched types of function " ^ func_name))
          else
            AReturn(fst ret), snd ret
        (* If function has not been inferred before, add the return type of the function
            *)
        else
          let _ = Hashtbl.add func_type_hash func_name d in
```

```
        AReturn(fst ret), snd ret
  )
| Break -> ABreak, env
| Continue -> AContinue, env
| ObjectInit( obj_list ) ->
  let new_env = List.fold_left (fun acc obj_name -> (NameMap.add obj_name (
      Object(-1)) acc)) env obj_list in
  AObjectInit(List.map (fun x -> (x, Object(-1))) obj_list), new_env
| While(e, s) ->
  (* Infer the condition *)
  let inf = fst (infer env e) in
  let _ = match type_of inf with
    | Bool -> "ok"
    | _ -> raise (failwith ("expected boolean in if statement"))
  (* unwrap the statement list from the block *)
  and bl = match s with
    | Block(sl) -> sl
    | _ -> raise (failwith ("expected block"))
  (* Call inference on the block *)
  in let abl = fst (map_infer_stmt bl env [] func_name) in
  (* let _ = print_endline (string_of_data_type (type_of inf)) in *)
  AWhile(inf, ABlock(abl)), env
| If(e, s1, s2) ->
  (* infer the condition *)
  let inf = fst (infer env e) in
  let _ = match type_of inf with
    | Bool -> "ok"
    | _ -> raise (failwith ("expected boolean in if statement"))
  (* unwrap the statement lists from the blocks *)
  and bl1 = match s1 with
    | Block(sl) -> sl
    | _ -> raise (failwith ("expected block"))
  and bl2 = match s2 with
    | Block(sl) -> sl
    | _ -> raise (failwith ("expected block"))
  (* infer the blocks *)
  in let abl1 = fst (map_infer_stmt bl1 env [] func_name)
  in let abl2 = fst (map_infer_stmt bl2 env [] func_name)
  in AIf(inf, ABlock(abl1), ABlock(abl2)), env
| ForRange(e1, e2, e3, s) ->
  (* infer the from-to expression pair *)
  let inf1_ae, inf1_env = infer env e1 in
  let inf2_ae, inf2_env = infer inf1_env e2 in
  let inf3_ae, _ = infer inf2_env e3 in
  let _ = match inf1_ae with
    | AAssign(_) -> (match type_of inf1_ae with
      | Num -> "ok"
```

```
        | _ -> raise (failwith ("assignment in for loop must be a number")))
      | _ -> raise (failwith ("expected assignment in for statement"))
    and _ = (match type_of inf2_ae with
      | Num -> "ok"
      | _ -> raise (failwith ("end in for statement must be a number")))
    and _ = (match type_of inf3_ae with
      | Num -> "ok"
      | _ -> raise (failwith ("by in for statement must be a number")))
    (* unwrap the statement list from the block *)
    and bl = match s with
      | Block(sl) -> sl
      | _ -> raise (failwith ("expected block"))
    (* infer the blocks *)
    in let abl = fst (map_infer_stmt bl inf1_env [] func_name)
    in AForRange(inf1_ae, inf2_ae, inf3_ae, ABlock(abl)), env
  | ForIn(e1, e2, s) ->
    (* infer the for-in expression pair *)
    let vname = (match e1 with
      | Id(s) -> s
      | _ -> raise (failwith "First term in For In must be a variable"))
    and inf2 = fst (infer env e2)
    in let t = (match type_of inf2 with
      | Dict(t, _) | List(t) -> t
      | _ -> raise (failwith "Second term in For In must be an iterable"))
    in let new_env = NameMap.add vname t env
    (* unwrap the statement list from the block *)
    in let bl = match s with
      | Block(sl) -> sl
      | _ -> raise (failwith ("expected block"))
    (* infer the block *)
    in let abl = fst (map_infer_stmt bl new_env [] func_name)
    in AForIn(AVal(vname, t), inf2, ABlock(abl)), new_env
  | _ -> raise (failwith ("unexpected statement"))

(*
map_infer_stmt: for a given statment, infer the type of it while maintaining the scope's
    name map
params:
  stmt_list: list of statements from a function's body
  env: Name map with keys of IDs and values of the IDs data type
  astmt_list: final return list of annotated statements
  f_name: the name of the function the statement is in - used to handle returns and
    adding to the function type map
returns:
  astmt_list: annotated version of the stmt list passed in
  env: the name map with keys of IDs and values of the IDs data type
    -only used for the recursion
```

```
*)
and map_infer_stmt (stmt_list: stmt list) (env: environment) (astmt_list: astmt list) (
    f_name: string): (astmt list * environment) =
  (* base case - all statements have been inferred *)
  match stmt_list with
      [] -> astmt_list, env
    | head :: tail ->
      (* infer the first statement, concat the annotated version to the accumulator *)
      let inf = infer_stmt env head f_name in
      let new_astmt_list = astmt_list @ [fst inf]
      (* update the environment *)
      and new_env = snd inf in
      map_infer_stmt tail new_env new_astmt_list f_name

(*
add_tuple_to_env: adds an id tuple to a local function environment
params:
  env: the data type map to add to
  tuple: the (data type, string) tuple to be added
returns:
  env: the environment after it has been updated
*)
and add_tuple_to_env (env: environment) (tuple: (data_type * string)) =
  NameMap.add (snd tuple) (fst tuple) env

(*
get_param_types: gets the parameter types of a function
params:
  formals: list of Ids that are the parameters
  types: In the same order, the list of the types of those paraeters
  env: The initial starting environment (should be NameMap.empty but passed just in
      case)
returns:
  final_env: the environment after the tuples have been added
  tuple_list: the (data_type * string) formals list to be input in aformals
*)
and get_param_types (formals: string list) (types: data_type list) (env: environment)
    =
  let tuple_list = List.map2 (fun a b -> (a, b)) types formals in
  let final_env = List.fold_left add_tuple_to_env env tuple_list in
  final_env, tuple_list

(*
infer_types_function: Calls inference on a function, given its declaration
params:
  func: The function declaration, as taken from the AST's list of fdecls
  param_types: The types of the parameters, as inferred from the first time the function
```

```
        is called
returns:
  abody: The annotated list of statements that constitute the annotated function's body
*)
and infer_types_function (func: func_decl) (param_types: data_type list) =
  (* Load the function parameters into the function environment, get aformals in the
      process *)
  let env_and_aformals = get_param_types func.formals param_types NameMap.empty in
  let func_env = fst env_and_aformals
  and aforms = snd env_and_aformals in
  (* Call inference on the function body *)
  let ret = map_infer_stmt func.body func_env [] func.fname in
  (* Add the function body and function formals to the global body and formal tables
      respectively *)
  let _ = Hashtbl.add func_body_hash func.fname (fst ret)
  and _ = Hashtbl.add func_formals_hash func.fname aforms
  and _ = if not (Hashtbl.mem func_type_hash func.fname)
    then Hashtbl.add func_type_hash func.fname Void in
  let print_env k v =
    log (k ^ ": " ^ string_of_data_type v)
    in let _ = NameMap.iter print_env (snd ret) in
    let _ = Hashtbl.add func_env_hash func.fname (snd ret) in
  fst ret


and handle_undet_expr (e: Sast.aexpr) (name: string) =
  match e with
    | ABinop(ae1, op, ae2, typ) ->
      let hae1 = handle_undet_expr ae1 name and hae2 = handle_undet_expr ae2 name
          in
      let t1 = type_of hae1 and t2 = type_of hae2 in
      let _ = match op with
        | Add | Minus | Times | Divide ->
          if not (t1 = Num && t2 = Num) then
            raise (failwith "Must be a Num")
          else ()
        | Greater | Less | Geq | Leq | Eq | Neq ->
          if not (t1 = t2) then
            raise (failwith "Must be a same type")
          else ()
        | And | Or ->
          if not (t1 = Bool && t2 = Bool) then
            raise (failwith "Must be a Bool")
          else ()
        | Concat ->
          if not (t1 = String && t2 = String) then
            raise (failwith "Must be a same type")
```

```
      else ()
    | Combine -> ()
  in ABinop(hae1, op, hae2, typ)
| AUnop(op, ae, typ) ->
  let hae = handle_undet_expr ae name in
  let t = type_of hae in
  let _ = match op with
    | Neg ->
      if not (t = Num) then
        raise (failwith "Must be a Num")
      else ()
    | Not ->
      if not (t = Bool) then
        raise (failwith "Must be a Bool")
      else ()
  in AUnop(op, hae, typ)
| AAssign(id, ae, typ) ->
  if fst (base_type_of typ 0) = Undetermined then
    let new_ae = handle_undet_expr ae name in
    let new_typ = type_of new_ae in
    let new_typ = if fst (base_type_of new_typ 0) = Undetermined then
      let _ = log ("[undet] " ^ string_of_aexpr e) in
      let _ = log ("[undet] " ^ string_of_data_type (NameMap.find id (Hashtbl.find
          func_env_hash name))) in
      NameMap.find id (Hashtbl.find func_env_hash name)
      else new_typ
    in AAssign(id, new_ae, new_typ)
  else
    AAssign(id, handle_undet_expr ae name, typ)
| ACall(id, aexpr_list, typ) ->
  let new_aexpr_list = List.map (fun x -> handle_undet_expr x name) aexpr_list in
  let new_typ = match typ with
    | Undetermined -> Hashtbl.find func_type_hash id
    | _ -> typ
  in ACall(id, new_aexpr_list, new_typ)
| AListDecl(aexpr_list, typ) ->
  let new_aexpr_list = List.map (fun x -> handle_undet_expr x name) aexpr_list in
  AListDecl(new_aexpr_list, typ)
| AListInsert(ae1, ae2, ae3, typ) ->
  AListInsert(handle_undet_expr ae1 name, handle_undet_expr ae2 name,
      handle_undet_expr ae3 name, typ)
| AListPush(ae1, ae2, typ) ->
  AListPush(handle_undet_expr ae1 name, handle_undet_expr ae2 name, typ)
| AListRemove(ae1, ae2, typ) ->
  AListRemove(handle_undet_expr ae1 name, handle_undet_expr ae2 name, typ)
| AListPop(ae, typ) ->
  AListPop(handle_undet_expr ae name, typ)
```

```
    | AListDequeue(ae, typ) ->
      AListDequeue(handle_undet_expr ae name, typ)
    | AListLength(ae, typ) ->
      AListLength(handle_undet_expr ae name, typ)
    | AListGet(ae1, ae2, typ) ->
      AListGet(handle_undet_expr ae1 name, handle_undet_expr ae2 name, typ)
    | AListSet(ae1, ae2, ae3, typ) ->
      AListSet(handle_undet_expr ae1 name, handle_undet_expr ae2 name,
          handle_undet_expr ae3 name, typ)
    | ADictDecl(ael1, ael2, typ) ->
      let map_handle = (fun x -> handle_undet_expr x name) in
      ADictDecl(List.map map_handle ael1, List.map map_handle ael2, typ)
    | ADictMem(ae1, ae2, typ) ->
      ADictMem(handle_undet_expr ae1 name, handle_undet_expr ae2 name, typ)
    | ADictDelete(ae1, ae2, typ) ->
      ADictDelete(handle_undet_expr ae1 name, handle_undet_expr ae2 name, typ)
    | ADictFind(ae1, ae2, typ) ->
      ADictFind(handle_undet_expr ae1 name, handle_undet_expr ae2 name, typ)
    | ADictSize(ae, typ) ->
      ADictSize(handle_undet_expr ae name, typ)
    | ADictMap(ae1, ae2, ae3, typ) ->
      ADictMap(handle_undet_expr ae1 name, handle_undet_expr ae2 name,
          handle_undet_expr ae3 name, typ)
    | AObjectField(ae, s, typ) ->
      AObjectField(handle_undet_expr ae name, s, typ)
    | AObjectAssign(ae1, s, ae2, typ) ->
      AObjectAssign(handle_undet_expr ae1 name, s, handle_undet_expr ae2 name, typ)
    | _ -> e



and handle_undet_stmt (s: Sast.astmt) (name: string) =
  match s with
    | AExpr(aexpr) -> AExpr(handle_undet_expr aexpr name)
    | ABlock(astmt_list) -> ABlock(List.map (fun x-> handle_undet_stmt x name)
        astmt_list)
    | AReturn(aexpr) -> AReturn(handle_undet_expr aexpr name)
    | AWhile(aexpr, astmt) -> AWhile(handle_undet_expr aexpr name,
        handle_undet_stmt astmt name)
    | AForIn(aexpr1, aexpr2, astmt) ->
      AForIn(handle_undet_expr aexpr1 name, handle_undet_expr aexpr2 name,
          handle_undet_stmt astmt name)
    | AForRange(aexpr1, aexpr2, aexpr3, astmt) ->
      AForRange(handle_undet_expr aexpr1 name, handle_undet_expr aexpr2 name,
              handle_undet_expr aexpr3 name, handle_undet_stmt astmt name)
    | AIf(aexpr, astmt1, astmt2) ->
      AIf(handle_undet_expr aexpr name, handle_undet_stmt astmt1 name,
```

```
            handle_undet_stmt astmt2 name)
    | APrint(expr) -> APrint(handle_undet_expr expr name)
    | _ -> s


and handle_undet_func (afdecl: Sast.afunc_decl) =
  let new_afunc_decl = {
    afname = afdecl.afname;
    aformals = afdecl.aformals;
    abody = List.map (fun x -> handle_undet_stmt x afdecl.afname) afdecl.abody;
    return = afdecl.return;
  } in new_afunc_decl

and handle_undet (sast: Sast.aprogram) =
  List.map handle_undet_func sast









(* ==================== OBJECTS ==================== *)

(* Maps each object to its list of fields: <obj, [field1, field2, field3]> *)
and obj_to_fields_hash = Hashtbl.create 42

(* Maps each object to its type id: <obj, id> *)
and obj_to_id_hash = Hashtbl.create 42

(* Maps each type id to its list of fields <id, [field1, field2, field3]> *)
and id_to_fields_hash = Hashtbl.create 42

(* Maps each type id to its list of fields with types <id, [(field1, Num), (field2,
    Bool), (field3, String)]> *)
and id_to_fields_typified_hash = Hashtbl.create 42

(* Maps function names to the list of object names that are equivalent to the function
    return *)
and fname_to_obj_equality_hash = Hashtbl.create 42

(* Maps function names to whether or not they have been visisted by obj_collect *)
and fname_to_visited_hash = Hashtbl.create 42

and obj_inserted = ref false

and obj_seen = ref false
```

```
and curr_id = ref 0

and gen_new_obj_id () : int =
  let n1 = !curr_id in
  let _ = incr curr_id in n1

(*
[ENTRY POINT]

infer_objs: Entry point into object inference algorithm
params:
  sast: the program to run object inference on
returns:
  a tuple of (afdecls, obj_fields_hash):
    afdecls: the augmented sast with object ids
    id_to_fields_hash : the map of ids to lists of fields
*)
and infer_objs (sast: Sast.aprogram) =
  (* Tracks objects that are of the same type: <obj1, obj2> *)
  let obj_equality_sets = obj_collect (Hashtbl.find afunc_decl_hash "main") [] in

  let _ = log "[infer_objs] obj_equality_sets" in
  let _ = print_set_list obj_equality_sets in
  let _ = print_obj_to_fields_hash () in

  let _ = obj_unify obj_equality_sets in

  let _ = print_obj_to_id_hash() in
  let _ = print_id_to_fields_hash () in

  let afdecls = obj_apply sast in
  let _ = typify_fields_hash () in
  let _ = print_id_to_fields_typified_hash () in
  (afdecls, id_to_fields_typified_hash )




(* ==================== OBJECT UTILS
    ==================== *)

and remove_dups lst = match lst with
  | [] -> []
  | h::t -> h::(remove_dups (List.filter (fun x -> x<>h) t))
```

```
and typify_fields_hash () =
  Hashtbl.iter (fun id fields ->
    let new_fields = List.map (fun field -> (field, Hashtbl.find field_hash field))
        fields in
    let new_unique_fields = remove_dups new_fields in
    Hashtbl.add id_to_fields_typified_hash id new_unique_fields
  ) id_to_fields_hash

and add_field_to_obj (field: string) (obj: string) =
  let _ = log ("Adding " ^ field ^ " to object called " ^ obj) in
  let field_list =
    if Hashtbl.mem obj_to_fields_hash obj
    then Hashtbl.find obj_to_fields_hash obj
    else []
  in
  let new_field_list =
    if List.mem field field_list
    then field_list
  else field_list @ [field]
  in
  Hashtbl.add obj_to_fields_hash obj new_field_list

(*
add_new_equality: Inserts an equality of two objects into the equality set list
*)
and add_new_equality (eq: ObjectTypeSet.t list) (obj1: string) (obj2: string) : (
    ObjectTypeSet.t list) =
  let _ = log ("[add_new_equality] adding a new equality between " ^ obj1 ^ " and " ^
      obj2) in
  let _ = log ("[add_new_equality] current set list looks like: ") in
  let _ = print_set_list eq in
  obj_inserted := false;
  let ret = List.map (fun (obj_set: ObjectTypeSet.t) ->
    let _ = log ("[add_new_equality] considering adding " ^ obj1 ^ "," ^ obj2 ^ " into
        the following set:") in
    let _ = print_set obj_set in
    let new_obj_set =
      if (ObjectTypeSet.mem obj1 obj_set && not (ObjectTypeSet.mem obj2 obj_set))
      then (obj_inserted := true; let _ = log ("[add_new_equality] adding " ^ obj2 ^ "
          into the above set") in ObjectTypeSet.add obj2 obj_set)
      else obj_set
    in
    let new_obj_set =
      if (ObjectTypeSet.mem obj2 obj_set && not (ObjectTypeSet.mem obj1 obj_set))
      then (obj_inserted := true; let _ = log ("[add_new_equality] adding " ^ obj1 ^ "
          into the above set") in ObjectTypeSet.add obj1 new_obj_set)
      else new_obj_set
```

```
    in
    new_obj_set
  ) eq in
  if !obj_inserted = false (*  if objected_inserted is equal to false *)
  then
    let _ = log ("[add_new_equality] adding a completely new set") in
    let new_set = ObjectTypeSet.add obj1 ObjectTypeSet.empty in
    let new_set = ObjectTypeSet.add obj2 new_set in
    let _ = print_set_list (ret @ [new_set]) in
    ret @ [new_set]
  else ret

and add_lone_equality (eq: ObjectTypeSet.t list) (obj: string) : (ObjectTypeSet.t list)
    =
  obj_seen := false;
  let _ = List.iter (fun obj_set ->
    if ObjectTypeSet.mem obj obj_set
    then (obj_seen := true; ())
    else ()
  ) eq
  in
  if !obj_seen = false
  then let new_set = ObjectTypeSet.add obj ObjectTypeSet.empty in
  eq @ [new_set]
  else eq

(*
get_arg_index: Returns the index of an argument in a call to a function
*)
and get_arg_index (args: aexpr list) (arg: string) : (int) =
  let rec get_idx key ael idx =
    match ael with
      | [] -> let _ = log "[get_arg_index] Base case" in raise (failwith "Arg not found
          in function")
      | head :: tail ->
        match head with
          | AVal(str, _) ->
            let _ = log ("[get_arg_index] Arg of name " ^ str ^ " found in the ACall,
                matching against " ^ key) in
            if str = key
            then idx
            else get_idx key tail (idx + 1)
          | _ -> raise (failwith("Argument not found in formals list"))
  in get_idx arg args 0

(*
get_obj_name_from_aexpr: Returns the object name from an aexpr or "" if it is not an
```

