FRY Language Reference

Tom DeVoe
tcd2123@columbia.edu

December 17, 2014

Contents

1 Introduction **2**

2 Language Tutorial **3**
2.1 Environment Setup . . . 3
2.2 Compiling and Running FRY Programs . . . 3
2.3 Basic Types . . . 3
2.4 Complex Types . . . 3
2.4.1 Lists . . . 3
2.4.2 Layouts . . . 4
2.4.3 Tables . . . 4
2.5 Functions . . . 5
2.5.1 User Defined Functions . . . 5
2.5.2 Built-in Functions . . . 5
2.6 Examples . . . 6
2.6.1 Generate Holiday Calendar . . . 6
2.6.2 Filtering Example . . . 7

3 Language Reference Manual **8**
3.1 Lexical Conventions . . . 8
3.1.1 Comments . . . 8
3.1.2 Identifiers . . . 8
3.1.3 Keywords . . . 8
3.1.4 Constants . . . 8
3.2 Syntax Notation . . . 8
3.3 Meaning of Identifiers . . . 9
3.3.1 Types . . . 9
3.4 Conversions . . . 9
3.4.1 Integer and Floating . . . 9
3.4.2 Arithmetic Conversions . . . 9
3.5 Expressions . . . 9
3.5.1 Primary Expressions . . . 9
3.5.2 Set Builder Expressions . . . 10
3.5.3 Postfix Expression . . . 10
3.5.4 Prefix Expressions . . . 11
3.5.5 Multiplicative Operators . . . 11
3.5.6 Additive Operators . . . 11
3.5.7 Relational Operators . . . 12
3.5.8 Equality Operators . . . 12
3.5.9 Logical AND Operator . . . 12
3.5.10 Logical OR Operator . . . 12
3.5.11 Assignment Expressions . . . 12
3.5.12 Function Calls . . . 13

3.5.13 List Initializers . . . 13
3.5.14 Layout Initializer . . . 13
3.5.15 Table Initializer . . . 13
3.5.16 Expressions . . . 13
3.6 Declarations . . . 14
3.6.1 Type Specifiers . . . 14
3.6.2 Variable Declarations . . . 14
3.6.3 Layout Declarations . . . 14
3.6.4 Function Declarations . . . 15
3.7 Statements . . . 15
3.7.1 Expression Statements . . . 15
3.7.2 Return Statement . . . 15
3.7.3 Statement Block . . . 16
3.7.4 Conditional Statements . . . 16
3.7.5 Iterative Statements . . . 16
3.7.6 while loop . . . 16
3.8 FRY Program . . . 16
3.9 Scope . . . 16

4 Project Plan **17**
4.1 Process . . . 17
4.1.1 Specification . . . 17
4.1.2 Development . . . 17
4.1.3 Testing . . . 17
4.2 Programming Style Guide . . . 17
4.3 Project Timeline . . . 17
4.4 Development Environment . . . 17
4.5 Project Log . . . 18

5 Compiler Architecture **20**
5.1 Scanner . . . 20
5.2 Parser . . . 20
5.3 Semantic Analysis . . . 20
5.4 Java Code Generation . . . 20

6 Test Plan **21**
6.1 Example Programs and Generated Code . . . 21
6.1.1 Holiday Calendar Example . . . 21
6.1.2 Filtering Example . . . 23
6.2 Test Suite . . . 25

7 Lessons Learned **25**

8 Appendix **25**

1 Introduction

As data becomes cheaper to store and more readily available, there is a need for tools to process and analyze that data. While there are a wealth of languages and software which already accomplish this, FRY sets out to do so in an elegant, concise manner. The main goal is to allow users to quickly and easily define data transformations so that they can get the most information out of their data. At its core, FRY is an imperative, staticly-typed language built to process delimited text files. Once these delimited text files are read into a FRY program, you can perform any number of transformations on this data using built-in or user defined functions, or using FRY's set-builder notation, which allows for concise statements to transform data.

2 Language Tutorial

2.1 Environment Setup

Since FRY compiles to Java, it should run on any machine which has Java 7 or greater installed. However, the FRY compiler wrapper scripts will only work in an environment with **sh**.

- Add the FRY Standard Library JAR to your classpath.
- Add the FRY binaries to your PATH

2.2 Compiling and Running FRY Programs

FRY includes two helper scripts which allow you to compile and run easily. To compile, simply run (assuming compile_fry.sh is in your path):

```
compile_fry.sh my_program.fry
```

Where **my_program.fry** would be replaced by the name of the FRY program you would like to compile. This will generate **fry.class** and then you need to just run **java fry** to execute your program. Similarly, to compile and run a FRY program, you just need to run:

```
run_fry.sh my_program.fry
```

Alternatively, you can use the FRY compiler directly, which just compiles to the java program. To do this, you only need to run the command:

```
fry -c < my_program.fry
```

2.3 Basic Types

FRY supports the primitive types **bool**, **str**, **int**, and **float**. To declare a basic variable, you just need type the type specifier followed by the identifier, like so:

```
# String declaration
str my_str;
# Integer declaration
int my_int;
```

You can also optionally assign these variables a variable in the same line as the declaration, or assign a value later on: ‘

```
str my_str = "Tom";
int my_int = 42;
```

2.4 Complex Types

FRY supports non-primitive types **List**, **Layout**, and **Table**.

2.4.1 Lists

A **List** is simply a collection of values of the same type. This is called an array in many other languages. A list is declared by specifying the type of the elements of the list, followed by the list keyword and the identifier:

```
# A list of strings
str List str_list;
# A list of ints
int List int_list;
```

And a list can be initialized to values using either of the list initializers. The first initializes the list to a list of values specified:

```
str List str_list = [ "This", "is", "a", "test" ];
int List int_list = [ 1, 2, 3, 4, 5 ];
```

The second initializes an integer list to a range of values. This list initializer takes two integers and generates an inclusive list of integers in that range.

```
int List int_list = [ 1 to 42 ];
```

You can also use slicing to obtain subsets of lists.

```
# Obtains the first 30 elements
int List int_sublist = int_list[:30];
# Obtains the elements 12 to end
int_sublist = int_list[12:];
# Obtains elements 12 to 15
int_sublist = int_list[12:15];
```

2.4.2 Layouts

A **Layout** is a compound type which can hold a collection of variables of different types. Layouts are also associated with **Tables**, representing the "Layout" of a record in that table. This will be covered more in . A Layout is declared with its constituent elements:

```
Layout date = {str: mon, int: day, int: year};
```

Layouts can be nested for more complex structures:

```
Layout user = { int: id, str: fname, str: lname, Layout date: bday };
```

And literals of this layout can be created like so:

```
Layout date christmas = {"Dec",25,2014};
```

2.4.3 Tables

A table is a programmatic representation of the delimited text files that FRY processes. Before a Table can contain data, it must first be initialized (optionally with an associated Layout):

```
Table date_tbl = Table (Layout date);
Table generic_tbl = Table ();
```

Once a table is initialized you can assign it a value by using the built-in function **Read** to read in a file:

```
date_tbl = Read("date_info.txt", ",");
```

Read takes two arguments, a string of the relative or absolute file to be read, and a delimiter that the file's data is separated by.

Or using set-builder notation you can obtain a table from an already existing table. FRY's set-builder notation is made up of three expressions:

- **A Output Record Format** - This is the ordering of the fields you would like in your output.
- **A Foreach Expression** - This specifies which table the records are coming from
- **A Record Filter** - This is a boolean expression which specifies which records are returned from the set-builder statement

```
# Returns all dates from May and loads into may_tbl
Table may_tbl = [ i | i <- date_tbl; i.{mon} == "May" ];
```

You can use the built-in function "Write" to write out the resulting tables to stdout, stderr, or a file.

```
Write("may_dates.txt",may_tbl);
```

You can also generate a table using the built-in function "Append". This function is illustrated in 2.6.1

2.5 Functions

2.5.1 User Defined Functions

You can define functions anywhere in a FRY program, but they can only be called after they are defined. As a best practice, it is good to define functions in the beginning of your program. Function definitions in FRY are similar to the style of those in C, Java, and many other imperative languages. The definition includes the return type, function identifier, a list of function arguments, and the function body which is a list of FRY statements.

For example, a function which computes the largest integer in a list of integers could be defined like:

```
int getLargestInt(int List l){
  int lrg = 0;
  for ( i <- l ){
    if ( i > lrg ){
      lrg = i;
    }
  }
  ret lrg;
}
```

Functions signatures must be unique. You can *overload* functions, that is defining multiple functions with the same name, but different arguments. When you call that function identifier, the arguments provided will decide which function will be called.

2.5.2 Built-in Functions

FRY has a number of Built in functions which mainly are used for IO and Table operations.

2.5.2.1 Write

Write is a function which reads in a FRY variable and writes it to the specified output stream. This stream can be `stdout`, `stderr`, a relative file path, or an absolute file path. Write will write any FRY data type except Lists. Write has the following signatures:

```
Write(str output_specifier, int value)
Write(str output_specifier, bool value)
Write(str output_specifier, float value)
Write(str output_specifier, str value)
Write(str output_specifier, Layout <type> value)
Write(str output_specifier, Table tbl)
Write(str output_specifier, Table tbl, str delimiter)
```

Where *Layout <type>* is any defined FRY Layout type.

2.5.2.2 Read

Read is a function which reads from **stdin**, a relative file path, or an absolute file path and returns a FRY Table. Read also takes a delimiter argument which signafies how the input data is delimited.

```
Read(str input_specifier, str delimiter)
```

2.5.2.3 Append

Append is a function which appends a Layout instance to a Table.

```
Append(Table tbl, Layout <type> lyt_instance)
```

Where *Layout <type>* is any defined FRY Layout type.