```
    object
*)
and get_obj_name_from_aexpr (ae: aexpr) : string =
  match ae with
    | AVal(name, typ) ->
      (match typ with
        | Object(_) -> name
        | _ -> "")
      )
    | AObjectField(_, field , typ) ->
      (match typ with
        | Object(_) -> field
        | _ -> "")
      )
    | _ -> ""

and get_list_type_from_aexpr (ae: aexpr) : int =
  let list_name = get_list_name_from_aexpr ae in
  let encoded_list_name = encode_list_name list_name in
  if Hashtbl.mem obj_to_id_hash encoded_list_name
  then Hashtbl.find obj_to_id_hash encoded_list_name
  else -1 (* Not a list of objects *)

and get_list_type_from_listdecl (ae: aexpr) : data_type =
  match ae with
    | AListDecl( ae_list , typ) ->
      if List.length ae_list = 0
      then typ
      else List(type_of_aexpr (List.hd ae_list ))
    | _ -> raise (failwith("Attempting to get List type from not a ListDecl"))

and get_list_name_from_aexpr (annotated_expr: aexpr) : string =
  match annotated_expr with
    | AVal(name, typ) ->
      (match typ with
        | List(_) -> name
        | _ -> "")
      )
    | AListInsert(ae, _, _, _) -> get_list_name_from_aexpr ae
    | AListPush(ae, _, _) -> get_list_name_from_aexpr ae
    | AListRemove(ae, _, _) -> get_list_name_from_aexpr ae
    | AListPop(ae, _) -> get_list_name_from_aexpr ae
    | AListDequeue(ae, _) -> get_list_name_from_aexpr ae
    | AListLength(ae, _) -> get_list_name_from_aexpr ae
    | AListGet(ae, _, _) -> get_list_name_from_aexpr ae
    | AListSet(ae, _, _, _) -> get_list_name_from_aexpr ae
    | _ -> ""
```

```
and encode_list_name (list_name: string) : string =
  "." ^ list_name

and inject_list_decl_type (ae: aexpr) (new_typ: data_type) : aexpr =
  match ae with
    | AListDecl(contents, _) -> AListDecl(contents, new_typ)
    | _ -> raise (failwith("Injecting: Not an AListDecl"))

and encode_dict_name (dict_name: string) : string =
  "," ^ dict_name

and get_dict_name_from_aexpr (annotated_expr: aexpr) : string =
  match annotated_expr with
    | AVal(name, typ) ->
      (match typ with
        | Dict(_, _) -> name
        | _ -> ""
      )
    | ADictMem(ae, _, _) -> get_dict_name_from_aexpr ae
    | ADictDelete(ae, _, _) -> get_dict_name_from_aexpr ae
    | ADictSize(ae, _) -> get_dict_name_from_aexpr ae
    | ADictFind(ae, _, _) -> get_dict_name_from_aexpr ae
    | ADictMap(ae, _, _, _) -> get_dict_name_from_aexpr ae
    | _ -> ""

and get_dict_val_type_from_aexpr (ae: aexpr) : int =
  let dict_name = get_dict_name_from_aexpr ae in
  let _ = log ("aexpr is " ^ string_of_aexpr ae ^ " and dict_name is " ^ dict_name) in
  if Hashtbl.mem obj_to_id_hash dict_name
  then Hashtbl.find obj_to_id_hash dict_name
  else -1 (* Not a list of objects *)

(* Binds an object name to a function return type, adds and equality if applicable, and
    returns the set list *)
(* [Magic] *)
and insert_obj_into_fname_to_obj_equality_hash (acc: ObjectTypeSet.t list) (fname:
    string) (obj: string) : (ObjectTypeSet.t list) =
  let curr_list =
    if Hashtbl.mem fname_to_obj_equality_hash fname
    then Hashtbl.find fname_to_obj_equality_hash fname
    else []
  in
  let acc =
    if List.length curr_list >= 1
    then add_new_equality acc obj (List.hd curr_list)
    else acc
```

```
  in
  let new_list =
    if List.mem obj curr_list
    then curr_list
    else curr_list @ [obj]
  in
  let _ = Hashtbl.add fname_to_obj_equality_hash fname new_list in
  acc




(* ====================== OBJECT COLLECT
    ====================== *)

and obj_collect_fields_aexpr (ae: aexpr) =
  match ae with
    (* Base Cases *)
    | AObjectField(ae, field, _) ->
      add_field_to_obj field (get_obj_name_from_aexpr ae)
    | AObjectAssign(ae, field, iae, _) ->
      let _ = obj_collect_fields_aexpr iae in
      add_field_to_obj field (get_obj_name_from_aexpr ae)
    (* Recursive Cases *)
    | ABinop(ae1, _, ae2, _) ->
      let _ = obj_collect_fields_aexpr ae1 in
      let _ = obj_collect_fields_aexpr ae2 in
      ()
    | AUnop(_, ae, _) ->
      obj_collect_fields_aexpr ae
    | AAssign(_, ae, _) ->
      obj_collect_fields_aexpr ae
    | ACall(_, ae_list, _) ->
      List.iter (fun ae -> obj_collect_fields_aexpr ae) ae_list
    | _ -> ()

(*
 obj_collect_fields_astmt : If the given astmt contains an AObjectField, record it in
    obj_to_fields_hash
params:
  annotated_stmt: an astmt to collect fields from
returns:
  unit
*)
and obj_collect_fields_astmt (annotated_stmt: astmt) =
  match annotated_stmt with
```

```
    | ABlock(astmt_list) -> List.iter  obj_collect_fields_astmt  astmt_list
    | AExpr(ae) -> obj_collect_fields_aexpr ae
    | AReturn(ae) -> obj_collect_fields_aexpr ae
    | AWhile(ae, astmt) ->
      let _ = obj_collect_fields_aexpr  ae in
      let _ = obj_collect_fields_astmt  astmt in
      ()
    | AForIn(ae1, ae2, astmt) ->
      let _ = obj_collect_fields_aexpr  ae1 in
      let _ = obj_collect_fields_aexpr  ae2 in
      let _ = obj_collect_fields_astmt  astmt in
      ()
    | AForRange(ae1, ae2, ae3, astmt) ->
      let _ = obj_collect_fields_aexpr  ae1 in
      let _ = obj_collect_fields_aexpr  ae2 in
      let _ = obj_collect_fields_aexpr  ae3 in
      let _ = obj_collect_fields_astmt  astmt in
      ()
    | AIf(ae, astmt1, astmt2) ->
      let _ = obj_collect_fields_aexpr  ae in
      let _ = obj_collect_fields_astmt  astmt1 in
      let _ = obj_collect_fields_astmt  astmt2 in
      ()
    | APrint(ae) -> obj_collect_fields_aexpr  ae
    | _ -> ()

(*
 obj_collect_equalities_param : Collects the param equality for one param in a function
     call
*)
and obj_collect_equalities_param (fname: string) (args: aexpr list ) (arg: aexpr) (acc:
    ObjectTypeSet.t list) : (ObjectTypeSet.t list) =
  match arg with
    | AVal(arg_name, typ) ->
      (* arg_name is the name of the object that is passed into the function call *)
      (* we find the index of arg_name in the function call , and get the corresponding
          param name in the afdecl *)
      (* then we add a new equality for them *)
      (match typ with
        | Object(_) ->
          let afdecl = Hashtbl.find afunc_decl_hash fname in
          let index = get_arg_index args arg_name in
          let param = List.nth afdecl.aformals index in
          let param_name = snd param in
          let acc = add_new_equality acc param_name arg_name in
          acc
        | _ -> acc
```

```
      )
    | _ -> acc

(*
 obj_collect_equalities_params : Collects all param equalities for one function call
*)
and obj_collect_equalities_params (fname: string) (args: aexpr list ) (acc:
    ObjectTypeSet.t list) : (ObjectTypeSet.t list) =
  List. fold_left (fun acc arg -> obj_collect_equalities_param fname args arg acc) acc
      args

and get_obj_or_list_or_map_name_from_aexpr (ae: aexpr) : string =
  let obj_name = get_obj_name_from_aexpr ae in
  if obj_name = ""
  then (
    let list_name = get_list_name_from_aexpr ae in
    if list_name = ""
    then (
      let dict_name = get_dict_name_from_aexpr ae in
      if dict_name = ""
      then ""
    else encode_dict_name dict_name
    )
    else encode_list_name list_name
  )
  else obj_name

(*
 obj_collect_equalities_binop : Collects the equality from 'obj1 == obj2'
*)
and obj_collect_equalities_binop (acc: ObjectTypeSet.t list) (lhs: aexpr) (op: binop) (
    rhs: aexpr) : (ObjectTypeSet.t list) =
  match op with
    | Eq ->
      (match type_of rhs with
        | Object(_) ->
          let lname = get_obj_or_list_or_map_name_from_aexpr lhs in
          let rname = get_obj_or_list_or_map_name_from_aexpr rhs in
          if (lname <> "") && (rname <> "")
          then add_new_equality acc lname rname
          else acc
        | _ -> acc
      )
    | _ -> acc

and obj_collect_equalities_list (acc: ObjectTypeSet.t list) (lst: aexpr) (obj: aexpr) =
  let list_name = get_list_name_from_aexpr lst in
```

```
  if list_name = ""
  then acc
  else (
    let obj_name = get_obj_name_from_aexpr obj in
    if obj_name = ""
    then acc
    else (
      let encoded_list_name = encode_list_name list_name in
      add_new_equality acc encoded_list_name obj_name
    )
  )

and obj_collect_equalities_dict (acc: ObjectTypeSet.t list) (dict: aexpr) (obj: aexpr)
    =
  let dict_name = get_dict_name_from_aexpr dict in
  if dict_name = ""
  then acc
  else (
    let obj_name = get_obj_name_from_aexpr obj in
    if obj_name = ""
    then acc
    else (
      let encoded_dict_name = encode_dict_name dict_name in
      add_new_equality acc encoded_dict_name obj_name
    )
  )

and obj_collect_equalities_aexpr (acc: ObjectTypeSet.t list) (aexpr: aexpr) : (
    ObjectTypeSet.t list) =
  match aexpr with
    (* Case 1; obj1 = obj2 -> (obj1, obj2) *)
    | AAssign(lhs_obj, aexpr, _) ->
      let acc = obj_collect_equalities_aexpr acc aexpr in
      (match aexpr with
        | AVal(rhs_obj, aval_data_type) ->
          (match aval_data_type with
            | Object(_) ->
              let acc = add_new_equality acc lhs_obj rhs_obj in
              acc
            | _ -> acc
          )
        | ACall(fname, _, _) ->
          insert_obj_into_fname_to_obj_equality_hash acc fname lhs_obj
        | AListDecl(ae_list, typ) ->
          (match typ with
            | List(Object(_)) ->
              List.fold_left (fun acc ae -> add_new_equality acc (encode_list_name
```

```
            lhs_obj) (get_obj_name_from_aexpr ae)) acc ae_list
    | _ -> acc
  )
| AListRemove(lst, _, typ) ->
  (match typ with
    | Object(_) ->
      let acc = add_new_equality acc lhs_obj (encode_list_name (
          get_list_name_from_aexpr lst)) in
      acc
    | _ -> acc
  )
| AListPop(lst, typ) ->
  (match typ with
    | Object(_) ->
      let acc = add_new_equality acc lhs_obj (encode_list_name (
          get_list_name_from_aexpr lst)) in
      acc
    | _ -> acc
  )
| AListGet(lst, _, typ) ->
  (match typ with
    | Object(_) ->
      let acc = add_new_equality acc lhs_obj (encode_list_name (
          get_list_name_from_aexpr lst)) in
      acc
    | _ -> acc
  )
| ADictDelete(dict, _, typ) ->
  (match typ with
    | Object(_) ->
      let acc = add_new_equality acc lhs_obj (encode_dict_name (
          get_dict_name_from_aexpr dict)) in
      acc
    | _ -> acc
  )
| ADictFind(dict, _, typ) ->
  (match typ with
    | Object(_) ->
      let acc = add_new_equality acc lhs_obj (encode_dict_name (
          get_dict_name_from_aexpr dict)) in
      acc
    | _ -> acc
  )
| AObjectField(_, field , typ) ->
  (match typ with
    | Object(_) ->
      let acc = add_new_equality acc lhs_obj field in
```

```
            acc
          | _ -> acc
        )
      | _ -> acc
    )
  (* Case 2; foo(obj1), def foo(a) -> (obj1, a) *)
  | ACall(func_name, args, _) ->
    let acc = obj_collect_equalities_params func_name args acc in
    if Hashtbl.mem fname_to_visited_hash func_name
    then acc
    else let _ = Hashtbl.add fname_to_visited_hash func_name true in
    (obj_collect (Hashtbl.find afunc_decl_hash func_name)) acc
  (* Case 3; if obj1 == obj2 -> (obj1, obj2) *)
  | ABinop(lhs, op, rhs, _) ->
    let acc = obj_collect_equalities_binop acc lhs op rhs in
    let acc = obj_collect_equalities_aexpr acc lhs in
    let acc = obj_collect_equalities_aexpr acc rhs in
    acc
  (* Case 4; a.obj1 = obj2 -> (obj1, obj2) *)
  | AObjectAssign(_, field, ae2, _) ->
    (match ae2 with
      | AVal(name, aval_typ) ->
        (match aval_typ with
          | Object(_) -> add_new_equality acc field name
          | _ -> acc
        )
      | _ -> acc
    )

  (* List Function Cases *)
  | AListInsert(lst, _, obj, _) ->
      obj_collect_equalities_list acc lst obj
  | AListPush(lst, obj, _) ->
      obj_collect_equalities_list acc lst obj
  | AListSet(lst, _, obj, _) ->
      obj_collect_equalities_list acc lst obj

  (* Dict Function Cases *)
  | ADictMap(dict, _, obj, _) ->
      obj_collect_equalities_dict acc dict obj

  (* Fallthrough *)
  | _ -> let _ = log ("[ obj_collect_equalities_aexpr ] Fallthrough with " ^ (
      string_of_aexpr aexpr)) in acc

(*
 obj_collect_equalities_astmt : Collects the equality list from an astmt
```

```
*)
and obj_collect_equalities_astmt (acc: ObjectTypeSet.t list) (annotated_stmt: astmt) (
    fname: string) : (ObjectTypeSet.t list) =
  let _ = print_set_list acc in
  match annotated_stmt with
    | ABlock(astmt_list) ->
      List.fold_left (fun acc astmt -> obj_collect_equalities_astmt acc astmt fname)
          acc astmt_list
    | AExpr(ae) -> obj_collect_equalities_aexpr acc ae
    | AReturn(ae) ->
      let acc = match ae with
        | AVal(ret_name, typ) ->
        (match typ with
          | Object(_) ->
            let acc = insert_obj_into_fname_to_obj_equality_hash acc fname ret_name
                in
            acc
          | _ -> acc
        )
        | _ -> acc
      in
       obj_collect_equalities_aexpr acc ae
    | AWhile(ae, _) -> obj_collect_equalities_aexpr acc ae
    | AForIn(ae1, ae2, body) ->
      let new_acc = obj_collect_equalities_aexpr acc ae1 in
      let newer_acc = obj_collect_equalities_aexpr new_acc ae2 in
      let newest_acc = obj_collect_equalities_astmt newer_acc body fname in
      newest_acc
    | AForRange(ae1, ae2, ae3, body) ->
      let new_acc = obj_collect_equalities_aexpr acc ae1 in
      let newer_acc = obj_collect_equalities_aexpr new_acc ae2 in
      let newest_acc = obj_collect_equalities_aexpr newer_acc ae3 in
      let newester_acc = obj_collect_equalities_astmt newest_acc body fname in
      newester_acc
    | AIf(ae, astmt1, astmt2) ->
      let new_acc = obj_collect_equalities_aexpr acc ae in
      let newer_acc = obj_collect_equalities_astmt new_acc astmt1 fname in
      let newest_acc = obj_collect_equalities_astmt newer_acc astmt2 fname in
      newest_acc
    | AObjectInit(obj_list) -> obj_collect_fields_obj_list obj_list acc
    | _ -> acc

and obj_collect_fields_obj_list ( obj_list : (string * data_type) list ) (acc:
    ObjectTypeSet.t list) : (ObjectTypeSet.t list) =
  match obj_list with
    | [] -> acc
    | [obj] -> add_lone_equality acc (fst obj)
```

```
    | obj1 :: obj2 :: tail -> let new_acc = (add_new_equality acc (fst obj1) (fst obj2))
        in
      ( obj_collect_fields_obj_list (obj2 :: tail ) new_acc)


(*
 obj_collect_equalities : Collects a list of object type sets based on operations between
     objects
params:
  astmts: an astmt list to collect equalities from
returns:
  a list of sets where each set contains objects that are of the same type
*)
and obj_collect_equalities (astmts: astmt list ) (acc: ObjectTypeSet.t list) (fname:
    string) : (ObjectTypeSet.t list) =
  let acc = List. fold_left (fun acc astmt -> obj_collect_equalities_astmt acc astmt
      fname) acc astmts in
  acc

(*
[1]

obj_collect : Populate obj_to_fields_hash and obj_equality_sets
params:
  afdecl: a function declaration to collect from
  obj_equality_sets : the list of sets to populate
returns:
  the populated obj_equality_sets
*)
and obj_collect (afdecl: Sast.afunc_decl) (acc: ObjectTypeSet.t list) : (ObjectTypeSet
    .t list) =
  let _ = List.map obj_collect_fields_astmt afdecl.abody in
  let obj_equality_sets = obj_collect_equalities afdecl.abody acc afdecl.afname in
  obj_equality_sets




(* ==================== OBJECT UNIFY
    ==================== *)

(*
obj_unify_one_set: Populate obj_to_id_hash and id_to_fields_hash for one set
params:
  set: a set of objects of the same type
```

```
  id: the current available type id
returns:
  unit
*)
and obj_unify_one_set (set: ObjectTypeSet.t) (id: int) =
  (* Populate id_to_fields hash *)
  let get_obj_fields obj =
    if Hashtbl.mem obj_to_fields_hash obj then Hashtbl.find obj_to_fields_hash obj else
      []
  in
  let fields = List.fold_left (fun acc obj -> acc @ (get_obj_fields obj)) [] (
      ObjectTypeSet.elements set) in
  let _ = Hashtbl.add id_to_fields_hash id fields in

  (* Populate obj_to_id_hash *)
  ObjectTypeSet.iter
    (fun x -> let _ = log ("[obj_unify_one_set] adding " ^ x ^ "->" ^ (string_of_int id)
      ) in
    Hashtbl.add obj_to_id_hash x id)
  set

and obj_unify_lone_object (obj: string) (fields: string list) =
  if Hashtbl.mem obj_to_id_hash obj
  then ()
  else
    let obj_id = gen_new_obj_id() in
    let _ = Hashtbl.add obj_to_id_hash obj obj_id in
    let _ = Hashtbl.add id_to_fields_hash obj_id fields in
    ()

(*
[2]

obj_unify: Populate obj_to_id_hash and id_to_fields_hash for all sets
params:
  obj_equality_sets: the list of sets to populate
returns:
  unit
*)
and obj_unify (obj_equality_sets: ObjectTypeSet.t list) =
  let _ = List.iter (fun x -> let obj_id = gen_new_obj_id() in obj_unify_one_set x
      obj_id) obj_equality_sets in
  Hashtbl.iter (fun k v -> obj_unify_lone_object k v) obj_to_fields_hash
```

```
(* ==================== OBJECT APPLY
    ==================== *)

and obj_apply_aexpr (aexpr: Sast.aexpr) : (Sast. aexpr) =
  match aexpr with
    | AVal(name, typ) ->
      if Hashtbl.mem obj_to_id_hash name
      then AVal(name, Object(Hashtbl.find obj_to_id_hash name))
      else (
        let encoded_list_name = encode_list_name name in
        if Hashtbl.mem obj_to_id_hash encoded_list_name
        then AVal(name, List(Object(Hashtbl.find obj_to_id_hash encoded_list_name)))
        else AVal(name, typ)
      )
    | ABinop(lhs, op, rhs, typ) ->
      let new_lhs = obj_apply_aexpr lhs in
      let new_rhs = obj_apply_aexpr rhs in
      ABinop(new_lhs, op, new_rhs, typ)
    | AUnop(op, ae, _) ->
      let new_ae = obj_apply_aexpr ae in
      AUnop(op, new_ae, type_of_aexpr new_ae)
    | AAssign(id, rhs, _) ->
      let new_rhs = obj_apply_aexpr rhs in
      if is_object_list_decl rhs = true && Hashtbl.mem obj_to_id_hash (
          encode_list_name id) = true
      then
        let new_typ = List(Object(Hashtbl.find obj_to_id_hash (encode_list_name id))) in
        let new_list_decl = inject_list_decl_type new_rhs new_typ in
        AAssign(id, new_list_decl , new_typ)
      else AAssign(id, new_rhs, type_of_aexpr new_rhs)
    | ACall(fname, args, typ) ->
      (match typ with
        | Object(_) -> ACall(fname, (List.map obj_apply_aexpr args), Hashtbl.find
            func_type_hash fname)
        | _ -> ACall(fname, (List.map obj_apply_aexpr args), typ)
      )
    (* Lists *)
    | AListDecl( ae_list , typ) ->
      let new_ae_list = List.map obj_apply_aexpr ae_list in
      AListDecl(new_ae_list, get_list_type_from_listdecl (AListDecl(new_ae_list, typ)))
    | AListInsert(ae1, ae2, ae3, _) ->
      let new_ae1 = obj_apply_aexpr ae1 in
      let new_ae2 = obj_apply_aexpr ae2 in
      let new_ae3 = obj_apply_aexpr ae3 in
      AListInsert(new_ae1, new_ae2, new_ae3, type_of_aexpr new_ae3)
```

```
| AListPush(ae1, ae2, _) ->
  let new_ae1 = obj_apply_aexpr ae1 in
  let new_ae2 = obj_apply_aexpr ae2 in
  AListPush(new_ae1, new_ae2, type_of_aexpr new_ae2)
| AListRemove(ae1, ae2, _) ->
  let new_ae1 = obj_apply_aexpr ae1 in
  let new_ae2 = obj_apply_aexpr ae2 in
  AListRemove(new_ae1, new_ae2, type_of_aexpr new_ae2)
| AListPop(ae, typ) ->
  let new_ae = obj_apply_aexpr ae in
  let obj_id = get_list_type_from_aexpr new_ae in
  if obj_id = -1
  then AListPop(new_ae, typ)
  else AListPop(new_ae, Object(obj_id))
| AListDequeue(ae, typ) ->
  let new_ae = obj_apply_aexpr ae in
  let obj_id = get_list_type_from_aexpr new_ae in
  if obj_id = -1
  then AListDequeue(new_ae, typ)
  else AListDequeue(new_ae, Object(obj_id))
| AListLength(ae, typ) ->
  let new_ae = obj_apply_aexpr ae in
  AListLength(new_ae, typ)
| AListGet(ae1, ae2, typ) ->
  let new_ae1 = obj_apply_aexpr ae1 in
  let new_ae2 = obj_apply_aexpr ae2 in
  let obj_id = get_list_type_from_aexpr new_ae1 in
  if obj_id = -1
  then AListGet(new_ae1, new_ae2, typ)
  else AListGet(new_ae1, new_ae2, Object(obj_id))
| AListSet(ae1, ae2, ae3, _) ->
  let new_ae1 = obj_apply_aexpr ae1 in
  let new_ae2 = obj_apply_aexpr ae2 in
  let new_ae3 = obj_apply_aexpr ae3 in
  AListSet(new_ae1, new_ae2, new_ae3, type_of_aexpr new_ae3)

(* Dicts *)
(* TODO ADictDecl *)
| ADictMem(ae1, ae2, typ) ->
  let new_ae1 = obj_apply_aexpr ae1 in
  let new_ae2 = obj_apply_aexpr ae2 in
  ADictMem(new_ae1, new_ae2, typ)
| ADictDelete(ae1, ae2, typ) ->
  let new_ae1 = obj_apply_aexpr ae1 in
  let new_ae2 = obj_apply_aexpr ae2 in
  let obj_id = get_dict_val_type_from_aexpr new_ae1 in
  if obj_id = -1
```

```
      then ADictDelete(new_ae1, new_ae2, typ)
      else ADictDelete(new_ae1, new_ae2, Object(obj_id))
  | ADictSize(ae, typ) ->
      let new_ae = obj_apply_aexpr ae in
      ADictSize(new_ae, typ)
  | ADictFind(ae1, ae2, typ) ->
      let new_ae1 = obj_apply_aexpr ae1 in
      let new_ae2 = obj_apply_aexpr ae2 in
      let obj_id = get_dict_val_type_from_aexpr new_ae1 in
      if obj_id = -1
      then ADictFind(new_ae1, new_ae2, typ)
      else ADictFind(new_ae1, new_ae2, Object(obj_id))
  | ADictMap(ae1, ae2, ae3, typ) ->
      let new_ae1 = obj_apply_aexpr ae1 in
      let new_ae2 = obj_apply_aexpr ae2 in
      let new_ae3 = obj_apply_aexpr ae3 in
      ADictMap(new_ae1, new_ae2, new_ae3, typ)

  (* Objects *)
  | AObjectField(ae, field, typ) ->
      let new_ae = obj_apply_aexpr ae in
      let new_typ =
        if Hashtbl.mem obj_to_id_hash field
        then Object(Hashtbl.find obj_to_id_hash field)
        else typ
      in
      AObjectField(new_ae, field, new_typ)
  | AObjectAssign(lhs_aexpr, field, rhs_aexpr, typ) ->
      let lhs = obj_apply_aexpr lhs_aexpr in
      let rhs = obj_apply_aexpr rhs_aexpr in
      let new_typ = match typ with
        | Object(_) -> let _ = Hashtbl.add field_hash field (type_of rhs) in type_of rhs
        | List(t) -> (match t with
          | Object(_) -> type_of rhs
          | _ -> typ)
        | Dict(_, t) -> (match t with
          | Object(_) -> type_of rhs
          | _ -> typ)
        | _ -> typ
      in AObjectAssign(lhs, field, rhs, new_typ)
  | _ -> aexpr

(*
obj_apply_stmt: Augment one statement with object types
params:
  astmt: the astmt to augment
  afname: the name of the enclosing function
```

```
returns:
  the augmented astmt
*)
and obj_apply_astmt (astmt: Sast.astmt) (afname: string) : (Sast.astmt) =
  match astmt with
    | ABlock(astmt_list) -> ABlock(List.map (fun astmt -> obj_apply_astmt astmt
      afname) astmt_list)
    | AExpr(ae) -> AExpr(obj_apply_aexpr ae)
    | AReturn(ae) ->
      let new_ae = obj_apply_aexpr ae in
      let typ = type_of_aexpr new_ae in
      let _ =
        (match typ with
          | Object(id) -> let _ = log ("Labeling the return type of " ^ afname ^ " to
            be " ^ string_of_int (id)) in Hashtbl.add func_type_hash afname (Object(
            id))
          | _ -> ()
        )
      in AReturn(new_ae)
    | AWhile(ae, astmt) -> AWhile(obj_apply_aexpr ae, obj_apply_astmt astmt afname)
    | AForIn(ae1, ae2, astmt) -> AForIn(obj_apply_aexpr ae1, obj_apply_aexpr ae2,
      obj_apply_astmt astmt afname)
    | AForRange(ae1, ae2, ae3, astmt) ->
      AForRange(obj_apply_aexpr ae1, obj_apply_aexpr ae2, obj_apply_aexpr ae3,
        obj_apply_astmt astmt afname)
    | AIf(ae, astmt1, astmt2) -> AIf(obj_apply_aexpr ae, obj_apply_astmt astmt1
      afname, obj_apply_astmt astmt2 afname)
    | APrint(ae) -> APrint(obj_apply_aexpr ae)
    | AObjectInit(obj_list) -> AObjectInit(obj_apply_obj_list obj_list)
    | _ -> astmt

and obj_apply_obj_list ( obj_list : (string * data_type) list ) =
  List.map (fun obj ->
    match obj with
      | (name, Object(_)) ->
        if Hashtbl.mem obj_to_id_hash (fst obj)
        then (name, Object(Hashtbl.find obj_to_id_hash (fst obj)))
        else raise (failwith ("Object has no id in obj_apply_obj_list "))
      | _ -> raise (failwith ("Attempting to ObjectInit a non-Object"))
  ) obj_list

and obj_apply_aformals (aformal: (data_type * string)) : (data_type * string) =
  let typ = fst aformal in
  let name = snd aformal in
  if Hashtbl.mem obj_to_id_hash name
  then (Object(Hashtbl.find obj_to_id_hash name), name)
  else (
```

```
    let list_name = encode_list_name name in
    if Hashtbl.mem obj_to_id_hash list_name
    then (List(Object(Hashtbl.find obj_to_id_hash list_name)), name)
    else (typ, name)
  )

(*
obj_apply_function: Augment one afdecl with object types
params:
  afdecl: the afdecl to augment
returns:
  the augmented afdecl
*)
and obj_apply_function (afdecl: Sast.afunc_decl) : (Sast.afunc_decl) =
  let new_afunc_decl = {
    afname = afdecl.afname;
    aformals = List.map obj_apply_aformals afdecl.aformals;
    abody = List.map (fun astmt -> obj_apply_astmt astmt afdecl.afname) afdecl.abody;
    return = afdecl.return;
  } in new_afunc_decl

(* Second pass to apply return types *)
and obj_apply_return_types (afdecl: Sast.afunc_decl) : (Sast.afunc_decl) =
  let new_afunc_decl = {
    afname = afdecl.afname;
    aformals = afdecl.aformals;
    abody = List.map (fun astmt -> obj_apply_astmt astmt afdecl.afname) afdecl.abody;
    return =
      if Hashtbl.mem fname_to_obj_equality_hash afdecl.afname
      then
        let obj_list = Hashtbl.find fname_to_obj_equality_hash afdecl.afname in
        let arbitrary_obj = List.hd obj_list in
        if Hashtbl.mem obj_to_id_hash arbitrary_obj
        then Object(Hashtbl.find obj_to_id_hash arbitrary_obj)
        else afdecl.return
      else afdecl.return;
  } in new_afunc_decl

(*
[3]

obj_apply: Augment the Sast with type ids on all object nodes
params:
  sast: the list of afdecls to augment
returns:
  the augmented list of afdecls
*)
```

```
and obj_apply (sast: Sast.aprogram) =
  let  first_sweep  = List.map obj_apply_function sast in
  List.map obj_apply_return_types first_sweep
;;
```

Pseudo/compiler/Makefile

```
# compiler Makefile
# - builds and manages all compiler components

OCAMLC = ocamlc
OCAMLLEX = ocamllex
OCAMLYACC = ocamlyacc
OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx pseudo.cmx
TESTOBJS = parser.cmo scanner.cmo

default: test

# pseudo: $(OBJS)
#	$(OCAMLC) -g -o pseudo $(OBJS)

.PHONY: test
test: $(TESTOBJS)
	$(OCAMLC) -g -o pseudo $(TESTOBJS)

pseudo.native:
	make clean
	ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis,str,llvm.bitreader -cflags
	    -w,+a-4 \
	        pseudo.native
	@mv pseudo.native pseudo

.PHONY: clean
clean:
	ocamlbuild -clean
	rm -rf testall.log *.diff scanner.ml parser.ml parser.mli pseudo
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o
	rm -rf *.err *.out *.ll

scanner.ml: scanner.mll
	$(OCAMLLEX) scanner.mll

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

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

%.cmi: %.mli
```

```
	$(OCAMLC) -c $<

%.cmx: %.ml
	ocamlfind ocamlopt -c -package llvm $<

# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo sast.cmi
codegen.cmx : ast.cmx sast.cmi
pseudo.cmo : scanner.cmo parser.cmi codegen.cmo ast.cmo printer.cmo infer.cmo
pseudo.cmx : scanner.cmx parser.cmx codegen.cmx ast.cmx printer.cmx infer.cmx
parser.cmo : ast.cmi parser.cmi
parser.cmx : ast.cmi parser.cmi
scanner.cmo : parser.cmi ast.cmi
scanner.cmx : parser.cmx ast.cmi
parser.cmi : ast.cmi
sast.cmi : ast.cmi
infer.cmo : sast.cmi ast.cmi
infer.cmx : sast.cmi ast.cmi
printer.cmo : sast.cmi
printer.cmx : sast.cmi
```

Pseudo/compiler/parser.mly

```
/* Ocamlyacc parser for Pseudo */

%{
open Ast
%}

/* Punctuation */

%token PRINT
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA DEF TILDE DOLLAR
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token COMBINE TO CONCAT
%token EQ NEQ LT LEQ GT GEQ
%token AND OR IN BY
%token RETURN IF ELSE ELSIF FOR WHILE CONTINUE BREAK
%token LBRACKET RBRACKET COLON DOT
%token INSERT POP PUSH DEQUEUE REMOVE LENGTH GET SET
%token MEM FIND MAP DEL SIZE
%token INIT
%token <string> ID
%token EOF

/* Literals */
%token <float> NUM_LITERAL
```

```
%token <string> STRING_LITERAL
%token <bool> BOOL_LITERAL

%right ASSIGN
%left COMBINE
%left OR
%left AND
%left EQ NEQ IS
%left LT GT LEQ GEQ
%left PLUS MINUS CONCAT
%left TIMES DIVIDE
%right NOT
%nonassoc DOT

%start program
%type <Ast.program> program

%%
program:
          func_decl_list  EOF { List.rev $1 }

 func_decl_list :
        /* nothing */ { [] }
        | func_decl_list  func_decl {$2 :: $1}

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

stmt:
          expr SEMI {Expr $1}
        | WHILE expr TILDE loop_stmt_list DOLLAR {While($2, Block(List.rev $4))}
        | FOR ID IN expr TILDE loop_stmt_list DOLLAR {ForIn(Id($2), $4, Block(List.
            rev $6))}
        | FOR expr TO expr TILDE stmt_list DOLLAR {ForRange($2, $4, Num_lit(1.0),
            Block(List.rev $6))}
        | FOR expr TO expr BY expr TILDE stmt_list DOLLAR {ForRange($2, $4, $6,
            Block(List.rev $8))}
        | IF expr TILDE stmt_list DOLLAR elsif_list {If($2, Block(List.rev $4), Block
            ([]))}
        | IF expr TILDE stmt_list DOLLAR elsif_list ELSE TILDE stmt_list DOLLAR {
            If($2, Block(List.rev $4), Block(List.rev $9))}
        | PRINT expr SEMI {Print($2)}
        | INIT obj_list  SEMI {ObjectInit($2)}
        | RETURN SEMI {Return Noexpr}
        | RETURN expr SEMI {Return $2}
```

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

loop_stmt_list :
          /* nothing */ { [] }
        | loop_stmt_list loop_stmt {$2 :: $1}

loop_stmt:
          stmt { $1 }
        | BREAK SEMI { Break }
        | CONTINUE SEMI { Continue }

 elsif_list :
          /* nothing */ { [] }
        |  elsif_list  elsif_stmt {$2 :: $1}

elsif_stmt :
        | ELSIF expr TILDE stmt_list DOLLAR {If($2, Block(List.rev $4), Block([]))}

expr:
          literal {$1}
        | arith_op {$1}
        | bool_op {$1}
        | string_op {$1}
        | LBRACKET pseudo_list RBRACKET {ListDecl(List.rev $2)}
        | list_op {$1}
        | dict_op {$1}
        | obj_op {$1}
        | ID {Id($1)}
        | ID ASSIGN expr {Assign($1, $3)}
        | LPAREN expr RPAREN {$2}
        | LBRACE dict RBRACE {DictDecl(List.rev $2)}
        | ID LPAREN actuals_opt RPAREN {Call($1, $3)}

arith_op:
          MINUS expr          {Unop(Neg, $2)}
        | expr PLUS expr      {Binop($1, Add, $3)}
        | expr MINUS expr     {Binop($1, Minus, $3)}
        | expr TIMES expr     {Binop($1, Times, $3)}
        | expr DIVIDE expr    {Binop($1, Divide, $3)}

bool_op:
          NOT expr            {Unop(Not, $2)}
        | expr EQ expr  {Binop($1, Eq, $3)}
        | expr NEQ expr {Binop($1, Neq, $3)}
        | expr LT expr  {Binop($1, Less, $3)}
```

```
        | expr LEQ expr {Binop($1, Leq, $3)}
        | expr GT expr  {Binop($1, Greater, $3)}
        | expr GEQ expr {Binop($1, Geq, $3)}
        | expr AND expr {Binop($1, And, $3)}
        | expr OR expr {Binop($1, Or, $3)}

string_op:
          expr CONCAT expr {Binop($1, Concat, $3)}

pseudo_list:
          /* nothing */ { [] }
        | expr { [$1] }
        | pseudo_list COMMA expr { $3 :: $1 }

dict:
          /* nothing */ { [] }
        | expr COLON expr { [($1, $3)] }
        | dict COMMA expr COLON expr { ($3, $5) :: $1 }

list_op :
        | expr COMBINE expr { Binop($1, Combine, $3) }
        | expr DOT INSERT LPAREN expr COMMA expr RPAREN { ListInsert($1, $5
            , $7) }
        | expr DOT PUSH LPAREN expr RPAREN { ListPush($1, $5) }
        | expr DOT POP LPAREN RPAREN { ListPop($1) }
        | expr DOT DEQUEUE LPAREN RPAREN { ListDequeue($1) }
        | expr DOT REMOVE LPAREN expr RPAREN { ListRemove($1, $5) }
        | expr DOT LENGTH LPAREN RPAREN { ListLength($1) }
        | expr DOT GET LPAREN expr RPAREN { ListGet($1, $5) }
        | expr DOT SET LPAREN expr COMMA expr RPAREN { ListSet($1, $5, $7) }
/*      | expr LBRACKET expr RBRACKET { ListGet($1, $3) }*/
/*      | expr LBRACKET expr RBRACKET ASSIGN expr { ListSet($1, $3, $6) }*/

dict_op:
        | expr DOT MEM LPAREN expr RPAREN { DictMem($1, $5) }
        | expr DOT FIND LPAREN expr RPAREN { DictFind($1, $5) }
        | expr DOT MAP LPAREN expr COMMA expr RPAREN { DictMap($1, $5, $7
            ) }
        | expr DOT DEL LPAREN expr RPAREN { DictDelete($1, $5) }
        | expr DOT SIZE LPAREN RPAREN { DictSize($1) }
/*      | expr LBRACE expr RBRACE { DictFind($1, $3) }*/
/*      | expr LBRACE expr RBRACE ASSIGN expr { DictMap($1, $3, $6) }*/

obj_op:
    | expr DOT ID {ObjectField($1, $3)}
    | expr DOT ID ASSIGN expr {ObjectAssign($1, $3, $5)}
```

```
func_decl:
        DEF ID LPAREN formal_opt RPAREN TILDE stmt_list DOLLAR
            { {
                fname = $2;
                formals = $4;
                body = List.rev $7
            } }

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

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

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

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

literal :
      | NUM_LITERAL       {Num_lit($1)}
      | STRING_LITERAL    { String_lit ($1)}
      | BOOL_LITERAL      {Bool_lit ($1)}
```