2.5.2.4 Column

Column is a function which takes in a Table and a column name or column number, and returns that column as a FRY List.

```
Column(Table tbl, str column_name)
Column(Table tbl, int column_number)
```

2.6 Examples

2.6.1 Generate Holiday Calendar

This example reads in a file with a list of holidays and generates a calendar with those holidays noted. "holidays.txt" is a comma delimited list of holidays along with the name of that holiday. **"holidays.txt"** looks like :

```
Jan,1,2015,New Years
Jan,19,2015,MLK Day
Feb,16,2015,Washington BDay
May,25,2015,Mem Day
Jul,4,2015,Independence Day
Sep,7,2015,Labor Day
Oct,12,2015,Columbus Day
Nov,11,2015,Veterans Day
Nov,26,2015,Thanksgiving
Dec,25,2015,Christmas Day
```

holiday_cal.fry

```
# Checks if the date is valid (i.e. no Feb 30th)
bool isValidDate(str mon, int day){
  if ( day > 28 and mon == "Feb"){
    ret false;
  }
  if ( day == 31 ){
    if ( mon == "Sep" or mon == "Apr" or mon == "Jun" or mon == "Nov"){
      ret false;
    }
    else {
      ret true;
    }
  }
  ret true;
}
# Define our file layout
Layout date = {str: mon, int: day, int: year, str: holiday};

# read in the file with the holiday listing
Table holidays = Table (Layout date);
holidays = Read("holidays.txt",",");

str holiday_name;
Table calendar = Table (Layout date);
str List holiday_list;
Table matching_days;

# Generate a calendar, marking the holidays down
for ( mon <- [ "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct",
    "Nov", "Dec" ]){
  for ( day <- [ 1 to 31 ]) {
```

```
    if(isValidDate(mon, day)){
        matching_days = [ i | i <- holidays; i.{mon} == mon and i.{day} == day ];
        holiday_list = Column(matching_days, "holiday");
        holiday_name = holiday_list[0];
        Append(calendar, Layout date {mon, day, 2015,holiday_name});
    }
  }
}

Write("holiday-calendar.txt", calendar);
```

The generated calendar file, **"holiday-calendar.txt"** looks like (where the "..." represents unincluded records) :

```
Jan,1,2015,New Years
Jan,2,2015,
Jan,3,2015,
...
Jan,18,2015,
Jan,19,2015,MLK Day
Jan,20,2015,
...
Feb,15,2015,
Feb,16,2015,Washington BDay
Feb,17,2015,
...
```

2.6.2 Filtering Example

This example reads a "—" delimited file which contains information about a number of website hits. This includes the user, site, and date of the access. The input, **website_hits.txt** looks like this:

```
...
11-09-2010 06:48:12|facebook.com|Pippen
07-06-2010 13:17:37|linkedin.com|Sam
28-01-2010 01:56:19|google.com|Merry
17-08-2010 22:09:24|twitter.com|Frodo
25-03-2010 08:34:39|twitter.com|Frodo
24-02-2010 04:26:52|facebook.com|Frodo
09-09-2010 06:41:53|twitter.com|Pippen
...
```

In this example, we want to filter out all of the Facebook accesses by Frodo and the Google accesses by Sam. This is a simple matter with FRY's set-builder notation:

```
# Define source layout and read source data
Layout weblog = {str: date, str: site, str: user};
Table weblog_data = Table (Layout weblog);
weblog_data = Read("website_hits.txt","|");

# Filter required records
Table frodo_facebook = [i | i <- weblog_data; i.{site} == "facebook.com" and i.{user
    } == "Frodo"];
Table sam_google = [i | i <- weblog_data; i.{site} == "google.com" and i.{user} == "
    Sam"];

# write out filtered records to respective files
Write("frodo_facebook.txt", frodo_facebook);
Write("sam_google.txt", sam_google);
```

This writes two files which contain the respective filtered outputs.

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Comments

Single line comments are denoted by the character, `#`. Multi-line comments are opened with `#/` and closed with `/#`.

```
# This is a single line comment

#/ This is a
multi-line comment /#
```

3.1.2 Identifiers

An identifier is a string of letters, digits, and underscores. A valid identifier begins with an letter or an underscore. Identifiers are case-sensitive and can be at most 31 characters long.

3.1.3 Keywords

The following identifiers are reserved and cannot be used otherwise:

```
int     str     float   bool     Layout
List    Table   if      else     elif
in      not     and     stdout
or      Write   Read    stderr   true
false
```

3.1.4 Constants

There is a constant corresponding to each Primitive data type mentioned in 3.3.1.1.

- **Integer Constants** - Integer constants are whole base-10 numbers represented by a series of numerical digits (0 - 9) and an optional leading sign character(+ or −). Absence of a sign character implies a positive number.
- **Float Constants** - Float constants are similar to Integer constants in that they are base-10 numbers represented by a series of numerical digits. However, floats must include a decimal separator and optionally, a fractional part. Can optionally include a sign character (+ or −). Absence of a sign character implies a positive number.
- **String Constants** - String constants are represented by a series of ASCII characters surrounded by quotation-marks (`" "`). Certain characters can be escaped inside of Strings with a backslash `''`. These characters are:

Character	**Meaning**
`\n`	Newline
`\t`	Tab
`\\`	Backslash
`\"`	Double Quotes

- **Boolean Constants** - Boolean constants can either have the case-sensitive value *true* or *false*.

3.2 Syntax Notation

Borrowing from the *The C Programming Language* by Kernigan and Ritchie, syntactic categories are indicated by *italic* type and literal words and characters in `typewriter` style. Optional tokens will be underscored by $_{opt}$.

3.3 Meaning of Identifiers

3.3.1 Types

3.3.1.1 Basic Types

- `int` - 64-bit signed integer value
- `str` - An ASCII text value
- `float` - A double precision floating-point number
- `bool` - A boolean value. Can be either `true` or `false`

3.3.1.2 Compound Types

- `List` - an ordered collection of elements of the same data type. Every column in a *Table* is represented as a List. Lists can be initialized to an empty list or one full of values like so:
- `Layout` - a collection of named data types. Layouts behave similar to structs from C. Once a Layout is constructed, that layout may be used as a data type. An instance of a Layout is referred to as a *Record* and every table is made up of records of the Layout which corresponds to that table.
- `Table` - a representation of a relational table. Every column in a table can be treated as a *List* and every row is a record of a certain *Layout*. Tables are the meat and potatoes of **FRY** and will be the focus of most programs.

3.4 Conversions

Certain operators can cause different basic data types to be converted between one another.

3.4.1 Integer and Floating

Integer and Floating point numbers can be converted between each other by simply creating a new identifier of the desired type and assigning the variable to be converted to that identifier.

3.4.2 Arithmetic Conversions

For any binary operator with a floating point and an integer operator, the integer will be promoited to a float before the operation is performed.

3.5 Expressions

An expression in **FRY** is a combination of variables, operators, constants, and functions. The list of expressions below are listed in order of precedence. Every expression in a subsubsection shares the same precedence (ex. Identifiers and Constants have the same precedence).

3.5.1 Primary Expressions

primary-expression :
    identifier
    literal
    (expression)

Primary Expressions are either identifiers, constants, or parenthesized expressions.

3.5.1.1 Identifiers

Identifiers types are specified during declaration by preceding that identifier by its type. Identifiers can be used for any primitive or compound data types and any functions.

3.5.1.2 Literal

Literals are either integer, string, float, or boolean constants as specified in 3.1.4

3.5.1.3 Parenthesized Expressions

Parenthesized expression is simply an expression surrounded by parentheses.

3.5.2 Set Builder Expressions

set-build-expression:
  primary-expression
  `[` *return-layout* `|` *identifier* `<-` *set-build-expression* `;` *expression* `]`

A set-build-expression consists of a *return-layout*, which is the format of the columns which should be returned, an identifier for records in the table identifier specified by *set-build-expression*, and an *expression* which is a boolean expression.

The Set-builder notation evaluates the boolean expression for every record in the source table. If the boolean expression is true, then the *return-layout* is returned for that record. The Set Builder expression finally returns a table composed of all of the records which passed the boolean condition, formatted with the *return-layout*.

return-layout:
  identifier
  { *layout specifier*$_{\text{opt}}$ *layout-instance-list* }

The *return-layout* must be a Layout type, and can be either a Layout *identifier* or a *Layout-instance-list* as described in 3.6.3.

3.5.3 Postfix Expression

Operators in a postfix expression are grouped from left to right.

postfix-expression :
  set-build-expression
  postfix-expression[slice-opt]
  postfix-expression.{*expression*$_{opt}$}
  expression−−
  expression++

slice-opt :
  :expr
  expr:
  expr:expr
  expr

3.5.3.1 List Element Reference

A list identifier followed by square brackets with an integer-valued expression inside denotes referencing the element at that index in the List. For instance `MyLst[5]` would reference the 6th element of the List, *MyLst*. Similarly, `MyLst[n]` would reference the $n-1^{\text{th}}$ element of MyLst. The type of this element is the same as the type of elements the List you are accessing contains.
Sublists can be returned by *slicing* the list. By specifying the optional colon (':') and indices before and/or after, the list is sliced and a sublist of the original list is returned. If there is an integer before the semi-colon and none after, then a sublist is returned spanning from the integer to the end of the list. If there is an integer after the colon and none before, the a sublist is returned spanning from the beginning of the list to the integer index. If there is an integer before and after the colon, then a sublist is returned spanning from the first integer index to the second integer index.

3.5.3.2 Layout Element Reference

A layout identifier followed by a dot and an expression in braces references an element of a layout. The expression in the braces must either be *(i)* the name of one of the member elements in the Layout you are accessing, such as `MyLyt.{elem_name}` or *(ii)* a integer reference to the nth element of the Layout, i.e. `MyLyt.{2}` would access the 1st member element. The type of the element returned will be the type that element was defined to be when the Layout was defined. If the member element you are accessing is itself a Layout, then the numeric and identifier references will both return a element of that Layout type.

3.5.3.3 *expression*−−

The double minus sign ('-') decrements an integer value by 1. The type of this expression must be integer.

3.5.3.4 *expression*++

The double plus sign ('+') increments an integer value by 1. The type of this expression must be integer.

3.5.4 Prefix Expressions

Unary operators are grouped from right to left and include logical negation, incrementation, and decrementation operators.

prefix-expression :
  postfix-expression
  `not` *unary-expression*

3.5.4.1 `not` *expression*

The `not` operator represents boolean negation. The type of the expression must be boolean.

3.5.5 Multiplicative Operators

These operators are grouped left to right.

multiplicative-expression :
  unary-expression
  *multiplicative-expression*multiplicative-expression*
  multiplicative-expression/multiplicative-expression

$*$ denotes mutltiplication, / denotes division, and % returns the remainder after division (also known as the modulo). The expressions on either side of these operators must be integer or floating point expressions. If the operand of / or % is 0, the result is undefined.

3.5.6 Additive Operators

These operators are grouped left to right.

additive-expression :
  multiplicative-expression
  additive-expression+additive-expression
  additive-expression-additive-expression

$+$ and $-$ denote addition and subtraction of the two operands respectively. Additionally the $+$ also denotes string concatenation. For $-$, the expressions on either side of the operators must be either integer or floating point valued. For $+$, the expressions can be integer, floating point or strings. Both operands must be strings or both operands must be float/int. You cannot mix string operands with numeric operands.

3.5.7 Relational Operators

relational-expression :
  additive-expression
  relational-expression>*relational-expression*
  relational-expression>=*relational-expression*
  relational-expression<*relational-expression*
  relational-expression<=*relational-expression*

> represents greater than, >= represents greater than or equal to, < represents less than, and <= represents less than or equal to. These operators all return a boolean value corresponding to whether the relation is `true` or `false`. The type of each side of the operator should be either integer or floating point.

3.5.8 Equality Operators

equality-expression :
  relational-expression
  equality-expression == *equality-expression*
  equality-expression != *equality-expression*

The == operator compares the equivalence of the two operands and returns the boolean value `true` if they are equal, `false` if they are not. != does the opposite, `true` if they are unequal, `false` if they are equal. This operator compares the value of the identifier, not the reference for equivalence. The operands can be of any type, but operands of two different types will never be equivalent.

3.5.9 Logical AND Operator

The logical AND operator is grouped left to right.

logical-AND-expression :
  equality-expression
  logical-AND-expression `and` *logical-AND-expression*

The logical *and* operator (`and`) only allows for boolean valued operands. This operator returns the boolean value true if both operands are true and false otherwise.