Pseudo/compiler/preprocessor.py

```
import re
import sys


def truncate_comment(line):
    in_comment = False
    for i, c in enumerate(line):
        if c == '"':
            in_comment = not in_comment
        elif not in_comment and i < len(line) - 1 and line[i:i+2] == '//':
            return line[:i]
    return line


def preprocess(file, outfile=sys.stdout):
    regex = re.compile('^def|^while|^for|^if|^else')
    prev_tabs = 0
    out_line = None
```

```
    num_tabs = 0
    out_code = []

    compressed = []
    prev_line = ""
    line = ""
    for line in file:
        left_str = line.strip('\n\t\r')

        if (len(prev_line.strip('\n\t\r')) > 0 or left_str.startswith('def')):
            compressed.append(prev_line)
        prev_line = line
    if len(line.strip('\n\t\r')) > 0:
        compressed.append(line)

    for line in compressed:
        leading_spaces = 4
        num_tabs = len(line) - len(line.lstrip('\t'))
        num_spaces = len(line) - len(line.lstrip(' '))
        if (num_tabs > 0 and num_spaces > 0):
            raise ValueError('Mixing spaces and tabs')
        else:
            num_tabs = max(num_tabs, num_spaces / leading_spaces)

        line = truncate_comment(line)
        line = line.strip('\n\t\r')

        if (len(line) > 0):
            out_line = line.rstrip('\n\t\r')
            if line.endswith(':') and re.match(regex, line):
                out_line = line[:len(line) - 1] + '~\n'
            elif len(line) > 0:
                out_line = out_line + ';\n'
            if num_tabs < prev_tabs:
                out_line = '$' * (prev_tabs - num_tabs) + out_line
                if not out_line.endswith('\n'):
                    out_line = out_line + '\n'
            out_code.append(out_line)
            prev_tabs = num_tabs
    end_dollars = num_tabs * '$'
    out_code.append(end_dollars)
    # print (''.join(out_code))
    outfile.write (''.join(out_code))


if __name__ == '__main__':
    prog_name = sys.argv[1]
```

```
    outfile  = sys.stdout
    if  len(sys.argv) > 2:
        outfile  = open(sys.argv[2],  'w')

    with open(prog_name, 'r') as  file :
        preprocess( file ,  outfile )

    if  len(sys.argv) > 2:
        outfile . close ()
```

Pseudo/compiler/printer.ml

```
open Ast
open Sast
open Printf

(* Unary operators *)
let  txt_of_unop = function
  | Neg -> "\"Neg\""
  | Not -> "\"Not\""

let  txt_of_prim = function
  | Num -> "Num"
  | Bool -> "Bool"
  | String -> "String"
  | Void -> "Void"
  | _ -> "Not prim"

(* Data type *)
let  txt_of_data_type = function
  | Num -> "\"Num\""
  | Bool -> "\"Bool\""
  | String -> "\"String\""
  | Void -> "\"Void\""
  | Object(id) -> sprintf "\"Object(%s)\"" (string_of_int id)
  | List(d) -> sprintf "\"List(%s)\"" (txt_of_prim d)
  | Undetermined -> "\"Undetermined\""
  | T(s) -> sprintf "\"T(%s)\"" s


(* Binary operators *)
let  txt_of_binop = function
  (* Arithmetic *)
  | Add -> "\"Add\""
  | Minus -> "\"Minus\""
  | Times-> "\"Times\""
  | Divide -> "\"Divide\""
  (* Boolean *)
  | Or -> "\"Or\""
```

```
  | And -> "\"And\""
  | Eq -> "\"Eq\""
  | Neq -> "\"Neq\""
  | Less -> "\"Less\""
  | Leq -> "\"Leq\""
  | Greater -> "\"Greater\""
  | Geq -> "\"Geq\""
  (* List *)
  | Combine -> "\"Combine\""
  (* String *)
  | Concat -> "\"Concat\""

(* Expressions *)
(*let txt_of_num = function
  | Num_int(x) -> string_of_int x
  | Num_float(x) -> string_of_float x *)

let rec txt_of_aexpr = function
  | ANumLit(x, t) -> sprintf "{ \"aexpr\": \"ANumLit\", \"data_type\": %s,      \"val
      \" : \"%s\" } " (txt_of_data_type t) ( string_of_float  x)
  | ABoolLit(x, t) -> sprintf "{ \"aexpr\": \"ABoolLit\",    \"data_type\": %s,      \"
      val\": \"%s\" }" (txt_of_data_type t) ( string_of_bool  x)
  | AStringLit(x, t) -> sprintf "{ \"aexpr\": \"AStringLit\",     \"data_type\": %s,
        \"val\": \"%s\" }" (txt_of_data_type t) x
  | AVal(s, d) -> sprintf "{ \"aexpr\": \"AVal\",     \"val_name\": \"%s\",     \"
      data_type\": %s }" s (txt_of_data_type d)
  (*| Id(x) -> sprintf "Id(%s)" x *)
  | ABinop(e1, op, e2, t) -> sprintf "{ \"aexpr\": \"ABinop\",    \"e1\": %s,      \"op
      \": %s,      \"e2\": %s,      \"val\": %s }"
      (txt_of_aexpr e1) (txt_of_binop op) (txt_of_aexpr e2) (txt_of_data_type t)
  | AUnop(op, e, t) -> sprintf "{ \"aexpr\": \"AUnop\",   \"e\": %s,      \"op\": %s,
        \"val\": %s }" (txt_of_unop op) (txt_of_aexpr
  e) (txt_of_data_type t)
  | AAssign(x, e, t) -> sprintf "{ \"aexpr\": \"AAssign\",     \"var\": \"%s\",      \"e
      \": %s,      \"data_type\": %s }" x (txt_of_aexpr e)
  (txt_of_data_type t)
  | ACall(f, args, t) -> sprintf "{ \"aexpr\": \"AACall\",     \"name\": \"%s\", \"
      args\": [%s],      \"data_type\": %s }"
      f ( txt_of_list  args) (txt_of_data_type t)
  | ANoexpr(t) -> sprintf "Noexpr(%s)" (txt_of_data_type t)

  (* List Operations *)
  | AListDecl(l, d) -> sprintf "{ \"aexpr\": \"AListDecl\",     \"val\": [%s],       \"
      data_type\": %s }" ( txt_of_list  l) (txt_of_data_type d)
  | AListInsert(e1, e2, e3, d) -> sprintf "{ \"aexpr\": \"AListInsert\",      \"e1\": %s
      ,      \"e2\": %s,      \"e3\": %s,\"data_type\": %s }" (txt_of_aexpr e1) (
      txt_of_aexpr e2) (txt_of_aexpr e3) (txt_of_data_type d)
```

```
  | AListPush(e1, e2, d) -> sprintf "{ \"aexpr\": \"AListPush\",    \"e1\": %s,      \"
      e2\": %s,      \"data_type\": %s }" (txt_of_aexpr e1) (txt_of_aexpr e2) (
      txt_of_data_type d)
  | AListRemove(e1, e2, d) -> sprintf "{ \"aexpr\": \"AListRemove\",  \"e1\": %s,
          \"e2\": %s,      \"data_type\": %s }" (txt_of_aexpr e1) (txt_of_aexpr e2) (
      txt_of_data_type d)
  | AListPop(e, d) -> sprintf "{ \"aexpr\": \"AListPop\",    \"e\": %s,      \"
      data_type\": %s }" (txt_of_aexpr e) (txt_of_data_type d)
  | AListDequeue(e, d) -> sprintf "{ \"aexpr\": \"AListDequeue\",  \"e\": %s,      \"
      data_type\": %s }" (txt_of_aexpr e) (txt_of_data_type d)
  | AListLength(e, d) -> sprintf "{ \"aexpr\": \"AListLength\",    \"e\": %s,      \"
      data_type\": %s }" (txt_of_aexpr e) (txt_of_data_type d)
  | AListGet(e1, e2, d) -> sprintf "{ \"aexpr\": \"AListRemove\",   \"e1\": %s,
      \"e2\": %s,      \"data_type\": %s }" (txt_of_aexpr e1) (txt_of_aexpr e2) (
      txt_of_data_type d)
  | AListSet(e1, e2, e3, d) -> sprintf "{ \"aexpr\": \"AListSet\",     \"e1\": %s,
      \"e2\": %s,      \"e3\": %s,\"data_type\": %s }" (txt_of_aexpr e1) (txt_of_aexpr
      e2) (txt_of_aexpr e3) (txt_of_data_type d)
  (* Objects *)
  | AObjectField(e, s, d) -> sprintf "{ \"aexpr\": \"AObjectField\",    \"e\": %s,
      \"s\": \"%s\",     \"data_type\": %s }" (txt_of_aexpr e) s (txt_of_data_type d)
  | AObjectAssign(e1, s, e2, d) -> sprintf "{ \"aexpr\": \"AObjectAssign\", \"e1\": %
      s,   \"s\": \"%s\",     \"e2\": %s,      \"data_type\": %s }" (txt_of_aexpr e1) s (
      txt_of_aexpr e2) (txt_of_data_type d)

and txt_of_objlist  obj_list =
  let rec aux acc = function
      | [] -> sprintf "[%s]" (String.concat " , " (acc))
      | obj :: tl -> aux (sprintf "(%s, %s)" (fst obj) (txt_of_data_type (snd obj)) ::
          acc) tl
  in aux []  obj_list

(* Lists *)
and txt_of_list = function
  | [] -> ""
  | [x] -> txt_of_aexpr x
  | _ as l -> String.concat " , " (List.map txt_of_aexpr l)

and txt_of_aformal = function
    | (d, s) -> sprintf "{ %s:\"%s\" }" (txt_of_data_type d) s

and txt_of_aformals (aformals) =
  let rec aux acc = function
      | [] -> sprintf "%s" (String.concat " , " (List.rev acc))
      | aformal :: tl -> aux (txt_of_aformal aformal:: acc) tl
  in aux [] aformals
```

```
(* Statements *)
and txt_of_astmt = function
    | AExpr(e) -> sprintf " { \"astmt\": \"AExpr\",    \"e\": %s }" (txt_of_aexpr e)
    | AReturn(e) -> sprintf " { \"astmt\": \"AReturn \",    \"e\": %s }" (txt_of_aexpr
        e)
    | ABreak -> sprintf " { \"astmt\": \"Break \"}"
    | AContinue -> sprintf " { \"astmt\": \"Break \"}"
    | AWhile(e, s) -> sprintf "  { \"aexpr\": \"AWhile\",    \"e\": %s,      \"s\": %s
       }" (txt_of_aexpr e) (txt_of_astmt s)
    | AForIn(e1, e2, s) -> sprintf "  { \"aexpr\": \"AForIn\",    \"e1\": %s,      \"e2
       \": %s,      \"s\": %s}" (txt_of_aexpr e1)
    (txt_of_aexpr e2) (txt_of_astmt s)
    | AForRange(e1, e2, e3, s) -> sprintf "  { \"aexpr\": \"AForRange\",   \"e1\": %s
       ,      \"e2\": %s,         \"e3\": %s,          \"s\": %s }" (txt_of_aexpr e1)
    (txt_of_aexpr e2) (txt_of_aexpr e3) (txt_of_astmt s)
    | AIf(e1, s1, s2) -> sprintf "  { \"aexpr\": \"AFIf\",    \"e1\": %s,      \"s1\":
       %s,      \"s2\": %s }" (txt_of_aexpr e1)

    (txt_of_astmt s1)  (txt_of_astmt s2)
    | ABlock(asl) -> (txt_of_astmts asl)
    | APrint(e) -> sprintf "  { \"astmt\": \"APrint\",    \"e\": %s }" (txt_of_aexpr e)
    | AObjectInit(obj_list) -> sprintf "  { \"astmt\": \"AObjectInit\",    \" objlist \":
       %s }" ( txt_of_objlist  obj_list )

and txt_of_astmts (astmts) =
  let rec aux acc = function
      | [] -> sprintf "[  %s  ] " (String.concat " , " (List.rev acc))
      | astmt :: tl -> aux (txt_of_astmt astmt :: acc) tl
  in aux [] astmts

(* Function declarations *)
and txt_of_afdecl (f: afunc_decl): string =
    sprintf " {    \"func name\" : \"%s\",        \"params\" : [%s],      \"body\": %s
       }" (f.afname)
                        (txt_of_aformals f.aformals) (txt_of_astmts f.abody)


and txt_of_afdecls afdecls =
  let rec aux acc = function
      | [] -> sprintf "[ %s ]" (String.concat " ,       " (List.rev acc))
      | afdecl :: tl -> aux ((txt_of_afdecl afdecl) :: acc) tl
  in aux [] afdecls

let print_sast aprogram =
    let sast_str = txt_of_afdecls aprogram in
    print_endline sast_str
```

Pseudo/compiler/pseudo.ml

```
open Printf

type action = Sast | SastNoPreprocess | Compile | CompileNoPreprocess |
    CompileAndRun

let read_file filename =
  let lines = ref [] in
  let chan = open_in filename in
    try
      while true; do
        lines := input_line chan :: !lines
      done; !lines
    with End_of_file ->
      close_in chan;
      List.rev !lines

(* Usage: ./pseudo [myprogram.clrs myprogram.ll] [-s myprogram.clrs] [-sp myprogram.
    clrs] *)
let _ =
  (* Get action from flag *)
  let action =
    if Sys.argv.(1) = "-s"
    then Sast
    else (
      if Sys.argv.(1) = "-sp"
      then SastNoPreprocess
      else (
        if Sys.argv.(1) = "-p"
        then CompileNoPreprocess
        else (
          if Array.length Sys.argv > 2
          then Compile
          else CompileAndRun
        )
      )
    )
  in
  (* Grab correct source code file *)
  let infile_arg_index =
    if action = Sast || action = SastNoPreprocess || action = CompileNoPreprocess
    then 2
    else 1
  in
  let infile_name = Sys.argv.( infile_arg_index ) in

  (* Pre-process if compile option is selected *)
  let processed_infile_name =
```

```
  if action = Sast || action = Compile || action = CompileAndRun
  then (* need to preprocess *)
    let command_str = sprintf ("python preprocessor.py %s %s") (infile_name) ("temp.
        pclrs") in
    let _ = Sys.command(command_str)
    in "temp.pclrs"
  else infile_name
in
(* Scan and Parse *)
let  infile_str  = List.fold_left  (fun s lst -> s ^ lst) "" ( read_file
    processed_infile_name) in
let _ = Sys.command(sprintf "rm -f %s" "temp.pclrs") in
let lexbuf = Lexing.from_string infile_str in
let ast = Parser.program Scanner.token lexbuf in
let sast, obj_table, obj_to_id_hash = Infer.infer_all ast in

match action with
  | Compile | CompileNoPreprocess ->
    let m = Codegen.translate sast obj_table obj_to_id_hash in
    Llvm_analysis.assert_valid_module m;
    let outfile_index =
      if action = Compile
      then 2
      else 3
    in
    let outfile_name = Sys.argv.(outfile_index) in
    let oc = open_out outfile_name in
    Printf.fprintf oc "%s" (Llvm.string_of_llmodule m);
    close_out oc;
  | CompileAndRun -> let m = Codegen.translate sast obj_table obj_to_id_hash in
    Llvm_analysis.assert_valid_module m;
    let lli = "lli-3.7" in
    let run_str = sprintf "%s <<< '%s'" (lli) (Llvm.string_of_llmodule m) in
    let _ = Sys.command(run_str) in ()
  | Sast | SastNoPreprocess -> Printer.print_sast sast
```

Pseudo/compiler/sast.mli

```
open Ast

type data_type =
  | Num
  | Bool
  | String
  | Void
  | List of data_type
  | Dict of data_type * data_type
  | Undetermined
  | T of string
```

```
  | Object of int (* object type id *)

type aexpr =
  | ANumLit of float * data_type
  | ABoolLit of bool * data_type
  | AStringLit of string * data_type
  | AVal of string * data_type
  | ABinop of aexpr * binop * aexpr * data_type
  | AUnop of unop * aexpr * data_type
  | AAssign of string * aexpr * data_type
  | ACall of string * aexpr list * data_type
  | ANoexpr of data_type
  (* List Ops *)
  | AListDecl of aexpr list * data_type
  | AListInsert of aexpr * aexpr * aexpr * data_type
  | AListPush of aexpr * aexpr * data_type
  | AListRemove of aexpr * aexpr * data_type
  | AListPop of aexpr * data_type
  | AListDequeue of aexpr * data_type
  | AListLength of aexpr * data_type
  | AListGet of aexpr * aexpr * data_type
  | AListSet of aexpr * aexpr * aexpr * data_type
  (* Dict Ops *)
  | ADictDecl of aexpr list * aexpr list * data_type
  | ADictMem of aexpr * aexpr * data_type
  | ADictDelete of aexpr * aexpr * data_type
  | ADictSize of aexpr * data_type
  | ADictFind of aexpr * aexpr * data_type
  | ADictMap of aexpr * aexpr * aexpr * data_type
  (* Object Ops *)
  | AObjectField of aexpr * string * data_type
  | AObjectAssign of aexpr * string * aexpr * data_type

(* Statements *)
type astmt =
    ABlock of astmt list
  | AExpr of aexpr
  | AReturn of aexpr
  | ABreak
  | AContinue
  | AWhile of aexpr * astmt
  | AForIn of aexpr * aexpr * astmt
  | AForRange of aexpr * aexpr * aexpr * astmt
  | AIf of aexpr * astmt * astmt
  | APrint of aexpr
  | AObjectInit of (string * data_type) list
```

```
(* Function Declarations *)
type afunc_decl = {
  afname: string;
  aformals: (data_type * string) list;  (* Parameters *)
  abody: astmt list;        (* Function Body *)
  return: data_type;
}

(* Program entry point *)
type aprogram = afunc_decl list
```

Pseudo/compiler/scanner.mll

```
(* Ocamllex scanner for MicroC *)

{ open Parser }

let numeric = ['0'-'9']

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"      { comment lexbuf }           (* Comments *)
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LBRACKET }
| ']'       { RBRACKET }
| ','       { COMMA }
| '.'       { DOT }
| ':'       { COLON }
| '~'       { TILDE }
| '$'       { DOLLAR }
| ';'       { SEMI }
(* BINOPS *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '^'       { CONCAT }
| "::"      { COMBINE }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&" | "and"   { AND }
```

```
| "||"  | "or"       { OR }
| "!"  | "not"       { NOT }

(* STRUCTURE KEYWORDS *)
| "def"      { DEF }
| "if"       { IF }
| "else"     { ELSE }
| "elsif"    { ELSIF }
| "for"      { FOR }
| "while"    { WHILE }
| "return"   { RETURN }
| "print"    { PRINT }
| "to"       { TO }
| "in"       { IN }
| "by"             { BY }
| "continue" { CONTINUE }
| "break"    { BREAK }

(* LIST KEYWORDS *)
| "get" { GET }
| "set" { SET }
| "length" { LENGTH }
| "insert" { INSERT }
| "remove" { REMOVE }
| "push" | "enqueue" | "append" { PUSH }
| "pop" { POP }
| "dequeue" { DEQUEUE }

(* MAP KEYWORDS *)
| "mem" { MEM }
| "find" { FIND }
| "map" { MAP }
| "delete" { DEL }
| "size" { SIZE }

(* OBJECT KEYWORDS *)
| "init" { INIT }

(* DATA TYPES *)
| "true" | "false" as boollit  { BOOL_LITERAL(bool_of_string boollit)}
| numeric* '.' numeric+
| numeric+ '.'numeric* as floatlit
        { NUM_LITERAL(float_of_string floatlit)}

| numeric+ as intlit  { NUM_LITERAL(float_of_string intlit) }
| '"'  (([^  '"'] | "\\\"")* as strlit )  '"'  { STRING_LITERAL(strlit) }
| ['a'-'z'  'A'-'Z']['a'-'z'  'A'-'Z' '0'-'9'  '_']* as lxm { ID(lxm) }
```

```
| eof { EOF }
| _ as char { raise (Failure(" illegal character " ^ Char.escaped char)) }


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

Pseudo/tests/Makefile

```
# test Makefile
# - builds all files needed for testing, then runs tests

default: test

all:
	cd ..; make all
	make

test: build
	./test_preprocessor.sh
	./test_scanner.sh
	./test_parser.sh
	cd ../compiler; make pseudo.native
	cd ../tests; ./test_inference.sh
	./test_end_to_end.sh
	./test_fail.sh

build:
	cd scanner; make
	cd parser; make

.PHONY: clean
clean:
	rm -f caml ocaml
	cd scanner; make clean
	cd parser; make clean
	rm -f end_to_end/*.clrs
```

Pseudo/tests/test_end_to_end.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="end_to_end/pass/*.in"
```

```
LLI="lli-3.7"
PSEUDO="../compiler/pseudo"
printf "${CYAN}Running end-to-end tests...\n${NC}"

for input_file in $INPUT_FILES; do
        target=${input_file%%.*}
        # python ../compiler/preprocessor.py $input_file > $target.clrs
        cd ../compiler
        input=$(./pseudo ../tests/$target.in | tr -d "[:space:]")
        output=$(tr -d "[:space:]" < ../tests/$target.out);
        if [[ "$input" == "$output" ]]; then
                printf "%-65s ${GREEN}SUCCESS\n${NC}" " - checking $input_file
                    ..."
        else
                printf "%-65s ${RED}ERROR\n${NC}" " - checking $input_file..."
                    1>&2
        printf "%s" $input
        printf "\n"
        fi
done
exit 0
```

Pseudo/tests/test_fail.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="end_to_end/fail/*.in"
LLI="lli-3.7"
PSEUDO="../compiler/pseudo"
printf "${CYAN}Running fail tests...\n${NC}"

for input_file in $INPUT_FILES; do
        target=${input_file%%.*}
        cd ../compiler
    $(./pseudo ../tests/$target.in > /dev/null 2>&1)
    result=$?
        if [[ "$result" -ne 0 ]] ; then
                printf "%-65s ${GREEN}SUCCESS\n${NC}" " - checking $input_file
                    ..."

        else
        printf "%-65s ${RED}ERROR\n${NC}" " - checking $input_file..." 1>&2

        fi
```

```
done
exit 0
```

Pseudo/tests/test_inference.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="inference/_*.in"
printf "${CYAN}Running inference tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
    input=$(../compiler/pseudo -sp $input_file | tr -d "[:space:]")
    output=$(tr -d "[:space:]" < $output_file);
    if [[ "$input" == "$output" ]]; then
        printf "%-65s ${GREEN}SUCCESS\n${NC}" " - checking $input_file..."
    else
        printf "%-65s ${RED}ERROR\n${NC}" " - checking $input_file..." 1>&2
        printf $input
        printf "\n"
    fi
done

exit 0
```

Pseudo/tests/test_parser.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="parser/_*.in"
printf "${CYAN}Running parser tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
    input=$(parser/parserize < $input_file | tr -d "[:space:]")
    # printf $input
    # printf "\n"
    output=$(tr -d "[:space:]" < $output_file);
    if [[ "$input" == "$output" ]]; then
```

```
        printf "%-65s ${GREEN}SUCCESS\n${NC}" " - checking $input_file..."
    else
        printf "%-65s ${RED}ERROR\n${NC}" " - checking $input_file..." 1>&2
    fi
done

exit 0
```

Pseudo/tests/test_preprocessor.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="preprocessor/*.in"
PREPROCESSOR="../compiler/preprocessor.py"
printf "${CYAN}Running preprocessor tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}

    input=$(python $PREPROCESSOR $input_file | tr -d "[:space:]")
    output=$(tr -d "[:space:]" < $output_file);
    if [[ "$input" == "$output" ]]; then
        printf "%-65s ${GREEN}SUCCESS\n${NC}" " - checking $input_file..."
    else
        printf "%-65s ${RED}ERROR\n${NC}" " - checking $input_file..." 1>&2
    fi
done

exit 0
```

Pseudo/tests/test_scanner.sh

```
#!/bin/bash

NC='\033[0m'
CYAN='\033[0;36m'
GREEN='\033[0;32m'
RED='\033[0;31m'

INPUT_FILES="scanner/*.in"
printf "${CYAN}Running scanner tests...\n${NC}"

for input_file in $INPUT_FILES; do
    output_file=${input_file/.in/.out}
```

```
    input=$(scanner/tokenize < $input_file | tr -d "[:space:]")
    output=$(tr -d "[:space:]" < $output_file)

    if [ "$input" == "$output" ]; then
        printf "%-65s ${GREEN}SUCCESS\n${NC}" " - checking $input_file..."
    else
        printf "%-65s ${RED}ERROR\n${NC}" " - checking $input_file..." 1>&2
    fi
done

exit 0
```

Pseudo/tests/end_to_end/fail/test_binop.in

```
def main():
    b = 1 ^ "foo"
```

Pseudo/tests/end_to_end/fail/test_binop.out

```
Fatal error: exception Not_found
```

Pseudo/tests/end_to_end/fail/test_call_notfunc.in

```
def main():
    a.val = 1
    a()
```

Pseudo/tests/end_to_end/fail/test_call_notfunc.out

```
Fatal error: exception Not_found
```

Pseudo/tests/end_to_end/fail/test_fnodecl.in

```
def main():
    foo()
```

Pseudo/tests/end_to_end/fail/test_fnodecl.out

```
Fatal error: exception Not_found
```

Pseudo/tests/end_to_end/fail/test_nested_func.in

```
def main():
    def foo():
        print 1
```

Pseudo/tests/end_to_end/fail/test_nested_func.out

```
Fatal error: exception Parsing.Parse_error
```

Pseudo/tests/end_to_end/fail/test_nomain.in

```
def foo():
    print 1
```

Pseudo/tests/end_to_end/fail/test_nomain.out

```
Fatal error: exception Failure("main function not found")
```

Pseudo/tests/end_to_end/fail/test_out_stmt.in

```
print 1
```

Pseudo/tests/end_to_end/fail/test_out_stmt.out

```
Fatal error: exception Failure("Variable not defined")
```

Pseudo/tests/end_to_end/fail/test_scope.in

```
def main():
    if (true):
        b = 10
    print b
```

Pseudo/tests/end_to_end/fail/test_scope.out

```
Fatal error: exception Failure("Variable not defined")
```

Pseudo/tests/end_to_end/fail/test_type.in

```
def main():
    a = 1
    b = a + "foo"
```

Pseudo/tests/end_to_end/fail/test_type.out

```
Fatal error: exception Failure("mismatched types")
```

Pseudo/tests/end_to_end/fail/test_undefined_var.in

```
def main():
        print x
```

Pseudo/tests/end_to_end/fail/test_undefined_var.out

```
Fatal error: exception Failure("Variable not defined")
```

Pseudo/tests/end_to_end/fail/test_while.in

```
def main():
    while(-1):
        print 1
```

Pseudo/tests/end_to_end/fail/test_while.out

```
Fatal error: exception Failure("expected boolean in if statement")
```

Pseudo/tests/end_to_end/pass/test-assignment.in

```
def main():
        x = 3
        print x
        x = 4 * 5 + 8 / 2
        print x
```

Pseudo/tests/end_to_end/pass/test-assignment.out

```
3.00
24.00
```

Pseudo/tests/end_to_end/pass/test-binop.in

```
def main():
        x = 4
        print 4 + 0
        print 18 - 24
        print -2 * 3
        print 7/2
        print x == 4
        print x < 5
        print x <= 200
        print x > 2
        print x >= 0
        print 4 > 5 or 2 < 7
        print 1 == 1 and 5 > 0
        print 3 != 6
        print ("Hello " ^ "World")
```

Pseudo/tests/end_to_end/pass/test-binop.out

```
4.00
-6.00
-6.00
3.50
1
1
1
1
1
1
1
1
Hello World
```

Pseudo/tests/end_to_end/pass/test-default_assign.in

```
def main():
        init a, b
        a.name = "Ben"
        a.age = 23
        b.friend = "Joe"
        print b.name
        print b.age
        print a.friend

        init c, d
        c.person = a
        c.arr = [1,2,3]
        print c.person.name
        print c.person.age
        if c.arr.get(0) == 1:
                e = c.arr.get(0) + 4
                print e
```

Pseudo/tests/end_to_end/pass/test-default_assign.out

```
0.00

Ben
23.00
5.00
```

Pseudo/tests/end_to_end/pass/test-fibonacci.in

```
def main():
	print fib(5)

def fib(n):
	if n == 1 or n == 0:
		return 1
	else:
		return fib(n-1) + fib(n-2)
```

Pseudo/tests/end_to_end/pass/test-fibonacci.out

```
8.00
```

Pseudo/tests/end_to_end/pass/test-for-in-list.in

```
def main():
	a = [1, 2, 3, 4, 5]

	print "FOR IN"
	for i in a:
		print i

	print ""
	print "FOR RANGE"
	for i = 0 to a.length():
		print a.get(a.length()-1-i)
```

Pseudo/tests/end_to_end/pass/test-for-in-list.out

```
FOR IN
1.00
2.00
3.00
4.00
5.00

FOR RANGE
5.00
4.00
3.00
2.00
1.00
```

Pseudo/tests/end_to_end/pass/test-for-range-by.in

```
def main():
	for i = 0 to 10 by 2:
		print i
```

Pseudo/tests/end_to_end/pass/test-for-range-by.out

```
0.00
2.00
4.00
6.00
8.00
```

Pseudo/tests/end_to_end/pass/test-for.in

```
def main():
        for i=0 to 5:
                print i
```

Pseudo/tests/end_to_end/pass/test-for.out

```
0.00
1.00
2.00
3.00
4.00
```

Pseudo/tests/end_to_end/pass/test-func.in

```
def main():
        printer()
        c = greater(3, 7)
        print c

def printer():
        print "Hello World!"

def greater(a, b):
        if a > b:
                return a
        if b > a:
                return b
        else:
                return 0
```

Pseudo/tests/end_to_end/pass/test-func.out

```
Hello World!
7.00
```

Pseudo/tests/end_to_end/pass/test-hello_world.in

```
def main():
        print 1
```

Pseudo/tests/end_to_end/pass/test-hello_world.out

```
1.00
```

Pseudo/tests/end_to_end/pass/test-list-in-obj.in

```
def main():
        a.numbers = [1,2,3]
        b = a.numbers.get(0)
        print b
        print a.numbers.get(1)
        print a.numbers.get(2)
```

Pseudo/tests/end_to_end/pass/test-list-in-obj.out

```
1.00
2.00
3.00
```

Pseudo/tests/end_to_end/pass/test-list-of-objects.in

```
def main():
        a.age = 21
        b.name = "Raymond"
         list  = [a, b]
        x = list.get(0)
        print x
        y = list.get(1)
        print y
        print  list.get(0).age
        print  list.get(1).age
```

Pseudo/tests/end_to_end/pass/test-list-of-objects.out

```
x.age: 21.00
x.name:
y.age: 0.00
y.name: Raymond
21.00
0.00
```

Pseudo/tests/end_to_end/pass/test-list-print.in

```
def main():
        a = ["a", "b", "c", "d"]
        print a

        b = [1, 2, 3, 4]
        print b

        c =[true, false, true]
        print c
```

Pseudo/tests/end_to_end/pass/test-list-print.out

```
[a, b, c, d]
[1.00, 2.00, 3.00, 4.00]
[1, 0, 1]
```

Pseudo/tests/end_to_end/pass/test-list-str.in

```
def main():
        a = ["P","L","T"]
        print a.get(0)
        print a.get(1)
        print a.get(2)
```

Pseudo/tests/end_to_end/pass/test-list-str.out

```
P
L
T
```

Pseudo/tests/end_to_end/pass/test-list.in

```
def main():
        a = [10,9,8,7]
        print a.get(0)
        print a.get(1)
        print a.get(2)
        print a.get(3)
```

Pseudo/tests/end_to_end/pass/test-list.out

```
10.00
9.00
8.00
7.00
```

Pseudo/tests/end_to_end/pass/test-listops.in

```
def main():
        a = [5,5,5,5]
        a.set(3,1)
        a.set(2,2)
        a.set(1,3)
        a.set(0,4)
        print a.get(0)
        print a.get(1)
        print a.get(2)
        print a.get(3)
        print ""
        print "a is [4, 3, 2, 1]"
        print ""
        print "PUSH"
        a.push(0)
        print a.get(4)
        print a.length()
        print ""
        print "DEQUEUE"
        print a.dequeue()
        print a.length()
        print ""
        print "POP"
        print a.pop()
        print a.length()
        print ""
        print "INSERT"
        print a.insert(0, 4)
        print a.get(0)
        print a.get(1)
        print a.get(2)
        print a.get(3)
        print a.length()
        print ""
```

```
        print "REMOVE"
        a.remove(0)
        print a.get(0)
        print a.length()
```

Pseudo/tests/end_to_end/pass/test-listops.out

```
4.00
3.00
2.00
1.00

a is [4, 3, 2, 1]

PUSH
0.00
5.00

DEQUEUE
4.00
4.00

POP
0.00
3.00

INSERT
4.00
4.00
3.00
2.00
1.00
4.00

REMOVE
3.00
3.00
```

Pseudo/tests/end_to_end/pass/test-obj-in-obj.in

```
def main():
        a.name = "hello"
        a.number = 2
        b.obj = a
        print b.obj.number
```

Pseudo/tests/end_to_end/pass/test-obj-in-obj.out

```
2.00
```

Pseudo/tests/end_to_end/pass/test-obj-return.in

```
def main():
        a.name = "kevin"
        c = func(a)
        print c.name
        print c.age
        print a.name

def func(b):
        b.age = 17
        b.name = "ben"
        return b
```

Pseudo/tests/end_to_end/pass/test-obj-return.out

```
ben
17.00
ben
```

Pseudo/tests/end_to_end/pass/test-obj_init.in

```
def main():
        init a, b, c
        a.age = 21
        a.name = "Raymond"
        b.alive = true
        print a
        print b
        print c
```

Pseudo/tests/end_to_end/pass/test-obj_init.out

```
a.age: 21.00
a.alive: 0
a.name: Raymond
b.age: 0.00
b.alive: 1
b.name:
c.age: 0.00
c.alive: 0
c.name:
```

Pseudo/tests/end_to_end/pass/test-obj_print.in

```
def main():
        a.name = "Ben"
        a.age = 23
        a.is_student = true
        print a

        b.num = 73.732
        print b

        c = a
        c.name = "Kevin"
        print c
```

Pseudo/tests/end_to_end/pass/test-obj_print.out

```
a.age: 23.00
a.is_student: 1
a.name: Ben
b.num: 73.73
c.age: 23.00
c.is_student: 1
c.name: Kevin
```

Pseudo/tests/end_to_end/pass/test-objects.in

```
def main():
        a.name = "kevin"
        func(a)

def func(b):
        b.age = 17
        print b.name
        print b.age
```

Pseudo/tests/end_to_end/pass/test-objects.out

```
kevin
17.00
```

Pseudo/tests/end_to_end/pass/test-print_string.in

```
def main():
        print "string"
```

Pseudo/tests/end_to_end/pass/test-print_string.out

```
string
```

Pseudo/tests/end_to_end/pass/test-quicksort.in

```
def main():
        arr = [5, 6, 2, 1, 4, 7]
        quicksort(arr, 0, arr.length() - 1)
        print arr

def quicksort(a, start, end):
        if start >= end:
                return a
        pivot = start
        new_pivot = partition(a, start, end, pivot)
        quicksort(a, start, new_pivot - 1)
        quicksort(a, new_pivot + 1, end)

def partition(a, start, end, pivot):
        swap(a, pivot, end)
        ret = start

        for i = start to end:
                if a.get(i) < a.get(end):
                        swap(a, i, ret)
                        ret = ret + 1

        swap(a, ret, end)
        return ret

def swap(a, i, j):
        temp = a.get(i)
        a.set(i, a.get(j))
        a.set(j, temp)
```

Pseudo/tests/end_to_end/pass/test-quicksort.out

```
[1.00, 2.00, 4.00, 5.00, 6.00, 7.00]
```

Pseudo/tests/end_to_end/pass/test-scope.in

```
def main():
        a = 3
        if a > 0:
                a = a - 7
                print a
                if a < 0:
                        a = a + 4
                        print a
                print a
        print a
```

Pseudo/tests/end_to_end/pass/test-scope.out

```
-4.00
0.00
0.00
0.00
```

Pseudo/tests/end_to_end/pass/test-selectionsort.in

```
def main():
        list = [5, 3, 7, 1, 2, 6]

        for i = 0 to list.length() - 1:
                min = 99999
                min_index = i + 1
                for j = i to list.length():
                        if list.get(j) < min:
                                min = list.get(j)
                                min_index = j
                temp = list.get(i)
                list.set(i, list.get(min_index))
                list.set(min_index, temp)

        print list
```

Pseudo/tests/end_to_end/pass/test-selectionsort.out

```
[1.00,2.00,3.00,5.00,6.00,7.00]
```

Pseudo/tests/end_to_end/pass/test-unop.in

```
def main():
        x = 3 - 7
        print x
        print x == -4
        print !(x == -4)
```

Pseudo/tests/end_to_end/pass/test-unop.out

```
-4.00
1
0
```

Pseudo/tests/inference/_assign.in

```
def main()~
    a = 1;
    b = 2 + 3;
    c = true or  false ;
    d = a + b;
    e = "hello";
    f = "  world";
    g = e ^ f;
    h = c and false;
    i = 1 + 2 + 3 + a;
    j = k = 1;
    l = -1;
    a = 2;
    return 1;
$
```

Pseudo/tests/inference/_assign.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"ANumLit","data_type
    ":"Num","val":"1."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"ABinop","e1
    ":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"op":"Add","e2":{"aexpr":"ANumLit","data_type":"Num","val
    ":"3."},"val":"Num"},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"ABinop","e1
    ":{"aexpr":"ABoolLit","data_type":"Bool","val":"true"},"op":"Or","e2":{"aexpr":"ABoolLit","data_type":"Bool","val":"false
    "},"val":"Bool"},"data_type":"Bool"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"d","e":{"aexpr":"ABinop","e1":{"
    aexpr":"AVal","val_name":"a","data_type":"Num"},"op":"Add","e2":{"aexpr":"AVal","val_name":"b","data_type":"Num"},"
    val":"Num"},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"e","e":{"aexpr":"AStringLit","data_type
    ":"String","val":"hello"},"data_type":"String"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"f","e":{"aexpr":"
    AStringLit","data_type":"String","val":"world"},"data_type":"String"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"g
    ","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"e","data_type":"String"},"op":"Concat","e2":{"aexpr":"AVal","
    val_name":"f","data_type":"String"},"val":"String"},"data_type":"String"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"
    h","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"c","data_type":"Bool"},"op":"And","e2":{"aexpr":"ABoolLit","
    data_type":"Bool","val":"false"},"val":"Bool"},"data_type":"Bool"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"i","e
    ":{"aexpr":"ABinop","e1":{"aexpr":"ABinop","e1":{"aexpr":"ABinop","e1":{"aexpr":"ANumLit","data_type":"Num","val
    ":"1."},"op":"Add","e2":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"val":"Num"},"op":"Add","e2":{"aexpr":"
    ANumLit","data_type":"Num","val":"3."},"val":"Num"},"op":"Add","e2":{"aexpr":"AVal","val_name":"a","data_type":"Num
    "},"val":"Num"},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"j","e":{"aexpr":"AAssign","var":"k
    ","e":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"data_type":"Num"},"data_type":"Num"}},{"astmt":"AExpr","e
    ":{"aexpr":"AAssign","var":"l","e":{"aexpr":"AUnop","e":"Neg","op":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"
    val":"Num"},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"ANumLit","data_type
    ":"Num","val":"2."},"data_type":"Num"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"1."}}]}]
```


Pseudo/tests/inference/_call.in