3.5.10 Logical OR Operator

The logical OR operator is grouped left to right.

logical-OR-expression :
  logical-AND-expression
  logical-OR-expression `or` *logical-OR-expression*

The logical *or* operator (`or`) only allows for boolean valued operands. This operator returns the boolean false if both operands are false and true otherwise.

3.5.11 Assignment Expressions

Assignment operators are grouped right to left.

assignment-expression :
  logical-OR-expression
  identifier=*assignment-expression*

Assignment operators expect a variable identifier on the left and a constant or variable of the same type on the right side.

3.5.12 Function Calls

func-call :
  assignment-expression
  assignment-expression(argument-list$_{opt}$)

A function call consists of a function identifer, followed by parentheses with a possibly empty argument list contained. A copy is made of each object passed to the function, so the value of the original object will remained unchanged. Function declarations are discussed in 3.6.4.

3.5.13 List Initializers

list-initializer :
  func-call
  [*list-intializer-list*]
  [*func-call* `to` *func-call*]

list -initializer-list :
  func-call
  list-intializer-list,func-call

A list initializer generates a list containing a range of values. The first form creates a list containing the values specified in the *list-initializer-list*. Every element of the *list-initializer-list* needs to be of the same type or an exception is thrown at compile time. The second form takes two integer values and returns an inclusive list containing the values from the first integer to the second. The first integer must be smaller than the second integer.

3.5.14 Layout Initializer

layout-initializer :
  list-initializer
  `Layout` *identifier layout-initializer-list*

layout-initializer-list :
  list-initializer
  list-intializer-list,list-initializer

A layout initializer creates an instance of the layout type specified. The type of each layout initializer field needs to match those defined in the layout.

3.5.15 Table Initializer

table-initializer :
  layout-initializer
  `Table` (*full-type*$_{opt}$)

A Table initializer, initializes a table and optionally associates a layout with that table. *full-type* is described in detail in 3.6.1

3.5.16 Expressions

expression :
  table-initializer

3.6 Declarations

3.6.1 Type Specifiers

The different type specifiers available are:

type-specifiers :
`int`
`str`
`float`
`bool`
`Table`

full-type :
type-specifier
type-specifier `List`
`Layout` *identifier*

3.6.2 Variable Declarations

variable-declaration :
full-type declarator

declarator :
identifier
identifier `=` *expr*

When assigning a value in a variable declaration, the type of the identifier in *full-type* must match that of the *expr* which it is assigned.

3.6.3 Layout Declarations

A Layout is a collection of optionally named members of various types.

layout-declaration :
`Layout` *identifier* `=` { *layout-declaration-list* }

A Layout declaration consists of the keyword `Layout` followed by an identifier and then an assignment from a *layout-declaration-list* surrounded by curly braces.

layout-declaration-list :
layout-element
layout-declaration-list, layout-element

layout-element :
full-type `:` $identifier_{opt}$

The *Layout-declaration-list* is a comma-separated list of *Layout-elements* which defines the members of the Layout being declared. If no identifier is provided for an element, it can be accessed using the numeric Layout element reference as described in 3.5.3.2.

An instance of an already created layout is created using similar syntax to the declaration:

Layout creations have a few special rules:

- Layout delcarations are treated as special statements in that they are evaluated out of order versus other statements. For example, you can create layouts at the end of your program, and reference that layout type as though it were created in the beginning. However, if that layout declarations have order respective to each other.
- Layouts are only allowed to be declared on the top level of scoping.

3.6.4 Function Declarations

Function declarations are created along with their definition and have the following format:

function-declaration :
    full-type identifier (*parameter-list*$_{opt}$) { *statement-list* }

The full-type in the beginning of the function declaration specifies what type is returned by that function. The identifier that follows is the name of the function and will be referenced anytime that function should be called.

Then there is a *parameter-list*, i.e. a list of arguments, inside of parentheses.

parameter-list :
    type-specifier identifier
    parameter-list, type-specifier identifier

These arguments must be passed with the function whenever it is called.

After the arguments comes the function definition inside of curly braces. The definition can contain any number statements, expressions, and declarations. The one caveat is the definition must contain a *ret* statement for the return type indicated. If the function does not need to return a value, it is a best practice to return an int as the error code. You can **overload** functions, meaning you can have multiple functions with the same name, so long as they have different signatures.

3.7 Statements

Unless otherwise described, statements are executed in sequence. Statements should be ended by a semi-colon(";"). Statements can be broken up into the following:

statement :
    expression-statement
    return-statement
    statement-block
    conditional-statement
    iterative-statement
    variable-declaration
    layout-declaration
    function-declaration

Statements are separated by newlines and a series of statements will be called a *statement-list*.

statement-list :
    statement
    statement-list `\n` *statement*

3.7.1 Expression Statements

Expressions statements make up the majority or statements:

expression-statements :
    expression

An expression statement is made up of one or more expressions as defined in 3.5. After the entire statement is evaluated and all effects are completed, then the next statement is executed.

3.7.2 Return Statement

return-statements :
    `ret` *expr*

A return statement is included in a function and indicates the value to be returned by that function. This expr needs to have the same type as defined in the function declaration.

3.7.3 Statement Block

statement-block :
{ *statement-list* }

A statement block groups multiple statements together. Any variable declared in a statement block is only in scope until that statement block is closed.

3.7.4 Conditional Statements

Conditional statements control the flow of a program by performing different sets of statements depending on some boolean value.

conditional-statements :
`if (`*expression*`)` *statement elif-list*
`if (`*expression*`)` *statement elif-list* `else` *statement*

elif-list :
`elif (`*expression*`)` *statement*
elif-list `elif (`*expression*`)` *statement*

The expression in the parentheses after `if`, `elif`, and `else` must be boolean-valued. If it is true, execute the corresponding statement and jump out of the conditional expression. If it is false, do not execute the statement and evaluate the next expression after an `elif` or `else`.

3.7.5 Iterative Statements

iterative-statements:
`for` *identifier* `<-` *expression statement*
`while (` *expression* `)` *statement*

3.7.5.1 for loop

The type of the expression following the left arrow (”¡-”) must be a list. A *for* loop executes the *statement* once for each elements in that List.

3.7.6 while loop

The *expression* inside of the parentheses of a while loop must be boolean-valued. The while loop repeatedly executes the *statement-list* as long as the value of the expression is `true`.

3.8 FRY Program

A Fry program is just a collection of statements and and function declarations. A function declaration can occur anywhere in your code and statements do not need to be inside of a function declaration.

3.9 Scope

Scope is handled simply in **FRY**, a variable cannot be referenced outside of the code block it was declared inside. In most cases, this block is denoted by curly braces. One exception is the *elements-of* subsection of a Set-builder statements 3.5.2, the scope for these variables are only inside the Set Builder statement (i.e. inside the square brackets). Any variable delcared outside of any code block is considered a global variable and can be referenced anywhere in the program.

4 Project Plan

4.1 Process

4.1.1 Specification

Specification for the FRY language came from the Language Proposal, and was further refined in the Language Reference Manual. With a bit of back and forth discussion and feedback with Prof. Edwards, the FRY language began to be more consistent and logical. Throughout development, if I realized one of the specifications was extremely difficult or impossible to achieve, I modified the specification to make more sense or be more easily achievable.

4.1.2 Development

Development of the FRY compiler started implementing a large portion of the parser and scanner as defined in the FRY LRM. Once programs were being parsed properly, I began writing the java generation code for some simple programs. This started initially with a simple program which added numbers together, then conditional statements, and so on. During this time, I also worked on the different Java classes as they were necessary. I continued to develop portions of the java generation to compile different programs. Once I had a significant set of programs compiled, I began work on the Semantic Analysis to properly check all of the programs I had developed. After the programs I had compiled were properly semantically analyzed, I started an iterative process of adding new functionality to the compiler (to all parts, Parser, Semantic Analyzer, and Java Code Generation).

4.1.3 Testing

The testing driver script (utils/run_tests.sh) was added early on in the development process, so that tests could be added in easily. Every time a new feature was implemented, I added in a new test which verified that feature was working.

4.2 Programming Style Guide

As the entire compiler was written by myself, the project did not really have a concrete "style guide". Instead, as I became more comfortable coding in OCaml, I refined my style guide. Because of this, there are some inconsistencies in the style of the code throughout. Towards the end, there were a few rules I abided by:

- Functions should include the type of the arguments.
- Each **match**, **if**, and **let** block should begin on a new-line and be indented one level further then it's parent block.

4.3 Project Timeline

The approximate timeline for the project looked something like this:

- **Sep 24** - Project Proposal Finished
- **Oct 25** - LRM Finished
- **Nov 9** - Scanner/Parser implemented (mostly) unambiguous
- **Nov 14** - Simple programs compiled to Java
- **Nov 16** - Testing framework created
- **Dec 14** - Most of the Language set up to implement is completed and implemented in compiler

4.4 Development Environment

The project was developed/built on an Ubuntu 14.04.1 LTS, Trusty Tahr virtual machine on my desktop. The entire project was source-controlled using Git and hosted on github.com. The compiler was written in OCaml and was developed using Sublime Text 3. The Java libraries which implemented the OCaml Data structures and built-in functions were developed in Eclipse. The bash testing driver script as well as all FRY programs written were written in ViM. Make was used to compile all of the OCaml and the Java was compiled automatically by Eclipse.

4.5 Project Log

I have had problems with my Laptop hard drive in the past and I did not want to lose my work, so I commited/-pushed often, as you will notice with the amount of commits. Keep in mind, all documents were tracked in the Git repository as well so there a number of commits that are not code-related.

09cb02e - (HEAD, origin/master, origin/HEAD, master) Final report project process being added (Sun Dec 14 17:02:29 2014 -0500) <Tom DeVoe>

f8b8ce1 - Fixed some issues with List, got Column function working, updated final report (Sun Dec 14 16:00:50 2014 -0500) <tcdevoe>

8773521 - Trying to add getColumn function, worked on Final Report (Sat Dec 13 17:44:19 2014 -0500) <Tom DeVoe>

6bba4ce - Cleaned up Language Reference for addition to Final Report (Wed Dec 10 23:17:19 2014 -0500) <Tom DeVoe>

ee1a423 - Getting rid of shift/reduce conflicts (Tue Dec 9 23:00:34 2014 -0500) <Tom DeVoe>

70150fb - Implemented Set builder notation (Tue Dec 9 16:58:49 2014 -0500) <Tom DeVoe>

64a915d - Implemented FRY Layout and FRY Table java classes (Sat Dec 6 18:38:56 2014 -0500) <Tom DeVoe>

a6a793a - Fixed function args issue, added function overloading (Thu Dec 4 00:29:58 2014 -0500) <Tom DeVoe>

3f48ce5 - Got Layout references working, need to fix functions that take args (Tue Dec 2 23:57:31 2014 -0500) <Tom DeVoe>

cc32d73 - Layout java generation and semantic analysis implemented, need to add reference (Mon Dec 1 23:56:17 2014 -0500) <Tom DeVoe>

fb8df36 - Added in Layout parsing, need to add semantic analysis/javagen support (Mon Dec 1 00:35:43 2014 -0500) <Tom DeVoe>

c04159d - Added in List references (Fri Nov 28 20:08:10 2014 -0500) <Tom DeVoe>

fb88eed - Merge branch 'master' of github.com:tcdevoe/FRY-lang (Tue Nov 25 23:34:20 2014 -0500) <Tom DeVoe>

4535996 - Lists, for loops, funcs, fully implemented/tested, All tests suceeding (Tue Nov 25 23:33:48 2014 -0500) <Tom DeVoe>

899eb57 - Create README.md (Mon Nov 24 23:41:29 2014 -0500) <tcdevoe>

0b86925 - Fixed parsing order - got mostly working check (Mon Nov 24 23:40:05 2014 -0500) <Tom DeVoe>

bdf1ad1 - Fixed path issue in run_tests, fixed Makefile to work with SAST interface (Sat Nov 22 19:33:31 2014 -0500) <Tom DeVoe>

4193161 - Compiling semantic analysis - almost fully implemented (Thu Nov 20 23:24:32 2014 -0500) <Tom DeVoe>

7ff8d13 - Started implementing semantic analysis, added in some more tests and updated run_tests.sh (Tue Nov 18 22:33:33 2014 -0500) <Tom DeVoe>

eee6eb6 - List generation syntax working (Mon Nov 17 23:41:31 2014 -0500) <Tom DeVoe>

adb160d - Updated run_tests script and got If conditionals working (Sun Nov 16 23:19:43 2014 -0500) <Tom DeVoe>

17b6573 - Added initial testing script (Sun Nov 16 18:36:44 2014 -0500) <Tom DeVoe>

6023309 - Added IOUtils and Makefile for java src (Sat Nov 15 19:03:05 2014 -0500) <Tom DeVoe>

dc7fc21 - Simple compile to java (Fri Nov 14 19:57:17 2014 -0500) <Tom DeVoe>

1dcb46e - Reworked the expr grammar, reduced shift/reduce conflicts (Thu Nov 13 23:07:06 2014 -0500) <Tom DeVoe>

7c3de09 - Main Executable added, Parsing-printer added (Mon Nov 10 23:32:28 2014 -0500) <Tom DeVoe>

f4b4109 - Compiling mostly-finished scanner/parser - finishing debug printing (Sun Nov 9 23:32:07 2014 -0500) <Tom DeVoe>

a274640 - Merge branch ’master’ of github.com:tcdevoe/FRY-lang (Sat Nov 8 15:07:53 2014 -0500) <tcdevoe>

3940d51 - Updating Docs Names (Sat Nov 8 15:07:46 2014 -0500) <tcdevoe>

2e1b77b - Working build, parser, scanner, ast made progress (Thu Nov 6 23:10:55 2014 -0500) <Tom DeVoe>

0c1227d - Finished LRM - still need to proofread (Sat Oct 25 22:28:56 2014 -0400) <tcdevoe>

a9c80c8 - Merge branch ’LanguageReference’ (Thu Oct 23 23:55:40 2014 -0400) <tcdevoe>

d0c9368 - (LanguageReference) Finished up the Expressions section and finished half the declarations (Thu Oct 23 23:52:05 2014 -0400) <tcdevoe>

23d3811 - Working gen_tokens.sh (Mon Oct 20 21:04:33 2014 -0400) <Tom DeVoe>

676a021 - Improved Language reference (Mon Oct 20 18:47:43 2014 -0400) <tcdevoe>

90f5191 - Updated Lanaguage Refrence on new branch (Mon Oct 20 17:50:07 2014 -0400) <tcdevoe>

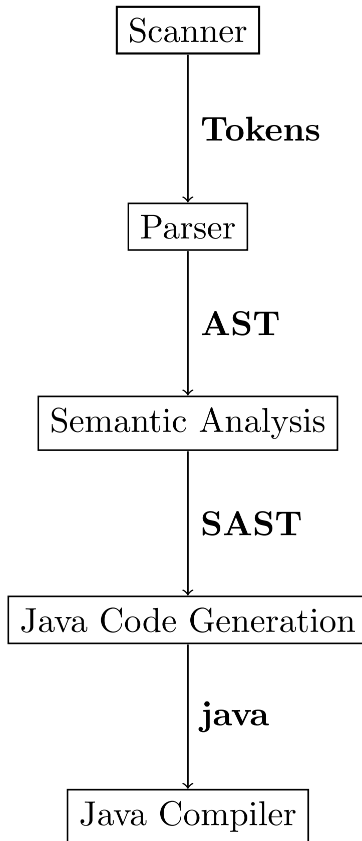
d20a668 - Saved Corrupt Repo (Mon Oct 20 17:47:45 2014 -0400) <Tom DeVoe>

3f8569a - Update (Sun Oct 12 17:33:18 2014 -0400) <tcdevoe>

be27326 - Started adding in the language reference information (Wed Oct 8 23:04:36 2014 -0400) <tcdevoe>

2fad90c - Initial Commit - Commiting Documents (Wed Oct 8 21:47:37 2014 -0400) <tcdevoe>

5 Compiler Architecture



As is shown in the diagram, the FRY compiler is split into 4 sections, the Scanner, Parser, Semantic Analysis and Java Code Generation.

5.1 Scanner

Scans the input program and produces a tokenized output. The Scanner discards all whitespace and comments and reports an error if any invalid character sequences are encountered, such as an invalid identifier or escape sequence. The scanner was written for `ocamlyacc`.

5.2 Parser

Parses the tokenized output of the scanner and generates an **AST (Abstract Syntax Tree)** . The parser checks the syntax and catches any syntax errors. The parser does not check type or semantics. The parser was written for `ocamllex` and the AST was written in `OCaml`.

5.3 Semantic Analysis

The Semantic Analyzer parses the AST and checks the type and semantics of every statement. This produces a **SAST (Semantically-checked Abstract Syntax Tree)**. Each element in the SAST keeps track of its type for use in the java code generation. Any type mismatches or semantic errors will be reported during this step. The Semantic Analyzer and the SAST were written in `OCaml`.

5.4 Java Code Generation

The Java Code Generator parses the SAST and produces corresponding Java code for each statement. At this point the SAST should be fully syntactically and semantically sound, so there should be no errors produced. The java code generator was written in `OCaml`.

6 Test Plan

6.1 Example Programs and Generated Code

6.1.1 Holiday Calendar Example

A more detailed description of this example is included in 2.6.1. **FRY Source Code**

```
# Checks if the date is valid (i.e. no Feb 30th)
bool isValidDate(str mon, int day){
  if ( day > 28 and mon == "Feb"){
    ret false;
  }
  if ( day == 31 ){
    if ( mon == "Sep" or mon == "Apr" or mon == "Jun" or mon == "Nov"){
      ret false;
    }
    else {
      ret true;
    }
  }
  ret true;
}
# Define our file layout
Layout date = {str: mon, int: day, int: year, str: holiday};

# read in the file with the holiday listing
Table holidays = Table (Layout date);
holidays = Read("holidays.txt",",");

str holiday_name;
Table calendar = Table (Layout date);
str List holiday_list;
Table matching_days;

# Generate a calendar, marking the holidays down
for ( mon <- [ "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct",
    "Nov", "Dec" ]){
  for ( day <- [ 1 to 31 ]) {
    if(isValidDate(mon, day)){
        matching_days = [ i | i <- holidays; i.{mon} == mon and i.{day} == day ];
        holiday_list = Column(matching_days, "holiday");
        holiday_name = holiday_list[0];
        Append(calendar, Layout date {mon, day, 2015,holiday_name});
    }
  }
}

Write("holiday-calendar.txt", calendar);
```

Generated Java Code

```
import fry.*;
import java.util.Arrays;
import java.io.IOException;
import java.util.ArrayList;
```

```
public class fry {
 public static class date extends FRYLayout {
   public String mon;
   public Integer day;
   public Integer year;
   public String holiday;
    public date(String mon, Integer day, Integer year, String holiday) {
      super();
     this.holiday = holiday;
     this.year = year;
     this.day = day;
     this.mon = mon;
   }
    public date() {
     super();
   }
    public String toString() {
     return mon.toString() + "|" + day.toString() + "|"
         + year.toString() + "|" + holiday.toString();
   }
 }
  public static Boolean isValidDate(String mon, Integer day) {
   if (day > 28 && mon.equals("Feb")) {
     return false;
   }
    if (day.equals(31)) {
     if (mon.equals("Sep") || mon.equals("Apr") || mon.equals("Jun")
         || mon.equals("Nov")) {
       return false;
     } else {
       return true;
     }
    }
    return true;
 }
  public static void main(String[] args) throws IOException {
   ;
   FRYTable holidays = new FRYTable(new date());
   holidays.readInput(IOUtils.Read("holidays.txt", ","));
   String holiday_name;
   FRYTable calendar = new FRYTable(new date());
   FRYList<String> holiday_list;
   FRYTable matching_days;
   for (String mon : new ArrayList<String>(Arrays.asList(
  new  String[]{("Jan"), ("Feb"), ("Mar"),
  ("Apr"), ("May"), ("Jun"), ("Jul"), ("Aug"),
  ("Sep"), ("Oct"), ("Nov"), ("Dec")}))) {
     {
       for (Integer day : FRYListFactory.getGeneratedFryList(1, 31)) {
         {
           if (isValidDate(mon, day)) {
             FRYList<String[]> __ret_data__ = new FRYList<String[]>(
                 holidays.getData().size());
             for (String[] i : holidays.getData()) {
                if ((((i[holidays.layout.getIdByName("mon")])))
                   .equals(mon)
                   && (new Integer(
```

```
                        Integer.parseInt(i[holidays.layout
                            .getIdByName("day")]))
                        .equals(day)) {
                  __ret_data__.add(i);
                }
              }
              FRYTable __tmp_tbl__ = new FRYTable(__ret_data__,
                  holidays.layout);
              ;
              matching_days = __tmp_tbl__;
              ;
              holiday_list = matching_days.getColumn("holiday");
              holiday_name = holiday_list.get(0);
              IOUtils.Append(calendar, new date(mon, day, 2015,
                  holiday_name));
            }
          }
        }
      }
    }
    IOUtils.Write("holiday-calendar.txt", calendar);
  }
}
```

6.1.2 Filtering Example

A more detailed description of this example is included in 2.6.2. **FRY Source Code**

```
# Define source layout and read source data
Layout weblog = {str: date, str: site, str: user};
Table weblog_data = Table (Layout weblog);
weblog_data = Read("website_hits.txt","|");

# Filter required records
Table frodo_facebook = [i | i <- weblog_data; i.{site} == "facebook.com" and i.{user
    } == "Frodo"];
Table sam_google = [i | i <- weblog_data; i.{site} == "google.com" and i.{user} == "
    Sam"];

# write out filtered records to respective files
Write("frodo_facebook.txt", frodo_facebook);
Write("sam_google.txt", sam_google);
```