```
def main()~
    a = 1;
    b = 2;
    c = get_str();
    d = one(a, b) + two(a, b);
    return sum(a, b);
$

def one(n1, n2)~
    return 1;
$

def two(n1, n2)~
    return 2;
$

def get_str()~
```

```
    return "hello";
$

def sum(n1, n2)~
    return n1 + n2;
$
```

Pseudo/tests/inference/_call.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"AACall","name":"get_str","args":[],"data_type":"String"},"data_type":"String"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"d","e":{"aexpr":"ABinop","e1":{"aexpr":"AACall","name":"one","args":[{"aexpr":"AVal","val_name":"a","data_type":"Num"},{"aexpr":"AVal","val_name":"b","data_type":"Num"}],"data_type":"Num"},"op":"Add","e2":{"aexpr":"AACall","name":"two","args":[{"aexpr":"AVal","val_name":"a","data_type":"Num"},{"aexpr":"AVal","val_name":"b","data_type":"Num"}],"data_type":"Num"},"val":"Num"},"data_type":"Num"}},{"astmt":"AReturn","e":{"aexpr":"AACall","name":"sum","args":[{"aexpr":"AVal","val_name":"a","data_type":"Num"},{"aexpr":"AVal","val_name":"b","data_type":"Num"}],"data_type":"Num"}}]},{"funcname":"get_str","params":[],"body":[{"astmt":"AReturn","e":{"aexpr":"AStringLit","data_type":"String","val":"hello"}}]},{"funcname":"one","params":[{"Num":"n1"},{"Num":"n2"}],"body":[{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"1."}}]},{"funcname":"two","params":[{"Num":"n1"},{"Num":"n2"}],"body":[{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"2."}}]},{"funcname":"sum","params":[{"Num":"n1"},{"Num":"n2"}],"body":[{"astmt":"AReturn","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"n1","data_type":"Num"},"op":"Add","e2":{"aexpr":"AVal","val_name":"n2","data_type":"Num"},"val":"Num"}}]}]
```


Pseudo/tests/inference/_combine.in

```
def main()~
    a = [1, 2, 3];
    b = [4, 5];
    c = a :: b;
    return 0;
$
```

Pseudo/tests/inference/_combine.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."},{"aexpr":"ANumLit","data_type":"Num","val":"3."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"4."},{"aexpr":"ANumLit","data_type":"Num","val":"5."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Num)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"b","data_type":"List(Num)"},"val":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]
```


Pseudo/tests/inference/_combine_chain.in

```
def main()~
    a = [1, 2, 3];
    b = [4, 5, 6];
    c = [7, 8, 9];
    d = a :: b;
    e = a :: b :: c;
    return 0;
$
```

Pseudo/tests/inference/_combine_chain.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."},{"aexpr":"ANumLit","data_type":"Num","val":"3."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"4."},{"aexpr":"ANumLit","data_type":"Num","val":"5."},{"aexpr":"ANumLit","data_type":"Num","val":"6."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"7."},{"aexpr":"ANumLit","data_type":"Num","val":"8."},{"aexpr":"ANumLit","data_type":"Num","val":"9."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"d","e":{"
```

aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Num)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"b","data_type":"List(Num)"},"val":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"e","e":{"aexpr":"ABinop","e1":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Num)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"b","data_type":"List(Num)"},"val":"List(Num)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"c","data_type":"List(Num)"},"val":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]}

Pseudo/tests/inference/_emb_undetermined_list.in

```
def main()~
    a = [[],   []];
    a.get(0).push(1);
    b = [[],   []];
    b.get(0).insert(0, 2);
    c = [[[],   []],   [[]]];
    c.get(1).get(0).push("test");
    d = a :: b;
    return 0;
$
```

Pseudo/tests/inference/_emb_undetermined_list.out

[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AListPush","e1":{"aexpr":"AListRemove","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Notprim)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"0."},"data_type":"List(Notprim)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AListInsert","e1":{"aexpr":"AListRemove","e1":{"aexpr":"AVal","val_name":"b","data_type":"List(Notprim)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"0."},"e3":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AListPush","e1":{"aexpr":"AListRemove","e1":{"aexpr":"AVal","val_name":"c","data_type":"List(Notprim)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"data_type":"List(Notprim)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"0."},"data_type":"List(Notprim)"},"e2":{"aexpr":"AStringLit","data_type":"String","val":"test"},"data_type":"String"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"d","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Notprim)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"b","data_type":"List(Notprim)"},"val":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]

Pseudo/tests/inference/_list_of_empty_lists.in

```
def main()~
    a = [[],   []];
    b = [[1,  2],  [3,  4]];
    c = a :: b;
    d = [1,  2];
    e = [[[],   []],   [[],   [],   []]];
    f = [[[1,  2],  [3,  4]],   [[5]],   [[2 + 2 + 2]]];
    g = e :: f;
    h = [[],   [1,  2]];
    i = b :: h;
    return 0;
$
```

Pseudo/tests/inference/_list_of_empty_lists.out

[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","

data_type":"Num","val":"2."}],"data_type":"List(Num)"},{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"3."},{"aexpr":"ANumLit","data_type":"Num","val":"4."}],"data_type":"List(Num)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Notprim)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"b","data_type":"List(Notprim)"},"val":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"d","e":{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"e","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"f","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."}],"data_type":"List(Num)"},{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"3."},{"aexpr":"ANumLit","data_type":"Num","val":"4."}],"data_type":"List(Num)"}],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"5."}],"data_type":"List(Num)"}],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"ABinop","e1":{"aexpr":"ABinop","e1":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"op":"Add","e2":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"val":"Num"},"op":"Add","e2":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"val":"Num"}],"data_type":"List(Num)"}],"data_type":"List(Notprim)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"g","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"e","data_type":"List(Notprim)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"f","data_type":"List(Notprim)"},"val":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"h","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[],"data_type":"List(Notprim)"},{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."}],"data_type":"List(Num)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"i","e":{"aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"h","data_type":"List(Notprim)"},"op":"Combine","e2":{"aexpr":"AVal","val_name":"b","data_type":"List(Notprim)"},"val":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]

Pseudo/tests/inference/_list_of_list.in

```
def main()~
    a = [[1,  2],  [3,  4],  [5,  6]];
    return 0;
$
```

Pseudo/tests/inference/_list_of_list.out

[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."}],"data_type":"List(Num)"},{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"3."},{"aexpr":"ANumLit","data_type":"Num","val":"4."}],"data_type":"List(Num)"},{"aexpr":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"5."},{"aexpr":"ANumLit","data_type":"Num","val":"6."}],"data_type":"List(Num)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]

Pseudo/tests/inference/_list_ops.in

```
def main()~
    a = [1,  2,  3];
    b = ["a",  "b",  "c"];

    a.insert(3,  4);
    b.push("d");

    a.remove(2);
    a.pop();
    b.dequeue();

    c = a.length();

    a.get(0);
    a.set(0,  5);
    b.set(0,  "e");

    return 0;
$
```

Pseudo/tests/inference/_list_ops.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[{"
    aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val":"2."},{"aexpr":"ANumLit
    ","data_type":"Num","val":"3."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt":"AExpr","e":{"aexpr":"
    AAssign","var":"b","e":{"aexpr":"AListDecl","val":[{"aexpr":"AStringLit","data_type":"String","val":"a"},{"aexpr":"
    AStringLit","data_type":"String","val":"b"},{"aexpr":"AStringLit","data_type":"String","val":"c"}],"data_type":"List(String)
    "},"data_type":"List(String)"}},{"astmt":"AExpr","e":{"aexpr":"AListInsert","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"List(Num)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"3."},"e3":{"aexpr":"ANumLit","data_type":"
    Num","val":"4."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AListPush","e1":{"aexpr":"AVal","val_name":"b","
    data_type":"List(String)"},"e2":{"aexpr":"AStringLit","data_type":"String","val":"d"},"data_type":"String"}},{"astmt":"
    AExpr","e":{"aexpr":"AListRemove","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Num)"},"e2":{"aexpr":"
    ANumLit","data_type":"Num","val":"2."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AListPop","e":{"aexpr":"
    AVal","val_name":"a","data_type":"List(Num)"},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AListDequeue","e":{"
    aexpr":"AVal","val_name":"b","data_type":"List(String)"},"data_type":"String"}},{"astmt":"AExpr","e":{"aexpr":"AAssign
    ","var":"c","e":{"aexpr":"AListLength","e":{"aexpr":"AVal","val_name":"a","data_type":"List(Num)"},"data_type":"Num"},"
    data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AListRemove","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(
    Num)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"0."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"
    AListSet","e1":{"aexpr":"AVal","val_name":"a","data_type":"List(Num)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val
    ":"0."},"e3":{"aexpr":"ANumLit","data_type":"Num","val":"5."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"
    AListSet","e1":{"aexpr":"AVal","val_name":"b","data_type":"List(String)"},"e2":{"aexpr":"ANumLit","data_type":"Num","
    val":"0."},"e3":{"aexpr":"AStringLit","data_type":"String","val":"e"},"data_type":"String"}},{"astmt":"AReturn","e":{"
    aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]
```


Pseudo/tests/inference/_listdecl.in

```
def main()~
a =[];
b = [1, 2, 3];
c = ["a","b","c"];
d = c;
e = 1;
f = 2;
g = [e, f];
$
```

Pseudo/tests/inference/_listdecl.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"a","e":{"aexpr":"AListDecl","val":[],"
    data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr
    ":"AListDecl","val":[{"aexpr":"ANumLit","data_type":"Num","val":"1."},{"aexpr":"ANumLit","data_type":"Num","val
    ":"2."},{"aexpr":"ANumLit","data_type":"Num","val":"3."}],"data_type":"List(Num)"},"data_type":"List(Num)"}},{"astmt
    ":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"AListDecl","val":[{"aexpr":"AStringLit","data_type":"String","val
    ":"a"},{"aexpr":"AStringLit","data_type":"String","val":"b"},{"aexpr":"AStringLit","data_type":"String","val":"c"}],"
    data_type":"List(String)"},"data_type":"List(String)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"d","e":{"aexpr":"
    AVal","val_name":"c","data_type":"List(String)"},"data_type":"List(String)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","
    var":"e","e":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"
    AAssign","var":"f","e":{"aexpr":"ANumLit","data_type":"Num","val":"2."},"data_type":"Num"}},{"astmt":"AExpr","e":{"
    aexpr":"AAssign","var":"g","e":{"aexpr":"AListDecl","val":[{"aexpr":"AVal","val_name":"e","data_type":"Num"},{"aexpr":"
    AVal","val_name":"f","data_type":"Num"}],"data_type":"List(Num)"},"data_type":"List(Num)"}}]}]
```


Pseudo/tests/inference/_obj_assign.in

```
def main()~
    a.name = "Foo";
    b.age = 22;
    a = b;
$
```

Pseudo/tests/inference/_obj_assign.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Foo"},"data_type":"String"}},{"
    astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"b","data_type":"Object(0)"},"s":"age","e2
    ":{"aexpr":"ANumLit","data_type":"Num","val":"22."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var
    ":"a","e":{"aexpr":"AVal","val_name":"b","data_type":"Object(0)"},"data_type":"Object(0)"}}]}]
```


Pseudo/tests/inference/_obj_dict.out

[Fdecl({name=main;params=[];body=[AObjectAssign(AVal(person1,Object(0)),name,AString_lit(String,Raymond),String);
AObjectAssign(AVal(person2,Object(0)),age,ANum_lit(Num,42),Num);AAssign(my_dict,ADictDecl([],[],Dict(Undetermined,
Undetermined)),Dict(Undetermined,Undetermined));ADictMap(AVal(my_dict,Object(0)),ANum_lit(Num,1),AVal(person1,
Object(0)),Object(0));ADictMap(AVal(my_dict,Object(0)),ANum_lit(Num,2),AVal(person2,Object(0)),Object(0));AAssign(
person,ADictFind(AVal(my_dict,Object(0)),ANum_lit(Num,2),Object(−1)),Object(−1));Return(ANum_lit(Num,0))]})]

Pseudo/tests/inference/_obj_equality.in

```
def main()~
        a.name = "Raymond";
        b.age = 21;
        a == b;
$
```

Pseudo/tests/inference/_obj_equality.out

[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Raymond"},"data_type":"String
"}},{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"b","data_type":"Object(0)"},"s":"age
","e2":{"aexpr":"ANumLit","data_type":"Num","val":"21."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"ABinop
","e1":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"op":"Eq","e2":{"aexpr":"AVal","val_name":"b","data_type
":"Object(0)"},"val":"Bool"}}]}]

Pseudo/tests/inference/_obj_field_list_of_objs.in

```
def main()~
        b.visited = false;
        a.neighbors = [b];
$
```

Pseudo/tests/inference/_obj_field_list_of_objs.out

[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"b","
data_type":"Object(1)"},"s":"visited","e2":{"aexpr":"ABoolLit","data_type":"Bool","val":"false"},"data_type":"Bool"}},{"
astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"s":"neighbors
","e2":{"aexpr":"AListDecl","val":[{"aexpr":"AVal","val_name":"b","data_type":"Object(1)"}],"data_type":"List(Notprim)"},"
data_type":"List(Notprim)"}}]}]

Pseudo/tests/inference/_obj_fields.in

```
def main()~
    a.name = "John";
    a.age = 25;
    a.name = a.name ^ " Doe";
    $
```

Pseudo/tests/inference/_obj_fields.out

[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"John"},"data_type":"String"}},{"
astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"s":"age","e2
":{"aexpr":"ANumLit","data_type":"Num","val":"25."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"
AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"s":"name","e2":{"aexpr":"ABinop","e1":{"
aexpr":"AObjectField","e":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"s":"name","data_type":"String"},"op
":"Concat","e2":{"aexpr":"AStringLit","data_type":"String","val":"Doe"},"val":"String"},"data_type":"String"}}]}]

Pseudo/tests/inference/_obj_func.in

```
def main()~
    person1.name = "Raymond";
        foo(person1);
        dog1.age = 5;
        return 0;
```

```
$

def foo(x)~
        person2.alive = true;
        person2 = x;
        return 0;
$
```

Pseudo/tests/inference/_obj_func.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"
    person1","data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Raymond"},"data_type
    ":"String"}},{"astmt":"AExpr","e":{"aexpr":"AACall","name":"foo","args":[{"aexpr":"AVal","val_name":"person1","
    data_type":"Object(0)"}],"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","
    val_name":"dog1","data_type":"Object(1)"},"s":"age","e2":{"aexpr":"ANumLit","data_type":"Num","val":"5."},"data_type":"
    Num"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]},{"funcname":"foo","params":[{"
    Object(0)":"x"}],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"person2","
    data_type":"Object(0)"},"s":"alive","e2":{"aexpr":"ABoolLit","data_type":"Bool","val":"true"},"data_type":"Bool"}},{"astmt
    ":"AExpr","e":{"aexpr":"AAssign","var":"person2","e":{"aexpr":"AVal","val_name":"x","data_type":"Object(0)"},"data_type
    ":"Object(0)"}},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val":"0."}}]}]
```


Pseudo/tests/inference/_obj_in_obj.in

```
def main()~
        a.name = "Raymond";
        b.obj = a;
        print b.obj.name;
$
```

Pseudo/tests/inference/_obj_in_obj.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Raymond"},"data_type":"String
    "}},{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"b","data_type":"Object(1)"},"s":"obj
    ","e2":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"data_type":"Object(0)"}},{"astmt":"APrint","e":{"aexpr":"
    AObjectField","e":{"aexpr":"AObjectField","e":{"aexpr":"AVal","val_name":"b","data_type":"Object(1)"},"s":"obj","
    data_type":"Object(0)"},"s":"name","data_type":"String"}}]}]
```


Pseudo/tests/inference/_obj_inequality.in

```
def main()~
        a.name = "Raymond";
        b.age = 21;
$
```

Pseudo/tests/inference/_obj_inequality.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Raymond"},"data_type":"String
    "}},{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"b","data_type":"Object(1)"},"s":"age
    ","e2":{"aexpr":"ANumLit","data_type":"Num","val":"21."},"data_type":"Num"}}]}]
```


Pseudo/tests/inference/_obj_init.in

```
def main()~
    init a;
    init x, y, z;
$
```

Pseudo/tests/inference/_obj_init.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AObjectInit","objlist":[(a,"Object(0)")]},{"astmt":"AObjectInit","objlist":[(x,"
    Object(1)"),(y,"Object(1)"),(z,"Object(1)")]}]}]
```

Pseudo/tests/inference/_obj_listdecl.in

```
def main()~
    a. field  = "test";
    b = [a];
    c = b.pop();
    print  c. field ;
$
```

Pseudo/tests/inference/_obj_listdecl.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"},"s":"field","e2":{"aexpr":"AStringLit","data_type":"String","val":"test"},"data_type":"String"}},{"
    astmt":"AExpr","e":{"aexpr":"AAssign","var":"b","e":{"aexpr":"AListDecl","val":[{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"}],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"
    AAssign","var":"c","e":{"aexpr":"AListPop","e":{"aexpr":"AVal","val_name":"b","data_type":"List(Notprim)"},"data_type":"
    Object(0)"},"data_type":"Object(0)"}},{"astmt":"APrint","e":{"aexpr":"AObjectField","e":{"aexpr":"AVal","val_name":"c","
    data_type":"Object(0)"},"s":"field","data_type":"String"}}]}]
```


Pseudo/tests/inference/_obj_lists.in

```
def main()~
        roster  =  [];
        person.age = 21;
        roster . insert (1,  person);
        person2 = roster.pop();
$
```

Pseudo/tests/inference/_obj_lists.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"roster","e":{"aexpr":"AListDecl","val
    ":[],"data_type":"List(Notprim)"},"data_type":"List(Notprim)"}},{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"
    aexpr":"AVal","val_name":"person","data_type":"Object(0)"},"s":"age","e2":{"aexpr":"ANumLit","data_type":"Num","val
    ":"21."},"data_type":"Num"}},{"astmt":"AExpr","e":{"aexpr":"AListInsert","e1":{"aexpr":"AVal","val_name":"roster","
    data_type":"List(Notprim)"},"e2":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"e3":{"aexpr":"AVal","val_name":"
    person","data_type":"Object(0)"},"data_type":"Object(0)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"person2","e
    ":{"aexpr":"AListPop","e":{"aexpr":"AVal","val_name":"roster","data_type":"List(Notprim)"},"data_type":"Object(0)"},"
    data_type":"Object(0)"}}]}]
```


Pseudo/tests/inference/_obj_nesting.in

```
def main()~
        person1.age = 7;
        person2.name = "Raymond";
        if  (person1 == person2)~
                person3 = person2;
                person4 = person3;
                person5 = person1;
        $
        return  0;
$
```

Pseudo/tests/inference/_obj_nesting.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"
    person1","data_type":"Object(0)"},"s":"age","e2":{"aexpr":"ANumLit","data_type":"Num","val":"7."},"data_type":"Num
    "}},{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"person2","data_type":"Object(0)"},"s
    ":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Raymond"},"data_type":"String"}},{"aexpr":"AFIf","e1":{"
    aexpr":"ABinop","e1":{"aexpr":"AVal","val_name":"person1","data_type":"Object(0)"},"op":"Eq","e2":{"aexpr":"AVal","
    val_name":"person2","data_type":"Object(0)"},"val":"Bool"},"s1":[{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"person3
    ","e":{"aexpr":"AVal","val_name":"person2","data_type":"Object(0)"},"data_type":"Object(0)"}},{"astmt":"AExpr","e":{"
    aexpr":"AAssign","var":"person4","e":{"aexpr":"AVal","val_name":"person3","data_type":"Object(0)"},"data_type":"Object
    (0)"}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"person5","e":{"aexpr":"AVal","val_name":"person1","data_type":"
    Object(0)"},"data_type":"Object(0)"}}],"s2":[]},{"astmt":"AReturn","e":{"aexpr":"ANumLit","data_type":"Num","val
    ":"0."}}]}]
```