Generated Java Code

```
package fry;
import java.util.Arrays;
import java.io.IOException;
import java.util.ArrayList;

public class mainTest {
  public static class weblog extends FRYLayout {
    public String date;
    public String site;
    public String user;

    public weblog(String date, String site, String user) {
```

```
    super();
    this.user = user;
    this.site = site;
    this.date = date;
  }

  public weblog() {
    super();
  }

  public String toString() {
    return date.toString() + "|" + site.toString() + "|"
        + user.toString();
  }
}

public static void main(String[] args) throws IOException {
  ;
  FRYTable weblog_data = new FRYTable(new weblog());
  weblog_data.readInput(IOUtils.Read("website_hits.txt", "|"));
  FRYTable frodo_facebook;
  {
    FRYList<String[]> __ret_data__ = new FRYList<String[]>(weblog_data
        .getData().size());
    for (String[] i : weblog_data.getData()) {

      if ((((i[weblog_data.layout.getIdByName("site")])))
          .equals("facebook.com")
          && (((i[weblog_data.layout.getIdByName("user")])))
              .equals("Frodo")) {
        __ret_data__.add(i);
      }
    }
    FRYTable __tmp_tbl__ = new FRYTable(__ret_data__,
        weblog_data.layout);
    frodo_facebook = __tmp_tbl__;
  }
  FRYTable sam_google;
  {
    FRYList<String[]> __ret_data__ = new FRYList<String[]>(weblog_data
        .getData().size());
    for (String[] i : weblog_data.getData()) {

      if ((((i[weblog_data.layout.getIdByName("site")])))
          .equals("google.com")
          && (((i[weblog_data.layout.getIdByName("user")])))
              .equals("Sam")) {
        __ret_data__.add(i);
      }
    }
    FRYTable __tmp_tbl__ = new FRYTable(__ret_data__,
        weblog_data.layout);
    sam_google = __tmp_tbl__;
  }
  IOUtils.Write("frodo_facebook.txt", frodo_facebook);
  IOUtils.Write("sam_google.txt", sam_google);
```

```
  }
}
```

6.2 Test Suite

All testing performed was functional testing. Every time a new functionality was added to FRY, I added in a few tests to test that functionality. A shell script was written which automated running through these tests. The shell script compiles and executes a specific program, and diffs the output of execution against the expected output. If an error occured in any of the phases (compilation, java compilation, execution), the error would be caught an printed to stdout. The source code for the testing driver script - **run_tests.sh** are located here 8 - and all of the tests are located here 8.

7 Lessons Learned

When I first learned we had to create our own programming language and compiler, it sounded impossible. And in fact, working on this project only reinforced that notion. While I managed somehow to come out of this class with a (somewhat) working compiler for FRY, the amount of effort needed to implement even the most basic compiler was fairly surprising. At the least, it put things in perspective and taught me just how much effort has gone into GCC, the Java Compiler, even the Bash interpreter; and how involved these pieces of software really are. The fact that they work so consistently, that they are able to do all of these optimizations, and still compile thousands of source files quickly is a small miracle.

While I did not get the same experience of people working in teams (which is probably not such a big deal, since I do that at work every day), working on a relatively large scale program on my own definitely taught my a thing or two. I definitely felt as though I needed to rush to complete this compiler. Because of that, from the start I implemented some things in a pretty kludgy fashion, and did plenty of things the quick and dirty way. Unfortunately, that mind set turned out to be somewhat "Penny-wise, Pound-foolish" as I eventually paid for my mostly commentless spaghetti code. Going back and changing certain things became much more painful than it needed to be. I realized it is worthwhile to write clean, maintainable code, even for a relatively short-term project where I was rushing. Had I done that, I may have been able to do *more* over the semester than I accomplished. Something I really came to appreciate was the automated testing. The amount of bugs I introduced would have wreaked havoc had they not been caught by frequent test cycles.

8 Appendix

All modules were built by Tom DeVoe.

Makefile

```
OBJS = ast.cmo SemanticAnalysis.cmo javagen.cmo parser.cmo scanner.cmo fry.cmo

fry : $(OBJS)
 ocamlc -g -o fry $(OBJS)

scanner.ml : scanner.mll
 ocamllex scanner.mll

parser.ml parser.mli : parser.mly
 ocamlyacc -v parser.mly

%.cmo : %.ml
 ocamlc -g -w A -c $<

%.cmi : %.mli
 ocamlc -g -w A -c $<
```

```
#.PHONY : java_make
#java_make :
# $(MAKE) -C java/

.PHONY : clean
clean : java_clean
 rm -f fry parser.ml parser.mli scanner.ml \
   *.cmo *.cmi parser.output

.PHONY : java_clean
java_clean :
 -$(MAKE) -C java/ clean


# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
fry.cmo : scanner.cmo parser.cmi javagen.cmo ast.cmo
fry.cmx : scanner.cmx parser.cmx javagen.cmx ast.cmx
javagen.cmo : sast.cmi ast.cmo
javagen.cmx : sast.cmi ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
SemanticAnalysis.cmo : sast.cmi ast.cmo
SemanticAnalysis.cmx : sast.cmi ast.cmx
parser.cmi : ast.cmo
sast.cmi : ast.cmo
```

ast.ml

```
(* Ref is List/Layout element reference *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | In |
    Notin | And | Or | From
type dataType  = String | Float | Bool | Int | Layout of string | Table | List of
    dataType | Void | Res
type post = Inc | Dec
type pre = Not
type ref = ListRef | LayRef | TblRef

type expr =
    StringLit of string
|   FloatLit of string
|   IntLit of int
|   BoolLit of string
|   ListLit of expr list
|   ListGen of expr * expr  (* List generator which keeps the min and max of the
    generated list *)
|   Id of string
|   Binop of expr * op * expr
|   Postop of expr * post
|   Preop of pre * expr
|   Ref of expr * ref * expr (* ID of object * reference type * index1 * index2 *)
|   Assign of string * expr
|   Call of string * expr list
|  Slice of expr * expr
|   LayoutLit of dataType * expr list
```

```
|   TableInit of dataType
|   SetBuild of expr * string * expr * expr (* [ return-layout | ID <- element-of;
    expression ] *)
|   Noexpr

and var_decl = VarDecl of dataType * expr

type stmt =
    Block of stmt list
|   Expr of expr
|   Return of expr
|   If of (expr * stmt) list * stmt
|   For of expr * expr * stmt (* needs to be a X in Y binop expr *)
|   While of expr * stmt
|  VarDeclS of var_decl
|   LayoutCreation of string * var_decl list


type func_decl = {
    fname : string;
    ret_type : dataType;
    formals : var_decl list;
    body : stmt list;
}

(* stmts includes var decls and statements *)
type program = { stmts: stmt list; funcs: func_decl list }


(* Low-level AST printing for debugging *)

let rec expr_s = function
  StringLit(l) -> "StringLit " ^ l
| FloatLit(l) -> "FloatLit " ^ l
| IntLit(l) -> "IntLit " ^ string_of_int l
| BoolLit(l) ->  "BoolLit " ^ l
| ListGen(e1, e2) -> "ListGen (" ^ expr_s e1 ^ ", "^ expr_s e2 ^ ")"
| ListLit(l) ->  "ListLit ("  ^ String.concat ", " (List.map (fun e -> "(" ^ expr_s
   e ^ ")") l) ^ ")"
| Id(s) -> "ID " ^ s
| Binop(e1, o, e2) -> "Binop (" ^ expr_s e1 ^ ") " ^
 (match o with Add -> "Add" | Sub -> "Sub" | Mult -> "Mult" |
      Div -> "Div" | Equal -> "Equal" | Neq -> "Neq" |
      Less -> "Less" | Leq -> "Leq" | Greater -> "Greater" |
      Geq -> "Geq" | In -> "In" | Notin -> "Notin" | And -> "And" |
      Or -> "Or" | From -> "From") ^ " (" ^ expr_s e2 ^ ")"
| Postop(e1, o) -> "Postop (" ^ expr_s e1 ^ ") " ^
 (match o with Inc -> "Inc" | Dec -> "Dec" )
| Preop(o,e1) -> "Not (" ^ expr_s e1 ^ ") "
| Slice(e1,e2) -> "Slice (" ^ expr_s e1 ^","^expr_s e2 ^")"
| Ref(e1, r, e2) -> "Reference (" ^ expr_s e1 ^ ") " ^
 (match r with ListRef -> "ListRef" | LayRef -> "LayRef") ^
 " (" ^ expr_s e2 ^ ") "
| Assign(v, e) -> "Assign " ^ v ^ " (" ^ expr_s e ^ ") "
| Call(f, es) -> "Call " ^ f ^ " [" ^
  String.concat ", " (List.map (fun e -> "(" ^ expr_s e ^ ")") es) ^ "]"
| TableInit(dataType) -> "TableInit ("^data_type_s dataType^")"
```

```
| LayoutLit(typ, e_list) -> "LayoutLit " ^ data_type_s typ ^ " [" ^ String.concat ",
    " (List.map (fun e -> expr_s e) e_list)
| SetBuild(e1, id, e2, e3) -> "SetBuild (" ^ expr_s e1 ^ ") " ^ id ^ " from (" ^
    expr_s e2 ^ ") (" ^ expr_s e3 ^ ") "
| Noexpr -> "Noexpr"

and data_type_s = function
  String -> "String"
| Float -> "Float"
| Bool -> "Bool"
| Int -> "Int"
| Layout(name) -> "Layout(" ^ name ^")"
| Table -> "Table"
| List(t) ->  data_type_s t ^ " List"


let var_decl_s = function
 VarDecl(d,e) -> "VarDecl (" ^ data_type_s d ^ " " ^ expr_s e ^ ")"

let rec stmt_s = function
  Block(ss) -> "Block [" ^ String.concat ",\n"
          (List.map (fun s -> "(" ^ stmt_s s ^ ")") ss) ^ "]"
| Expr(e) -> "Expr (" ^ expr_s e ^ ") "
| Return(e) -> "Return (" ^ expr_s e ^ ")"
| If(elif_l, s) -> (match elif_l with
     [] -> ""
    |   [x] -> "If ( " ^ expr_s (fst x) ^ ", " ^ stmt_s (snd x) ^ ")\n"
    |   h::t -> "If ( " ^ expr_s (fst h) ^ ", " ^ stmt_s (snd h) ^ ")\n" ^
     String.concat "\n" (List.map (fun tl ->
     "Elif (" ^ expr_s (fst tl) ^ ", " ^ stmt_s (snd tl) ^ ")" ) t) ) ^
    (* Don't really care about empty else for pretty printing, see javagen for
       implementation *)
    "\n Else (" ^ stmt_s s ^ ")"
| For(e1, e2, s) -> "For (" ^ expr_s e1 ^ " in " ^ expr_s e2 ^ ") (" ^ stmt_s s ^ ")
    "
| While(e, s) -> "While (" ^ expr_s e ^ ") (" ^ stmt_s s ^ ") "
| VarDeclS(v) -> var_decl_s v
| LayoutCreation(name, v_list) -> "Layout " ^ name ^ "[" ^ String.concat "," (List.
    map (fun v -> var_decl_s v) v_list) ^ "]\n"


let func_decl_s f =
" { fname = \"" ^ f.fname ^ "\"\n ret_type = \"" ^ data_type_s f.ret_type ^ "\"\n
    formals = [" ^
String.concat ", " (List.map var_decl_s f.formals) ^ "]\n body = [" ^
String.concat "\n" (List.map stmt_s f.body) ^
"]}\n"

(* stmt list is built backwards, need to reverse *)
let program_s prog = "([" ^  String.concat ",\n" (List.rev (List.map stmt_s prog.
    stmts)) ^ "],\n" ^
       "[" ^ String.concat ",\n" (List.map func_decl_s prog.funcs) ^"])"
```