Pseudo/tests/inference/_obj_return.in

```
def main()~
        a.name = "Kevin";
        c = func(a);
        print c.age;
$

def func(b)~
        b.age = 4;
        return b;
$
```

Pseudo/tests/inference/_obj_return.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"},"s":"name","e2":{"aexpr":"AStringLit","data_type":"String","val":"Kevin"},"data_type":"String
    "}},{"astmt":"AExpr","e":{"aexpr":"AAssign","var":"c","e":{"aexpr":"AACall","name":"func","args":[{"aexpr":"AVal","
    val_name":"a","data_type":"Object(0)"}],"data_type":"Object(0)"},"data_type":"Object(0)"}},{"astmt":"APrint","e":{"aexpr
    ":"AObjectField","e":{"aexpr":"AVal","val_name":"c","data_type":"Object(0)"},"s":"age","data_type":"Num"}}]},{"funcname
    ":"func","params":[{"Object(0)":"b"}],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","
    val_name":"b","data_type":"Object(0)"},"s":"age","e2":{"aexpr":"ANumLit","data_type":"Num","val":"4."},"data_type":"
    Num"}},{"astmt":"AReturn","e":{"aexpr":"AVal","val_name":"b","data_type":"Object(0)"}}]}]
```


Pseudo/tests/inference/_obj_update.in

```
def main()~
    a.age = 5;
    a.age = a.age + 1;
$
```

Pseudo/tests/inference/_obj_update.out


```
[{"funcname":"main","params":[],"body":[{"astmt":"AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","
    data_type":"Object(0)"},"s":"age","e2":{"aexpr":"ANumLit","data_type":"Num","val":"5."},"data_type":"Num"}},{"astmt":"
    AExpr","e":{"aexpr":"AObjectAssign","e1":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"s":"age","e2":{"aexpr
    ":"ABinop","e1":{"aexpr":"AObjectField","e":{"aexpr":"AVal","val_name":"a","data_type":"Object(0)"},"s":"age","data_type
    ":"Num"},"op":"Add","e2":{"aexpr":"ANumLit","data_type":"Num","val":"1."},"val":"Num"},"data_type":"Num"}}]}]
```


Pseudo/tests/inference/_undetermined_list.in

```
def main()~
    a = [];
    a.insert(0, 1);
    a.push(2);
    a.dequeue();
    return 0;
$
```

Pseudo/tests/inference/_undetermined_list.out


```
[
        {
                "funcname":"main",

                "params":[],
                "body":[
                        {"astmt":"AExpr",
                        "e":{"aexpr":"AAssign",
                        "var":"a",
                        "e":{"aexpr":"AListDecl",
                        "val ":[],
                        "data_type":"List(Notprim)"},
                        "data_type":"List(Notprim)"}},
                        {"astmt":"AExpr",
```

```
                        "e":{"aexpr":"AListInsert",
                        "e1":{"aexpr":"AVal",
                        "val_name":"a",
                        "data_type":"List(Notprim)"},
                        "e2":{"aexpr":"ANumLit",
                        "data_type":"Num",
                        "val":"0."},
                        "e3":{"aexpr":"ANumLit",
                        "data_type":"Num",
                        "val":"1."},"data_type":"Num"}},
                        {"astmt":"AExpr",
                        "e":{"aexpr":"AListPush",
                        "e1":{"aexpr":"AVal",
                        "val_name":"a",
                        "data_type":"List(Num)"},
                        "e2":{"aexpr":"ANumLit",
                        "data_type":"Num",
                        "val":"2."},
                        "data_type":"Num"}},
                        {"astmt":"AExpr",
                        "e":{"aexpr":"AListDequeue",
                        "e":{"aexpr":"AVal",
                        "val_name":"a",
                        "data_type":"List(Num)"},
                        "data_type":"Num"}},
                        {"astmt":"AReturn",
                        "e":{"aexpr":"ANumLit",
                        "data_type":"Num",
                        "val":"0."}}
                        ]
        }
]
```


Pseudo/tests/inference/_while.in

```
def main()~
    a = 1;
    while a == 1~
        b = 2;
    $
    return a;
$
```

Pseudo/tests/inference/_while.out


```
[
  {
    "funcname": "main",
    "params": [

    ],
    "body": [
      {
        "astmt": "AExpr",
        "e": {
          "aexpr": "AAssign",
          "var": "a",
          "e": {
            "aexpr": "ANumLit",
            "data_type": "Num",
            "val": "1."
          },
          "data_type": "Num"
        }
      },
      {
        "aexpr": "AWhile",
        "e": {
          "aexpr": "ABinop",
          "e1": {
            "aexpr": "AVal",
            "val_name": "a",
            "data_type": "Num"
          },
          "op": "Eq",
          "e2": {
            "aexpr": "ANumLit",
            "data_type": "Num",
            "val": "1."
          },
          "val": "Bool"
        },
```

```
      "s": [
        {
          "astmt": "AExpr",
          "e": {
            "aexpr": "AAssign",
            "var": "b",
            "e": {
              "aexpr": "ANumLit",
              "data_type": "Num",
              "val": "2."
            },
            "data_type": "Num"
          }
        }
      ]
    },
    {
      "astmt": "AReturn",
      "e": {
        "aexpr": "AVal",
        "val_name": "a",
        "data_type": "Num"
      }
    }
  ]
 }
]
```


Pseudo/tests/inference/fail/_fail_for_in.in

```
def main()~
    a = 1;
    b = 2;
    for a in b~
        "test";
    $
    return 0;
$
```

Pseudo/tests/inference/fail/_fail_for_range.in

```
def main()~
        a = 1;
    for 1 to 10~
        a = a + 1;
    $
    return a;
$
```

Pseudo/tests/inference/fail/_fail_if_else.in

```
def main()~
        a = 1;
        if true~
        b = 1;
                if false~
            b = 2;
                        return "b";
                $elsif true~
            b = 3;
                        return "c";
                $else~
            b = 4;
```

```
                    return "d";
                $
        $else~
        b = 5;
                return "d";
        $
        return "0";
$
```

Pseudo/tests/inference/fail/_reassign.in

```
def main()~
    a = 1;
    a = 2;
    a = "hi";
$
```

Pseudo/tests/parser/_combine.in

```
def main()~
    a = [1, 2, 3];
    b = [4, 5];
    c = a :: b;
    return 0;
$
```

Pseudo/tests/parser/_combine.out

```
[Fdecl({params=[];
    body=[Assign(a,ListDecl([Num_lit(1.);
    Num_lit(2.);Num_lit(3.)]));
    Assign(b,ListDecl([Num_lit(4.);
    Num_lit(5.)]));
    Assign(c,Binop(Id(a),Combine,Id(b)));
    Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/_combine_chain.in

```
def main()~
    a = [1, 2, 3];
    b = [4, 5, 6];
    c = [7, 8, 9];
    d = a :: b;
    e = a :: b :: c;
    return 0;
$
```

Pseudo/tests/parser/_combine_chain.out

```
[Fdecl({params=[];
body=[Assign(a,ListDecl([Num_lit(1.);Num_lit(2.);Num_lit(3.)]));
Assign(b,ListDecl([Num_lit(4.);Num_lit(5.);Num_lit(6.)]));
Assign(c,ListDecl([Num_lit(7.);Num_lit(8.);Num_lit(9.)]));
Assign(d,Binop(Id(a),Combine,Id(b)));
Assign(e,Binop(Binop(Id(a),Combine,Id(b)),Combine,Id(c)));
Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/_comments.in

```
def MAIN()~
if "// edge case"~
print "cool //";
$$
```

Pseudo/tests/parser/_comments.out

```
[Fdecl({params=[];body=[If(String_lit(//edgecase),[Print([ String_lit (cool//)]) ],[]) ]})]
```

Pseudo/tests/parser/_dict_decl.in

```
def main()~
    a = {};
    b = {"one": 1, "two": 2};
    c = {"test":  []};
    d = {"test":  [[1]]};
    return 0;
$
```

Pseudo/tests/parser/_dict_decl.out

```
[Fdecl({params=[];body=[Assign(a,DictDecl([]));Assign(b,DictDecl([(String_lit(one),Num_lit(1.));( String_lit (two),Num_lit(2.))]));
    Assign(c,DictDecl([( String_lit (test),ListDecl ([]) )]));Assign(d,DictDecl([( String_lit (test),ListDecl([ListDecl([Num_lit(1.)])
    ])])));Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/_dict_of_lists.in

```
def main()~
    a = {1:  []};
    a.find(1).push(0);
$
```

Pseudo/tests/parser/_dict_of_lists.out

```
[Fdecl({params=[];body=[Assign(a,DictDecl([(Num_lit(1.),ListDecl([]))]));ListPush(DictFind(Id(a),Num_lit(1.)),Num_lit(0.))]})]
```

Pseudo/tests/parser/_dict_ops.in

```
def main()~
    a = {"one": 1, "two": 2};
    b = a.mem("one");
    c = a.delete("one");
    d = c or true;
    e = a.size() + 5;
    f = a.find("two") + 1;
    g = a.map("three", 3);
    h = g and true;
$
```

Pseudo/tests/parser/_dict_ops.out

```
[Fdecl({params=[];

    body=[Assign(a,DictDecl([(String_lit(one),Num_lit(1.));( String_lit (two),Num_lit(2.))]));
    Assign(b,DictMem(Id(a),String_lit(one)));
    Assign(c,DictDelete(Id(a), String_lit (one)));
    Assign(d,Binop(Id(c),Or,Bool_lit(true)));
    Assign(e,Binop(DictSize(Id(a)),Add,Num_lit(5.)));
    Assign(f,Binop(DictFind(Id(a),String_lit(two)),Add,Num_lit(1.)));
    Assign(g,DictMap(Id(a),String_lit(three),Num_lit(3.)));
    Assign(h,Binop(Id(g),And,Bool_lit(true)))]})]
```

Pseudo/tests/parser/_emb_undetermined_list.in

```
def main()~
    a = [[],   []];
    a.get(0).push(1);
    b = [[],   []];
    b.get(0).insert(0, 2);
    c = [[[],   []],   [[]]];
    c.get(1).get(0).push("test");
    d = a :: b;
    return 0;
$
```

Pseudo/tests/parser/_emb_undetermined_list.out

```
[Fdecl({params=[];
    body=[Assign(a,ListDecl([ListDecl([]);ListDecl([])]));
    ListPush(ListGet(Id(a),Num_lit(0.)),Num_lit(1.));
    Assign(b,ListDecl([ListDecl([]);
    ListDecl([])]));
    ListInsert(ListGet(Id(b),Num_lit(0.)),Num_lit(0.),Num_lit(2.));
    Assign(c,ListDecl([ListDecl([]);
    ListDecl([])]);
    ListDecl([ListDecl([])])]));
    ListPush(ListGet(ListGet(Id(c),Num_lit(1.)),Num_lit(0.)), String_lit(test));
    Assign(d,Binop(Id(a),Combine,Id(b)));
    Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/_empty_list.in

```
def main()~
    a = [];
    b = [1, 2, 3];
    c = a :: b;
    return 0;
$
```

Pseudo/tests/parser/_empty_list.out

```
[Fdecl({params=[];body=[Assign(a,ListDecl([]));Assign(b,ListDecl([Num_lit(1.);Num_lit(2.);Num_lit(3.)]));Assign(c,Binop(Id(a),
    Combine,Id(b)));Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/_eof.in

```
def HELLO()~
if True~
if True~
if True~
print "eof";
$$$
$
```

Pseudo/tests/parser/_eof.out

```
[Fdecl({params=[];body=[If(Id(True),[If(Id(True),[If(Id(True),[Print([String_lit(eof)]) ],[]) ],[]) ],[]) ]})]
```

Pseudo/tests/parser/_fdecl.in

```
def HELLO()~
print "hello world";
$

def MAIN()~
  HELLO();
$
```

Pseudo/tests/parser/_fdecl.out

```
[Fdecl({params=[];body=[Print([String_lit(helloworld)])]});Fdecl({params=[];body=[Call(HELLO,[])]})]
```

Pseudo/tests/parser/_for_range.in

```
def main()~
    for i = 0 to 5~
        if i == 3~
            return;
        $
    $
    return;
$
```

Pseudo/tests/parser/_for_range.out

```
[Fdecl({params=[];body=[ForRange(Assign(i,Num_lit(0.)),Num_lit(5.),Num_lit(1.),[If(Binop(Id(i),Eq,Num_lit(3.)),[Return(Noexpr)
    ],[])]);Return(Noexpr)]})]
```

Pseudo/tests/parser/_hello.in

```
def HELLO()~
    print "hello world";
$
```

Pseudo/tests/parser/_hello.out

```
[Fdecl({params=[];body=[Print([String_lit(helloworld)])]})]
```

Pseudo/tests/parser/_list_of_dicts.in

```
def main()~
    a = [{}, {}];
    a.get(0).map("one", 1);
    a.get(0).delete("one");

    b = [[{}, {}], [{}]];
    b.get(0).get(1).map("one", 1);
    b.get(0).get(1).delete("one");

    c = b.get(0) :: a;
    return c;
$
```

Pseudo/tests/parser/_list_of_dicts.out

[Fdecl({params=[];body=[Assign(a,ListDecl([DictDecl([]);DictDecl([])]));DictMap(ListGet(Id(a),Num_lit(0.)),String_lit(one),Num_lit(1.));DictDelete(ListGet(Id(a),Num_lit(0.)), String_lit (one));Assign(b,ListDecl([ListDecl([DictDecl([]);DictDecl([])]);ListDecl([DictDecl([])])]));DictMap(ListGet(ListGet(Id(b),Num_lit(0.)),Num_lit(1.)),String_lit(one),Num_lit(1.));DictDelete(ListGet(ListGet(Id(b),Num_lit(0.)),Num_lit(1.)), String_lit (one));Assign(c,Binop(ListGet(Id(b),Num_lit(0.)),Combine,Id(a)));Return(Id(c))]})]

Pseudo/tests/parser/_list_of_empty_lists.in

```
def main()~
    a = [[],   []];
    b = [[1,  2],  [3,  4]];
    c = a :: b;
    d = [1,  2];
    e = [[[],   []],   [[],   [],   []]];
    f = [[[1,  2],  [3,  4]],   [[5]],   [[2  + 2 + 2]]];
    g = e :: f;
    h = [[],   [1,  2]];
    i = b :: h;
    return 0;
$
```

Pseudo/tests/parser/_list_of_empty_lists.out

[Fdecl({params=[];body=[Assign(a,ListDecl([ListDecl([]);ListDecl([])]));Assign(b,ListDecl([ListDecl([Num_lit(1.);Num_lit(2.)]);ListDecl([Num_lit(3.);Num_lit(4.)])]));Assign(c,Binop(Id(a),Combine,Id(b)));Assign(d,ListDecl([Num_lit(1.);Num_lit(2.)]));Assign(e,ListDecl([ListDecl([ListDecl([]);ListDecl([])]);ListDecl([ListDecl([]);ListDecl([]);ListDecl([])])]));Assign(f,ListDecl([ListDecl([ListDecl([Num_lit(1.);Num_lit(2.)]);ListDecl([Num_lit(3.);Num_lit(4.)])]);ListDecl([ListDecl([Num_lit(5.)])])]);ListDecl([ListDecl([Binop(Binop(Num_lit(2.),Add,Num_lit(2.)),Add,Num_lit(2.))])])]));Assign(g,Binop(Id(e),Combine,Id(f)));Assign(h,ListDecl([ListDecl([]);ListDecl([Num_lit(1.);Num_lit(2.)])]));Assign(i,Binop(Id(b),Combine,Id(h)));Return(Num_lit(0.))]})]

Pseudo/tests/parser/_list_of_list.in

```
def main()~
    a = [[1,  2],  [3,  4],  [5,  6]];
    return 0;
$
```

Pseudo/tests/parser/_list_of_list.out

[Fdecl({params=[];body=[Assign(a,ListDecl([ListDecl([Num_lit(1.);Num_lit(2.)]);ListDecl([Num_lit(3.);Num_lit(4.)]);ListDecl([Num_lit(5.);Num_lit(6.)])]));Return(Num_lit(0.))]})]

Pseudo/tests/parser/_list_ops.in

```
def main()~
    a = [1,  2,  3];
    b = ["a", "b", "c"];

    a.insert(3,  4);
    b.push("d");

    a.remove(2);
    a.pop();
    b.dequeue();

    c = a.length();
```

```
    a.get(0);
    a.set(0, 5);
    b.set(0, "e");

    return 0;
$
```

Pseudo/tests/parser/_list_ops.out

```
[Fdecl({params=[];body=[Assign(a,ListDecl([Num_lit(1.);Num_lit(2.);Num_lit(3.)]));Assign(b,ListDecl([ String_lit (a); String_lit (b);
    String_lit (c)]));ListInsert(Id(a),Num_lit(3.),Num_lit(4.));ListPush(Id(b), String_lit (d));ListRemove(Id(a),Num_lit(2.));
    ListPop(Id(a));ListDequeue(Id(b));Assign(c,ListLength(Id(a)));ListGet(Id(a),Num_lit(0.));ListSet(Id(a),Num_lit(0.),Num_lit
    (5.));ListSet(Id(b),Num_lit(0.), String_lit (e));Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/_listdecl.in

```
def HELLO()~
a =[];
b = [1, 2, 3];
c = ["a","b","c"];
d = c;
e = 1;
f = 2;
g = [e, f];
$
```

Pseudo/tests/parser/_listdecl.out

```
[Fdecl({params=[];body=[Assign(a,ListDecl([]));Assign(b,ListDecl([Num_lit(1.);Num_lit(2.);Num_lit(3.)]));Assign(c,ListDecl([
    String_lit (a); String_lit (b); String_lit (c)]));Assign(d,Id(c));Assign(e,Num_lit(1.));Assign(f,Num_lit(2.));Assign(g,ListDecl([
    Id(e);Id(f)]))]})]
```

Pseudo/tests/parser/_nesting.out

```
def A()~
if True~
print "a";
if True~
print "b";
$else~
print "c";
$$else~
print "d";
$$
```

Pseudo/tests/parser/_no_return.in

```
def main()~
    foo();
$

def foo()~
$
```

Pseudo/tests/parser/_no_return.out

```
[Fdecl({params=[];body=[Call(foo,[])]});Fdecl({params=[];body=[]})]
```

Pseudo/tests/parser/_noexpr_return.in

```
def main()~
    return;
$
```

Pseudo/tests/parser/_noexpr_return.out

```
[Fdecl({params=[];body=[Return(Noexpr)]})]
```

Pseudo/tests/parser/_obj.in

```
def HELLO()~
    a.a = 5;
$
```

Pseudo/tests/parser/_obj.out

```
[Fdecl({params=[];body=[ObjectAssign(Id(a),a,Num_lit(5.))]})]
```

Pseudo/tests/parser/_obj_init.in

```
def main()~
    init a;
    init x, y, z;
$
```

Pseudo/tests/parser/_obj_init.out

```
[Fdecl({params=[];body=[ObjectInit([a]);ObjectInit([x,y,z])]})]
```

Pseudo/tests/parser/_objreassign.in

```
def HELLO()~
    a.name = "Rayray";
    b.name = a.name;
$
```

Pseudo/tests/parser/_objreassign.out

```
[Fdecl({params=[];body=[ObjectAssign(Id(a),name,String_lit(Rayray));ObjectAssign(Id(b),name,ObjectField(Id(a),name))]})]
```

Pseudo/tests/parser/_trailing_newlines.out

Pseudo/tests/parser/_undetermined_dict.in

```
def main()~
    a = {};
    a.map(1, "one");
    a.delete(1);

    b = {};
    b.map(1, []);
    return b.find(1);
$
```

Pseudo/tests/parser/_undetermined_dict.out

```
[Fdecl({params=[];
    body=[Assign(a,DictDecl([]));
    DictMap(Id(a),Num_lit(1.),String_lit(one));
    DictDelete(Id(a),Num_lit(1.));
    Assign(b,DictDecl([]));
    DictMap(Id(b),Num_lit(1.),ListDecl([]));
    Return(DictFind(Id(b),Num_lit(1.)))]})]
```

Pseudo/tests/parser/_undetermined_list.in

```
def main()~
    a = [];
    a.insert(0, 1);
    a.push(2);
    a.dequeue();
    return 0;
$
```

Pseudo/tests/parser/_undetermined_list.out

```
[Fdecl({params=[];
    body=[Assign(a,ListDecl([]));
    ListInsert(Id(a),Num_lit(0.),Num_lit(1.));
    ListPush(Id(a),Num_lit(2.));
    ListDequeue(Id(a));
    Return(Num_lit(0.))]})]
```

Pseudo/tests/parser/Makefile

```
# Pseudo: test/parser Makefile
# - builds the parserize executable for printing parsed strings from stdin

OCAMLC = ocamlc
OBJS = ../../compiler/parser.cmo ../../compiler/scanner.cmo parserize.cmo
INCLUDES = -I ../../compiler

default: parserize

all:
	cd ..; make all

parserize: $(OBJS)
	$(OCAMLC) $(INCLUDES) -o parserize $(OBJS)

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

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

.PHONY: clean
clean:
	rm -f parserize *.cmo *.cmi

# Generated by ocamldep *.ml
parserize.cmo:
parserize.cmx:
```

Pseudo/tests/parser/parserize.ml

```
open Ast
open Printf

(* Unary operators *)
let txt_of_unop = function
    | Neg -> "Neg"
    | Not -> "Not"

(* Binary operators *)
let txt_of_binop = function
  (* Arithmetic *)
  | Add -> "Add"
  | Minus -> "Minus"
  | Times-> "Times"
  | Divide -> "Divide"
  (* Boolean *)
  | Or -> "Or"
  | And -> "And"
  | Eq -> "Eq"
  | Neq -> "Neq"
  | Less -> "Less"
  | Leq -> "Leq"
  | Greater -> "Greater"
  | Geq -> "Geq"
  (* List *)
  | Combine -> "Combine"
  (* String *)
  | Concat -> "Concat"

(* Expressions *)
(*let txt_of_num = function
  | Num_int(x) -> string_of_int x
  | Num_float(x) -> string_of_float x*)

let rec txt_of_expr = function
  | Num_lit(x) -> sprintf "Num_lit(%s)" (string_of_float x)
  | String_lit (x) -> sprintf " String_lit (%s)" x
  | Bool_lit (x) -> sprintf "Bool_lit(%s)" (string_of_bool x)
  (* | None_lit -> sprintf "None lit" *)
  | Id(x) -> sprintf "Id(%s)" x
  | Unop(op, e) -> sprintf "Unop(%s, %s)" (txt_of_unop op) (txt_of_expr e)
  | Binop(e1, op, e2) -> sprintf "Binop(%s, %s, %s)"
      (txt_of_expr e1) (txt_of_binop op) (txt_of_expr e2)
  | Call(str , args) -> sprintf "Call(%s, [%s])"
      str ( txt_of_list args)
  | Assign(x, e) -> sprintf "Assign(%s, %s)" x (txt_of_expr e)
```

```
  | ListDecl(l) -> sprintf "ListDecl([%s])" ( txt_of_list  l)
  (* List operations *)
  | ListInsert(e1, e2, e3) -> sprintf "ListInsert(%s, %s, %s)" (txt_of_expr e1) (
      txt_of_expr e2) (txt_of_expr e3)
  | ListPush(e1, e2) -> sprintf "ListPush(%s, %s)" (txt_of_expr e1) (txt_of_expr e2)
  | ListRemove(e1, e2) -> sprintf "ListRemove(%s, %s)" (txt_of_expr e1) (txt_of_expr e2
      )
  | ListPop(e) -> sprintf "ListPop(%s)" (txt_of_expr e)
  | ListDequeue(e) -> sprintf "ListDequeue(%s)" (txt_of_expr e)
  | ListLength(e) -> sprintf "ListLength(%s)" (txt_of_expr e)
  | ListGet(e1, e2) -> sprintf "ListGet(%s, %s)" (txt_of_expr e1) (txt_of_expr e2)
  | ListSet(e1, e2, e3) -> sprintf "ListSet(%s, %s, %s)" (txt_of_expr e1) (txt_of_expr
      e2) (txt_of_expr e3)
  (* Dict Operations *)
  | DictDecl(d) -> sprintf "DictDecl([%s])" (txt_of_dict  d)
  | DictMem(e1, e2) -> sprintf "DictMem(%s, %s)" (txt_of_expr e1) (txt_of_expr e2)
  | DictFind(e1, e2) -> sprintf "DictFind(%s, %s)" (txt_of_expr e1) (txt_of_expr e2)
  | DictMap(e1, e2, e3) -> sprintf "DictMap(%s, %s, %s)" (txt_of_expr e1) (txt_of_expr
  e2) (txt_of_expr e3)
  | DictDelete(e1, e2) -> sprintf "DictDelete(%s, %s)" (txt_of_expr e1) (txt_of_expr e2)
  | DictSize(e) -> sprintf "DictSize(%s)" (txt_of_expr e)
  (*| Assert(e) -> sprintf "Assert([%s])" (txt_of_expr e) *)
  | Noexpr -> sprintf "Noexpr"
  | ObjectField(e, s2) -> sprintf "ObjectField(%s, %s)" (txt_of_expr e) s2
  | ObjectAssign(e1, s2, e2) -> sprintf "ObjectAssign(%s, %s, %s)" (txt_of_expr e1) s2
      (txt_of_expr e2)


(* Lists *)
and txt_of_list = function
  | [] -> ""
  | [x] -> txt_of_expr x
  | _ as l -> String.concat " ; " (List.map txt_of_expr l)

and txt_of_tup (tup: expr * expr) =
  sprintf "(%s, %s)" (txt_of_expr (fst tup)) (txt_of_expr (snd tup))

and txt_of_dict = function
  | [] -> ""
  | [t] -> (txt_of_tup t)
  | _ as d -> String.concat " ; " (List.map txt_of_tup d)

and txt_of_stringlist ( stringlist : string list ) =
  let rec aux acc = function
    | [] -> sprintf "%s" (String.concat ", " (acc))
    | hd :: tl -> aux (hd :: acc) tl
  in aux [] stringlist
```

```
(* Statements *)
and txt_of_stmt (st: stmt) =
  match st with
    | Expr(e) -> sprintf "%s" (txt_of_expr e)
    | Return(e) -> sprintf "Return(%s)" (txt_of_expr e)
    | Break -> sprintf "Break"
    | Continue -> sprintf "Continue"
    | While(e, s) -> sprintf "While(%s, %s)" (txt_of_expr e) (txt_of_stmt s)
    | ForIn(e1, e2, s) -> sprintf "ForIn(%s, %s, %s)" (txt_of_expr e1) (txt_of_expr e2)
        (txt_of_stmt s)
    | ForRange(e1, e2, e3, s) ->
        sprintf "ForRange(%s, %s, %s, %s)" (txt_of_expr e1) (txt_of_expr e2)
                                           (txt_of_expr e3) (txt_of_stmt s)
    | If(e1, s1, s2) -> sprintf "If(%s, %s, %s)"
      (txt_of_expr e1) (txt_of_stmt s1) (txt_of_stmt s2)
    | Block(sl) -> (txt_of_stmts sl)
    | Print(e) -> sprintf "Print([%s])" (txt_of_expr e)
    | ObjectInit( obj_list ) -> sprintf "ObjectInit([%s])" ( txt_of_stringlist obj_list )

and txt_of_stmts (stmts: stmt list): string =
  let rec aux acc = function
      | [] -> sprintf "[%s]" (String.concat " ; " (List.rev acc))
      | stmt :: tl -> aux (txt_of_stmt stmt :: acc) tl
  in aux [] stmts

(* Function declarations *)
and txt_of_fdecl (f: func_decl): string =
    sprintf "Fdecl({ params=[%s] ; body=%s})" (String.concat " ; " f.formals) (
        txt_of_stmts f.body)


and txt_of_fdecls fdecls =
  let rec aux acc = function
      | [] -> sprintf "[%s]" (String.concat " ; " (List.rev acc))
      | fdecl :: tl -> aux ((txt_of_fdecl fdecl) :: acc) tl
  in aux [] fdecls


(* Program entry point *)
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let result = txt_of_fdecls program in
  print_endline result
```

Pseudo/tests/preprocessor/_call.in

```
def HELLO():
```

```
    print "hello world"

def MAIN():
    HELLO()
```

Pseudo/tests/preprocessor/_call.out

```
def HELLO()~
    print"helloworld";

$

def MAIN()~
    HELLO();
$
```

Pseudo/tests/preprocessor/_comments.in

```
def MAIN(): // this is a function
    if "// edge case": // comment
        print "cool //"
```

Pseudo/tests/preprocessor/_comments.out

```
def MAIN()~
if "// edge case"~
print "cool //";
$$
```

Pseudo/tests/preprocessor/_eof.in

```
if True:
    if True:
        if True:
            print "eof"
```

Pseudo/tests/preprocessor/_eof.out

```
if True~
if True~
if True~
print "eof";
$$$
```

Pseudo/tests/preprocessor/_hello.in

```
def HELLO():
    print "hello world"
```

Pseudo/tests/preprocessor/_hello.out

```
def HELLO()~
print "hello world";
$
```

Pseudo/tests/preprocessor/_hello_spaces.in

```
def HELLO():
    print "hello world"
```

Pseudo/tests/preprocessor/_hello_spaces.out

```
def HELLO()~
print "hello world";
$
```

Pseudo/tests/preprocessor/_nesting.in

```
def A():
        if True:
                print "a"
                if True:
                        print "b"
                else:
                        print "c"
        else:
                print "d"
```

Pseudo/tests/preprocessor/_nesting.out

```
def A()~
if True~
print "a";
if True~
print "b";
$else~
print "c";
$$else~
print "d";
$$
```

Pseudo/tests/preprocessor/_nesting_spaces.in

```
def A():
        if True:
                print "a"
                if True:
                         print "b"
                else:
                        print "c"
        else:
                print "d"
```

Pseudo/tests/preprocessor/_nesting_spaces.out

```
def A()~
if True~
print "a";
if True~
print "b";
$else~
print "c";
$$else~
print "d";
$$
```

Pseudo/tests/preprocessor/_rec.in

```
def rec(a):
        if a == 0:
                return 0
        return rec(a - 1) + 1

def main():
        b = rec(5)
        print b
```

Pseudo/tests/preprocessor/_rec.out

```
def rec(a)~
    if a == 0~
    return 0;
    $
    return rec(a - 1) + 1;
$

def main()~
    b = rec(5);
    print b;
$
```

Pseudo/tests/preprocessor/_run_tests.py

```
"""
Runs tests for the preprocessor.

Usage:
    python run_tests.py
"""

import filecmp
import os


TEST_DIR = 'tests'


def run_tests():
    for filename in os.listdir(TEST_DIR):
        if filename.endswith('.in'):
            infile = TEST_DIR + '/' + filename
            outfile = infile + 'p'
            test_command = 'python preprocessing.py {} {}' \
                .format(infile, outfile)
            print test_command
            os.system(test_command)
            print 'PASS' if filecmp.cmp(infile[:-3] + '.out', outfile) \
                else 'FAIL'


if __name__ == '__main__':
    run_tests()
```

Pseudo/tests/preprocessor/_selectionsort.in

```
def main():
        list = [5, 3, 7, 1, 2, 6]
```

```
for i = 0 to 6 - 1: // tests if we supports expressions in for loops
        min = 99999
        min_index = i + 1
        for j = i to 6:
                if list.get(j) < min:
                        min = list.get(j)
                        min_index = j
        temp = list.get(i)
        list.set(i, list.get(min_index))
        list.set(min_index, temp)

for i = 0 to 6: // see if we can reuse i
        print list.get(i)
```

Pseudo/tests/preprocessor/_selectionsort.out

```
def main()~
        list = [5,3,7,1,2,6];

        for i =0 to 6-1~min=99999;min_index=i+1;

        forj=ito6~
                iflist .get(j)<min~min=list.get(j);
                        min_index=j;
                $
        $

        temp=list.get(i);
        list.set(i, list.get(min_index));
        list.set(min_index,temp);$
        fori=0to6~
                printlist.get(i);
        $
$
```

Pseudo/tests/scanner/_assign.in

```
=
```

Pseudo/tests/scanner/_assign.out

```
ASSIGN
```

Pseudo/tests/scanner/_binops.in

```
+ - * /
```

Pseudo/tests/scanner/_binops.out

```
PLUS
MINUS
TIMES
DIVIDE
```

Pseudo/tests/scanner/_brace.in

```
{ }
```

Pseudo/tests/scanner/_brace.out

```
LBRACE
RBRACE
```

Pseudo/tests/scanner/_colon.out

```
;
```

Pseudo/tests/scanner/_cond.in

```
if  else
```

Pseudo/tests/scanner/_cond.out

```
IF
ELSE
```

Pseudo/tests/scanner/_control.in

```
while  for  in  continue  break
```

Pseudo/tests/scanner/_control.out

```
WHILE
FOR
IN
CONTINUE
BREAK
```

Pseudo/tests/scanner/_def.in

```
def
```

Pseudo/tests/scanner/_def.out

```
DEF
```

Pseudo/tests/scanner/_eqneq.in

```
== !=
```

Pseudo/tests/scanner/_eqneq.out

```
EQ
NEQ
```

Pseudo/tests/scanner/_id.in

```
aa_b
Aaaa
aa
a_a
a92
```

Pseudo/tests/scanner/_id.out

```
ID
ID
ID
ID
ID
```

Pseudo/tests/scanner/_is.out

```
IS
```

Pseudo/tests/scanner/_key.in

```
return
```

Pseudo/tests/scanner/_key.out

```
RETURN
```

Pseudo/tests/scanner/_literals.in

```
1 4 100 3.0 1. .3 "string lit" "hello world" true false
```

Pseudo/tests/scanner/_literals.out

```
NUM_LITERAL
NUM_LITERAL
NUM_LITERAL
NUM_LITERAL
NUM_LITERAL
NUM_LITERAL
STRING_LITERAL
STRING_LITERAL
BOOL_LITERAL
BOOL_LITERAL
```

Pseudo/tests/scanner/_logic.in

```
&& || !
```

Pseudo/tests/scanner/_logic.out

```
AND
OR
NOT
```

Pseudo/tests/scanner/_ltgt.in

```
< >
```

Pseudo/tests/scanner/_ltgt.out

```
LT
GT
```

Pseudo/tests/scanner/_obj.in

```
A.neighbors
```

Pseudo/tests/scanner/_obj.out

```
ID DOT ID
```

Pseudo/tests/scanner/_obj_init.in

```
init a;
init b, c;
init node1, node2, node3;
```

Pseudo/tests/scanner/_obj_init.out

```
INIT ID SEMI
INIT ID COMMA ID SEMI
INIT ID COMMA ID COMMA ID SEMI
```

Pseudo/tests/scanner/_paren.in

```
()
```

Pseudo/tests/scanner/_paren.out

```
LPAREN
RPAREN
```

Pseudo/tests/scanner/_utils.in

print

Pseudo/tests/scanner/_utils.out

```
PRINT
```

Pseudo/tests/scanner/Makefile

```
# test/scanner Makefile
# − builds the tokenize executable for printing scanned tokens from stdin

OCAMLC = ocamlc
OBJS = ../../compiler/scanner.cmo tokenize.cmo
INCLUDES = −I ../../compiler

default: tokenize

all:
	cd ..; make all

tokenize: $(OBJS)
	$(OCAMLC) $(INCLUDES) −o tokenize $(OBJS)

%.cmo: %.ml
	$(OCAMLC) $(INCLUDES) −c $<

%.cmi: %.mli
	$(OCAMLC) $(INCLUDES) −c $<

.PHONY: clean
clean:
	rm −f tokenize ∗.cmo ∗.cmi

# Generated by ocamldep ∗.ml
tokenize.cmo:
tokenize.cmx:
```

Pseudo/tests/scanner/tokenize.ml

```
open Parser
open Ast

type num =
  | Num_int of int
  | Num_float of float

let stringify = function
  (∗ Punctuation ∗)
  | LPAREN −> "LPAREN" | RPAREN −> "RPAREN"
  | LBRACE −> "LBRACE" | RBRACE −> "RBRACE"
```

```
| LBRACKET -> "LBRACKET" | RBRACKET -> "RBRACKET"
| COMMA -> "COMMA" | TILDE  -> "TILDE"
| DOLLAR -> "DOLLAR" | SEMI  -> "SEMI"
| DOT -> "DOT"

(* Arithmetic Operators *)
| PLUS -> "PLUS"      | MINUS   -> "MINUS"
| TIMES -> "TIMES"    | DIVIDE  -> "DIVIDE"

(* List Operators *)
| COMBINE -> "COMBINE" | PUSH -> "PUSH"
| INSERT -> "INSERT" | DEQUEUE -> "DEQUEUE"
| POP    -> "POP"     | REMOVE -> "REMOVE"
| LENGTH -> "LENGTH"

(* Dict Operation *)
| GET    -> "GET"     | SET      -> "SET"
| MEM    -> "MEM"     | FIND     -> "FIND"
| DEL    -> "DEL"     | MAP      -> "MAP"
| SIZE     -> "SIZE"

(* String Operators *)
| CONCAT -> "CONCAT"

(* Relational Operators *)
| EQ -> "EQ" | NEQ -> "NEQ"
| LT -> "LT"  | GT -> "GT"
| LEQ -> "LEQ" | GEQ -> "GEQ"

(* Logical Operators & Keywords *)
| AND -> "AND" | OR -> "OR"
| NOT -> "NOT"

(* Assignment Operator *)
| ASSIGN -> "ASSIGN"

(* Conditional Operators *)
| IF -> "IF" | ELSE -> "ELSE"
| ELSIF -> "ELSIF"

(* Control Flow *)
| WHILE -> "WHILE" | FOR -> "FOR"
| BY -> "BY" | TO -> "TO"
| IN -> "IN" | CONTINUE -> "CONTINUE"
| BREAK -> "BREAK"

(* Function Symbols & Keywords *)
```

```
  | DEF -> "DEF"
  | COLON -> "COLON"
  | RETURN -> "RETURN"

  (* UTILS *)
  | PRINT -> "PRINT" (* | ASSERT -> "ASSERT" *)

  (* End-of-File *)
  | EOF -> "EOF"

  (*  Identifiers  *)
  | ID(string) -> "ID"

  (*  Literals  *)
  | NUM_LITERAL(num) -> "NUM_LITERAL"
  | STRING_LITERAL(string) -> "STRING_LITERAL"
  | BOOL_LITERAL(bool) -> "BOOL_LITERAL"
  (*| VOID_LITERAL -> "VOID_LITERAL"*)
  (* | NONE -> "NONE"*)

  (* Objects *)
  | INIT -> "INIT"

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let rec print_tokens = function
    | EOF -> " "
    | token ->
      print_endline (stringify token);
      print_tokens (Scanner.token lexbuf) in
  print_tokens (Scanner.token lexbuf)
```