fry.ml

```
open Printf
type action = Raw | Ast | Compile
```

```
let _ =
 let action =
  if Array.length Sys.argv > 1 then
   List.assoc Sys.argv.(1) [ ("-r", Raw);
            ("-c", Compile)]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
  Raw -> print_string (Ast.program_s program)
  |  Compile ->
     let checked_program = SemanticAnalysis.check_prgm program in
     let compiled_program = Javagen.j_prgm checked_program
      in let file = open_out ("fry.java") in
        fprintf file "%s"  compiled_program;
```

javagen.ml

```
open Sast
open Ast

let rec j_prgm (prog: Sast.s_program) =
 "import fry.*;" ^
 "import java.util.Arrays;\n" ^
 "import java.io.IOException;\n" ^
 "import java.util.ArrayList;\n" ^
 "public class fry{\n" ^
 (* Write Layouts as private classes *)
 String.concat "\n" (List.map j_layout prog.syms.layouts) ^
 (* Write function declarations *)
 String.concat "\n" (List.map j_fdecl prog.funcs) ^
 "\n\npublic static void main(String[] args) throws IOException{\n" ^
  String.concat "\n" (List.map j_stmt prog.stmts) ^
  "\n}\n}"

and j_data_type = function
 String  -> "String"
|   Int  -> "int"
| Float -> "float"
|  Bool  -> "boolean"
(* Need to add in java libs to handle Layout/Table *)


and j_obj_data_type (typ: dataType)  = match typ with
 String  -> "String"
|   Int -> "Integer"
|   Float -> "Float"
|   Bool -> "Boolean"
|   List(t) ->  "FRYList<" ^ j_obj_data_type t ^ ">"
| Layout(name)  -> name
|   Table ->  "FRYTable"

and getListType = function
 List(t) -> j_obj_data_type t

(************************************
 STATEMENT HELPER FUNCTIONS
************************************)
```

```
and j_expr = function
 S_StringLit(s) -> "\"" ^ s ^ "\""
|   S_FloatLit(f) -> f
|   S_IntLit(i) -> string_of_int i
|   S_BoolLit(b) -> b
|  S_ListGen(e1, e2) -> "FRYListFactory.getGeneratedFryList(" ^ j_expr e1 ^ "," ^
    j_expr e2 ^ ")"
(* Should determine the data type in the check pass and add it to the constructor *)
|  S_ListLit(l, typ) -> "new FRYList<"  ^ j_obj_data_type typ ^ ">(Arrays.asList(new
      "^ j_obj_data_type typ ^ "[]{" ^ String.concat ", " (List.rev (List.map (fun e
     -> "(" ^ j_expr e ^ ")") l)) ^ "}))"
|   S_Id(x,_) -> x
|   S_Binop(e1, o, e2) -> writeBinOp e1 o e2
|  S_Postop(e1, o) -> j_expr e1 ^
 (match o with Inc -> "++" | Dec -> "--" )
|  S_Preop(o,e1) -> "!" ^ j_expr e1
|   S_Slice(e1, e2) -> if e1 = S_Noexpr then
         "0,"^ j_expr e2
        else if e2 = S_Noexpr then
        j_expr e1
        else
         j_expr e1 ^ "," ^ j_expr e2
|  S_Ref(e1, r, e2, typ, inSetBuild) -> (match r with
        ListRef -> writeListRef e1 r e2 typ
       |  LayRef -> writeLayRef e1 r e2 typ inSetBuild)
|  S_Assign(v, e) ->  writeAssign v e
|   S_LayoutLit(d,t, inSetBuild) -> writeLayoutLit d t inSetBuild
|   S_Call(f, es) -> writeFuncCall f es
|   S_TableInit(typ) -> "new FRYTable( new " ^ j_obj_data_type typ ^ "() )"
| S_SetBuild(e1, id, e2, e3) -> writeSetBuild e1 id e2 e3
| S_Noexpr -> ""

and writeFuncCall f es =
match f with
 | "Write" -> "IOUtils.Write" ^ "(" ^
  String.concat ", " (List.map elemForIO es) ^ ")"
 | "Read" -> "IOUtils.Read" ^ "(" ^
  String.concat ", " (List.map elemForIO es) ^ ")"
 | "Append" -> "IOUtils.Append" ^ "(" ^
   String.concat ", " (List.map j_expr es) ^ ")"
 | "Column" -> (match es with [tbl; col_name] -> j_expr tbl ^ ".getColumn("^j_expr
    col_name^")")
 | _ -> f ^ "(" ^
  String.concat ", " (List.map (fun e -> j_expr e ) es) ^ ")"

and writeLayoutLit d t inSetBuild =
 if inSetBuild = "NOT" then
  "new " ^ j_obj_data_type d ^ " (" ^ String.concat "," (List.rev (List.map j_expr t
     )) ^ ")"
 else
  "new String[]{" ^ String.concat "," (List.rev (List.map j_expr t)) ^ "}"

and writeSetBuild e1 id e2 e3 =  "FRYList<String[]>  __ret_data__ = new FRYList<
   String[]>("^j_expr e2^".getData().size());\nfor(String[] "^id^" : " ^ j_expr e2
   ^ ".getData()){\n"^ "\nif("^ j_expr e3 ^ "){ __ret_data__.add("^j_expr e1^");}\n
   }FRYTable __tmp_tbl__  = new FRYTable(__ret_data__,"^j_expr e2^".layout);"
```

```
and writeAssign v e = match e with
 S_Call(f, es) ->
  (match f with
   "Read" -> v ^ ".readInput("^ j_expr e ^")"
  |  _ ->  v ^ " = " ^ j_expr e)
|   S_SetBuild(_,_,_,_) -> "{" ^ j_expr e ^";\n" ^ v ^ " = __tmp_tbl__;}"
|   _ -> v ^ " = " ^ j_expr e

and elemForIO e = match e with S_Id("stdout", String) -> "IOUtils.stdout"
       |  S_Id("stderr", String) -> "IOUtils.stderr"
       |   S_Id("stdin", String) -> "IOUtils.stdin"
       |  _ -> j_expr e

and writeForLoop e1 e2 s = match e1 with
 S_Id(name, typ) -> "for (" ^ getListType typ ^ " " ^ j_expr e1 ^ ": " ^ j_expr e2 ^
     ") {\n" ^ j_stmt s ^"}"

and writeIf (elif_l:(s_expr * s_stmt) list) (else_stmt: s_stmt) = (match elif_l with
     [] -> ""
    |   [x] -> "if ( " ^ j_expr (fst x) ^ ")\n" ^ j_stmt (snd x) ^ "\n"
    |   h::t -> "if ( " ^ j_expr (fst h) ^ ")\n" ^ j_stmt (snd h) ^ "\n" ^
     String.concat "\n" (List.map (fun tl ->
     "else if (" ^ j_expr (fst tl) ^ " )\n" ^ j_stmt (snd tl) ^ "\n" ) t) ) ^
    (match else_stmt with
     S_Block(_,[]) -> ""
    |  _ -> "else\n" ^ j_stmt else_stmt ^ "\n")

and writeWhileLoop (e: s_expr) (s: s_stmt) =
 "while(" ^ j_expr e ^ ") { \n" ^
  j_stmt s ^
 "\n}"

and writeVarDecl = function
 S_BasicDecl(d, e) -> (match e with S_Assign(v,e') ->
       (match e' with
         S_SetBuild(_,_,_,_) -> j_obj_data_type d ^ " " ^ v ^ ";{" ^ j_expr e' ^"\n"
              ^ v ^ " = __tmp_tbl__;}"
       | _ -> j_obj_data_type d ^ " " ^ j_expr e ^ ";")
      | _ -> j_obj_data_type d ^ " " ^ j_expr e ^ ";")
|   S_ListDecl(d, e) ->  j_obj_data_type d ^ " " ^ j_expr e ^ ";"
|   S_LayoutDecl(d, e) -> j_obj_data_type d ^ " " ^ j_expr e ^ ";"

and writeBinOp e1 o e2 =
 match o with
 Equal -> j_expr e1 ^".equals("^j_expr e2^") "
 | Neq -> "!" ^ j_expr e1 ^".equals("^j_expr e2^") "
 | _ -> j_expr e1 ^
  (match o with Add -> "+" | Sub -> "-" | Mult -> "*" |
      Div -> "/" |
      Less -> "<" | Leq -> "<=" | Greater -> ">" |
      Geq -> ">=" (* |  In -> "In" | Notin -> "Notin" | From -> "From" *)
      | And -> "&&" | Or -> "||" ) ^ j_expr e2

and writeListRef e1 r e2 typ = match e2 with
       S_Slice(es1, es2) ->"new FRYList<"  ^ j_obj_data_type typ ^ ">(" ^
```

```
             j_expr e1 ^ ".subList(" ^
            (match es2 with
              S_Noexpr -> j_expr e2 ^ ", " ^ j_expr e1 ^ ".size()"
            | _ -> j_expr e2 )
            ^ "))"
     |   _ -> j_expr e1 ^".get(" ^ j_expr e2 ^")"

and writeLayRef e1 r e2 typ inSetBuild =
 if inSetBuild = "NOT" then
  j_expr e1 ^ "." ^ j_expr e2
 else
 (match typ with
  Int ->  "(new Integer(Integer.parseInt("
 |   Float -> "(new Float(Float.parseFloat("
 |   Bool -> "(new Boolean(Boolean.parseBoolean("
 |   _ -> "(((" )
   ^ j_expr e1 ^"["^inSetBuild^".layout.getIdByName(\""^j_expr e2^"\")])))"

and j_stmt = function
 S_Block(_,ss) -> "{\n" ^ String.concat "\n" ( List.rev (List.map j_stmt ss)) ^ "\n}
    "
|  S_Expr(e,_) -> j_expr e ^ ";"
|   S_Return(e,_) -> "return " ^ j_expr e ^";"
|  S_If(elif_l, s) -> writeIf elif_l s
|  S_For(e1, e2, s) -> writeForLoop e1 e2 s
|  S_While(e, s) -> writeWhileLoop e s
|  S_VarDecl(v) -> writeVarDecl v

and j_fdecl (f: s_func_decl) = "public static " ^ j_obj_data_type f.ret_type ^ " " ^
    f.fname ^
          "(" ^ String.concat "," (List.map writeFormal f.formals) ^") {\n" ^
          String.concat "\n" (List.map j_stmt f.body) ^ "}"

and writeFormal (v: s_var_decl) = match v with
 S_BasicDecl(d, e) -> j_obj_data_type d ^ " " ^ j_expr e
|   S_ListDecl(d, e) -> j_obj_data_type d ^ " " ^ j_expr e
|   S_LayoutDecl(d, e) -> j_obj_data_type d ^ " " ^ j_expr e

and j_layout (layout: string * s_var_decl list) =
 let (name, v_decs) = layout in
 "public static class " ^ name ^ " extends FRYLayout{\n public " ^ String.concat "\
    npublic " (List.rev (List.map writeVarDecl v_decs)) ^
 j_layout_constructor v_decs name ^
 j_toString v_decs ^
 "}"

and j_layout_constructor (v_decs: s_var_decl list) (name: string) =
"\npublic " ^ name ^ "(" ^ String.concat "," (List.rev (List.map writeFormal v_decs)
   ) ^ ")" ^
"{\n\n super();\n" ^ String.concat ";\n" (List.map (fun v_dec -> let v_name = (match
    v_dec with
 S_BasicDecl(_, e)
|   S_ListDecl(_, e)
|   S_LayoutDecl(_, e) -> j_expr e) in "this." ^ v_name ^ "=" ^ v_name) v_decs) ^
";}\n" ^
"public " ^ name ^ "(){\nsuper();}"
```

```
and j_toString (v_decs: s_var_decl list) =
"\npublic String toString(){\nreturn " ^
String.concat "+\"|\"+" (List.rev (List.map (fun v_dec -> let vname = (match v_dec
   with
 S_BasicDecl(_, e)
|   S_ListDecl(_, e)
|   S_LayoutDecl(_, e) -> j_expr e) in vname ^".toString()") v_decs))
^ ";\n}"
```

parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token BAR SEMI COMMA PLUS MINUS TIMES
%token DIVIDE ASSIGN EQ NEQ LT LEQ
%token GT GEQ INT STRING FLOAT BOOL
%token LAYOUT LIST TABLE IN NOT FROM
%token IF ELSE ELIF AND OR CONT BREAK
%token INC DEC PERIOD COLON
%token RETURN FOR WHILE TO
%token <int> INT_LIT
%token <string> FLOAT_LIT BOOL_LIT STRING_LIT ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left LBRACK
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left IN NOT_IN
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT
%left INC DEC

%start program
%type <Ast.program> program

%%

/********************
 EXPRESSIONS
********************/

literal:
    STRING_LIT      { StringLit($1) }
|   FLOAT_LIT       { FloatLit($1) }
|   INT_LIT         { IntLit($1) }
|   BOOL_LIT        { BoolLit($1) }

primary_expr:
|   ID              { Id($1) }
| literal   { $1 }
| LPAREN expr RPAREN { $2 }
```

```
set_build_expr:
 primary_expr { $1 }
| LBRACK expr BAR ID FROM set_build_expr SEMI expr RBRACK { SetBuild($2,$4,$6,$8) }

postfix_expr:
 set_build_expr { $1 }
|   postfix_expr INC        { Postop($1, Inc) }
|   postfix_expr DEC        { Postop($1, Dec) }
|   postfix_expr LBRACK slice_opt RBRACK { Ref($1, ListRef, $3) }
|   postfix_expr PERIOD LBRACE expr RBRACE { Ref($1, LayRef, $4) }


prefix_expr:
    postfix_expr { $1 }
|   NOT prefix_expr      { Preop(Not, $2) }

multi_expr:
 prefix_expr  { $1 }
| multi_expr TIMES multi_expr { Binop($1, Mult,$3) }
|   multi_expr DIVIDE multi_expr { Binop($1, Div, $3) }

add_expr:
 multi_expr { $1 }
|   add_expr PLUS add_expr  { Binop($1, Add, $3) }
|   add_expr MINUS add_expr { Binop($1, Sub, $3) }

relational_expr:
 add_expr  { $1 }
|   relational_expr LT relational_expr    { Binop($1, Less, $3) }
|   relational_expr LEQ relational_expr   { Binop($1, Leq, $3) }
|   relational_expr GT relational_expr    { Binop($1, Greater, $3) }
|   relational_expr GEQ relational_expr   { Binop($1, Geq, $3) }

equality_expr:
 relational_expr { $1 }
|   equality_expr EQ equality_expr    { Binop($1, Equal, $3) }
|   equality_expr NEQ equality_expr   { Binop($1, Neq, $3) }

logical_AND_expr:
 equality_expr { $1 }
| logical_AND_expr AND logical_AND_expr   { Binop($1, And, $3) }

logical_OR_expr:
 logical_AND_expr { $1 }
|   logical_AND_expr OR logical_OR_expr    { Binop($1, Or, $3) }

assign_expr:
 logical_OR_expr { $1 }
|   ID ASSIGN expr { Assign($1, $3) }

func_call:
 assign_expr { $1 }
|  ID LPAREN actuals_opt RPAREN { Call($1, $3) }

list_initializer:
 func_call  { $1 }
```

```
| LBRACK list_initializer_list RBRACK { ListLit($2) }
| LBRACK func_call TO func_call RBRACK { ListGen($2, $4) }

list_initializer_list:
 func_call  { [$1] }
| list_initializer_list COMMA func_call { $3::$1 }

layout_lit:
 list_initializer     { $1 }
| LAYOUT ID LBRACE layout_lit_list RBRACE { LayoutLit(Layout($2), $4) }

layout_lit_list:
 layout_lit { [$1] }
| layout_lit_list COMMA layout_lit { $3::$1 }

table_initializer:
 layout_lit { $1 }
|   TABLE LPAREN full_type RPAREN { TableInit($3) }

expr:
 table_initializer { $1 }



/**************************
  DECLARATIONS
**************************/

type_spec:
 STRING    { String }
| INT    { Int }
| FLOAT    { Float }
|  BOOL    { Bool }
| TABLE   { Table }

full_type:
 type_spec    { $1 }
| type_spec LIST   { List($1) }
|   LAYOUT ID    { Layout($2) }

declarator:
 ID   { Id($1) }
| ID ASSIGN expr  { Assign($1, $3) }

vdecl:
    full_type declarator  { VarDecl($1, $2) }

layout_creation:
 LAYOUT ID ASSIGN LBRACE layout_creation_list RBRACE { LayoutCreation($2, $5) }

layout_creation_list:
 layout_type_spec {[$1]}
| layout_creation_list COMMA layout_type_spec { $3::$1 }

layout_type_spec:
 full_type COLON ID { VarDecl($1,Id($3)) }
```

```
program:
    /* nothing */ { { stmts = []; funcs = [] } }
    /* List is built backwards */
|   program fdecl { { stmts = $1.stmts; funcs = $2::$1.funcs } }
| program stmt  { { stmts = $2::$1.stmts; funcs = $1.funcs } }

fdecl:
    full_type ID LPAREN param_list RPAREN LBRACE stmt_list RBRACE
     {{
      fname = $2;
      ret_type = $1;
      formals = $4;
      body = List.rev $7;
     }}

param_list:
 /* nothing */ { [] }
| full_type ID { [VarDecl($1, Id($2))] }
| param_list COMMA full_type ID { VarDecl($3, Id($4))::$1}

stmt:
 expr SEMI { Expr($1) }
| RETURN expr SEMI { Return($2) }
| LBRACE stmt_list RBRACE { Block(List.rev $2) }
| conditional_stmt  { $1 }
| FOR LPAREN expr FROM expr RPAREN stmt { For($3, $5, $7) }
| WHILE expr stmt { While($2, $3) }
| vdecl SEMI   { VarDeclS($1) }
| layout_creation SEMI { $1 }

conditional_stmt:
 IF LPAREN expr RPAREN stmt elif_list %prec NOELSE { If(($3,$5)::$6, Block([])) }
| IF LPAREN expr RPAREN stmt elif_list ELSE stmt { If(($3,$5)::$6, $8) }

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

elif_list:
 /* nothing */ { [] }
| elif_list ELIF LPAREN expr RPAREN stmt { ($4, $6)::$1 }


slice_opt:
|   COLON expr    { Slice(Noexpr, $2) }
| expr COLON   { Slice($1, Noexpr) }
| expr COLON expr  { Slice($1, $3) }
| expr        { $1 }

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

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

sast.mli

```
open Ast

type s_expr =
    S_StringLit of string
|   S_FloatLit of string
|   S_IntLit of int
|   S_BoolLit of string
|   S_ListLit of s_expr list * dataType
|   S_ListGen of s_expr * s_expr  (* List generator which keeps the min and max of
   the generated list *)
|   S_Id of string * dataType
|   S_Binop of s_expr * op * s_expr
|   S_Postop of s_expr * post
|   S_Preop of pre * s_expr
|   S_Ref of s_expr * ref * s_expr * dataType * string (* ID of object * reference
   type * access expression * type *)
|   S_Slice of s_expr * s_expr
|   S_Assign of string * s_expr
|   S_Call of string * s_expr list
|   S_LayoutLit of dataType * s_expr list * string
|   S_TableInit of dataType
|   S_SetBuild of s_expr * string * s_expr * s_expr (* [ return-layout | ID <-
   element-of; expression ] *)
|   S_Noexpr

type symbol_table = {
 (* Parent symbol table included so variables are included in child scope *)
 parent: symbol_table option;
 mutable variables: (dataType * string * s_expr) list;
 (* Keeps track of tables' associated layouts *)
 mutable tables: (dataType * string) list;
 (* Tracks the layout name and its constituent values *)
 mutable layouts: (string * s_var_decl list) list;
}

and s_var_decl =
 S_BasicDecl of dataType * s_expr
|   S_ListDecl of dataType * s_expr
|   S_LayoutDecl of dataType * s_expr

type s_stmt =
    S_Block of symbol_table * s_stmt list
|   S_Expr of s_expr * dataType
|   S_Return of s_expr * dataType
|   S_If of (s_expr * s_stmt) list * s_stmt
|   S_For of s_expr * s_expr * s_stmt
|   S_While of s_expr * s_stmt
|  S_VarDecl of s_var_decl


type s_func_decl = {
    fname : string;
    ret_type : dataType;
    formals : s_var_decl list;
    body : s_stmt list;
```

```
}

type translation_environment = {
 mutable funcs: s_func_decl list;
 scope: symbol_table;
 return_type: dataType;
 in_func: bool;
 inSetBuild: string;
}


type s_program = { stmts: s_stmt list; funcs: s_func_decl list; syms: symbol_table }
```

scanner.mll

```
{ open Parser }

let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let identifier = letter (letter | digit | '_')*

rule token = parse
    [' ' '\t' '\r' '\n' ] { token lexbuf } (* Whitespace *)
|   "/#"    { comment lexbuf }
|    '#'     { sl_comment lexbuf }
|   '('     { LPAREN }
|   ')'     { RPAREN }
|   '{'     { LBRACE }
|   '}'     { RBRACE }
| ']'  { RBRACK }
| '['  { LBRACK }
| '|'  { BAR }
|   ';'     { SEMI }
|    ':'     { COLON }
|   ','     { COMMA }
|   '+'     { PLUS }
|   '-'     { MINUS }
|   '*'     { TIMES }
|   '/'     { DIVIDE }
|   '='     { ASSIGN }
|    '.'     { PERIOD }
|    "++"    { INC }
|    "--"    { DEC }
| "=="    { EQ }
| "!="    { NEQ }
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| "int"   { INT }
| "str"   { STRING }
| "float" { FLOAT }
| "bool"  { BOOL }
| "Layout" { LAYOUT }
| "List" { LIST }
| "Table" { TABLE }
| "in" { IN }
|    "if"    { IF }
```

```
|    "else"   { ELSE }
|    "elif"   { ELIF }
|    "and"    { AND   }
|    "or"     { OR }
|  "for" { FOR }
|  "to" { TO }
|  "while" { WHILE }
|    "continue"   { CONT }
|    "break"      { BREAK }
|    "not"    { NOT }
|    "<-"     { FROM }
|  "ret" { RETURN }
|    (digit)+ as lxm { INT_LIT(int_of_string lxm) }
|    ((digit)+'.'|(digit)*'.'(digit)+) as lxm { FLOAT_LIT(lxm) }
|    '"' (( '\\' _    | [^'"'] )* as lxm)'"' { STRING_LIT(lxm) }
|    "true" | "false" as lxm { BOOL_LIT(lxm) }
|    identifier as lxm { ID(lxm) }
|  eof { EOF }
|    _ as char { raise (Failure("Illegal Character found :" ^ Char.escaped char)) }


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

and sl_comment = parse
    "\n" { token lexbuf}
|    _    { sl_comment lexbuf }
```

SemanticAnalysis.ml

```
(****************************************
check.ml

Purpose:
 Semantic Analysis of a program
 and generate a SAST
****************************************)

open Ast
open Sast
exception Error of string

let print_sym_tbl (syms: symbol_table) = let str = "SYMBOL TABLE: \n[ Variables :" ^
     String.concat "\n" (List.map (fun (typ, name, _) -> "[" ^ Ast.data_type_s typ ^
     " " ^ name ^ "]") syms.variables) ^ "\n" ^
                 " Layouts :" ^ String.concat "\n" (List.map (fun (name,_) -> "[
                   Layout " ^ name ^ "]") syms.layouts) ^ "]" ^
                 " Tables :" ^ String.concat "\n" (List.map (fun (typ,name) -> "[
                   Table " ^ Ast.data_type_s typ ^ " " ^ name ^ "]") syms.tables)
                   ^ "]"
           in print_endline str

let print_funcs (f_decs: s_func_decl list) = print_endline (String.concat "\n" (List
   .map
 (fun f_dec -> "FUNC : " ^ f_dec.fname ^"\nreturn type : "^ data_type_s f_dec.
    ret_type) f_decs))
```

```
let rec find_variable (scope: symbol_table) name =
 try
  List.find (fun (_, s, _) -> s = name ) scope.variables
 with Not_found ->
  match scope.parent with
  Some(parent) -> find_variable parent name
  | _ -> raise (Error("Unrecognized identifier " ^ name))

let rec find_layout (scope: symbol_table) name =
 try
  List.find (fun (s, _) -> s = name ) scope.layouts
 with Not_found ->
  match scope.parent with
  Some(parent) -> find_layout parent name
  | _ -> raise (Error("Unrecognized layout type " ^ name))

let rec find_table (scope: symbol_table) name =
 try
  List.find (fun (_, s) -> s = name ) scope.tables
 with Not_found ->
  match scope.parent with
  Some(parent) -> find_table parent name
  (* Should return a "Generic Layout" if the table is not found *)
  | _ -> raise (Error("Unrecognized table " ^ name))


let rec find_func (funcs: s_func_decl list) fname =
 try
 (* Return all functions with that name, for overloading *)
  let funcs' = List.filter (fun fn -> fn.fname = fname ) funcs in
   funcs'
 with Not_found -> raise (Error("Unrecognized Function call " ^ fname))

let get_math_binop (t1: dataType) (t2: dataType) (env: translation_environment) =
 match (t1, t2) with
  (Int, Float) ->  Float
 | (Float, Int) -> Float
 | (Int, Int) ->  Int
 | (Float, Float) ->  Float
 |   (_,_) ->  raise (Error("Incompatible types for math operator"))

let get_equal_binop (t1: dataType) (t2: dataType) (env: translation_environment) =
 match (t1, t2) with
  (Int, Float) ->  Bool
 | (Float, Int) -> Bool
 | (Int, Int) ->  Bool
 | (Float, Float) ->  Bool
 |   (String, String) -> Bool
 |   (Bool, Bool) -> Bool
 |   (_,_) ->  raise (Error("Incompatible types for equality operator"))

let get_ineq_binop (t1: dataType) (t2: dataType) (env: translation_environment) =
 match (t1, t2) with
  (Int, Float) ->  Bool
 | (Float, Int) -> Bool
 | (Int, Int) ->  Bool
```

```
 | (Float, Float) ->  Bool
 |   (_,_) ->  raise (Error("Incompatible types for inequality operator"))

let get_contain_binop (t1: dataType) (t2: dataType) (env: translation_environment) =
 match (t1, t2) with
  (Int, Int) ->  Bool
 | (Float, Float) -> Bool
 |   (String, String) -> Bool
 |   (Bool, Bool) -> Bool
 |   (_,_) ->  raise (Error("Incompatible types for containment operator"))

let get_logic_binop (t1: dataType) (t2: dataType) (env: translation_environment) =
 match (t1, t2) with
       (Bool, Bool) -> Bool
 |   (_,_) ->  raise (Error("Incompatible types for logical operator"))

let rec check_expr (e: expr) (env: translation_environment) : (Sast.s_expr * Ast.
    dataType)  =
match e with
 StringLit(l) -> S_StringLit(l), String
|  FloatLit(l) ->  S_FloatLit(l), Float
|  IntLit(l) -> S_IntLit(l), Int
|  BoolLit(l) ->  S_BoolLit(l), Bool
|   Id(x) ->
 let vdecl = try
  find_variable env.scope x
 with Not_found ->
  raise (Error("Undeclared identifier " ^ x ))
 in
 let (typ,vname,_) = vdecl in
  S_Id(vname,typ), typ
(*  need to check both expressions are integers for List generator *)
|  ListGen(e1, e2) -> let (e,t) = check_list_gen e1 e2 env in e, List(t)
(* Need to check that every expression in the list has the same type as the first
    expr *)
|  ListLit(l) ->  let (e,t) = check_list_lit l env in e, List(t)
|  Binop(e1, o, e2) -> get_Binop_return e1 o e2 env
|  Postop(e1, o) -> get_Postop_return e1 o env
|  Preop(o,e1) -> get_Preop_return e1 o env
|   Slice(e1, e2) -> check_slice e1 e2 env
|  Ref(e1, r, e2) -> get_ref_return e1 r e2 env
|  Assign(v, e) -> check_assign v e env
|  Call(f, es) -> check_call f es env
|   LayoutLit(typ, e_list) -> check_layout_lit typ e_list env
|   TableInit(typ) -> (match typ with Layout(typ) -> S_TableInit(Layout(typ)), Table
     | _ -> raise (Error ("Table initializer must have layout type")))
|  SetBuild(e1, id, e2, e3) -> check_set_build e1 id e2 e3 env
|  Noexpr -> S_Noexpr, Void

and get_Binop_return (e1: expr) (o: op) (e2: expr) (env: translation_environment) :
    (Sast.s_expr * Ast.dataType) =
 (* Check children expr *)
 let (e1, t1) = check_expr e1 env
 and (e2, t2) = check_expr e2 env in

 match o with
 Add -> (match (t1, t2) with
```

```
   (String, String) -> S_Binop(e1, o, e2), String
  | (Int, Float) -> S_Binop(e1, o, e2), Float
  | (Float, Int) -> S_Binop(e1, o, e2), Float
  | (Int, Int) -> S_Binop(e1, o, e2), Int
  | (Float, Float) -> S_Binop(e1, o, e2), Float
  |   (String, Int) -> S_Binop(e1, o, e2), String
  |   (Int, String) -> S_Binop(e1, o, e2), String
  |   (String, Float) -> S_Binop(e1, o, e2), String
  |   (Float, String) -> S_Binop(e1, o, e2), String
  | _ -> raise (Error("Incompatible types for addition operator.")))
 |  Sub -> S_Binop(e1, o, e2), get_math_binop t1 t2 env
 |  Mult -> S_Binop(e1, o, e2), get_math_binop t1 t2 env
 |  Div -> S_Binop(e1, o, e2), get_math_binop t1 t2 env
 |  Equal -> S_Binop(e1, o, e2), get_equal_binop t1 t2 env
 |  Neq -> S_Binop(e1, o, e2), get_equal_binop t1 t2 env
 |  Less -> S_Binop(e1, o, e2), get_ineq_binop t1 t2  env
 |  Leq -> S_Binop(e1, o, e2), get_ineq_binop t1 t2   env
 |  Greater -> S_Binop(e1, o, e2), get_ineq_binop t1 t2  env
 |  Geq -> S_Binop(e1, o, e2), get_ineq_binop t1 t2  env
 |  In -> S_Binop(e1, o, e2), get_contain_binop t1 t2  env
 |  Notin -> S_Binop(e1, o, e2), get_contain_binop t1 t2  env
 |  And -> S_Binop(e1, o, e2), get_logic_binop t1 t2  env
 |  Or -> S_Binop(e1, o, e2), get_logic_binop t1 t2  env
  |  From -> S_Binop(e1, o, e2), Table

and get_Postop_return (e: expr) (o: post) (env: translation_environment) : (Sast.
    s_expr * Ast.dataType)  =
 let (e, t) = check_expr e env in

 if t = Int then
  S_Postop(e,o), Int
 else
  raise (Error("Expression must have integer type for postfix operator."))

and get_Preop_return (e: expr) (o: pre) (env: translation_environment)  : (Sast.
    s_expr * Ast.dataType) =
 let (e, t) = check_expr e env in

 if t = Bool then
  S_Preop(o, e), Bool
 else
  raise (Error("Expression must have boolean type for Not operator."))

and check_assign (v: string) (e: expr) (env: translation_environment) : (Sast.s_expr
     * Ast.dataType) =
 let (t1,_,_) = find_variable env.scope v
 and (e, t2) = check_expr e env in
 if t1 = t2 then
  S_Assign(v, e), t2
 else
  raise (Error("Variable and assignment must have same type"))

and check_list_gen (e1: expr) (e2: expr) (env: translation_environment) =
 let (e1, t1) = check_expr e1 env in
 let (e2, t2) = check_expr e2 env in
 if t1 = Int && t2 = Int then
  S_ListGen(e1, e2), Int
```

```
 else
     raise (Error("List generator must have integer valued ranges."))

and check_list_lit (l: expr list) (env: translation_environment)  =
 let e = List.hd l in
 let (_,t) = check_expr e env in
 let s_l = List.map (fun ex -> let (e',t') = check_expr ex env in
     if t = t' then
      e'
     else
      raise (Error("Elements of a list literal must all be of the same type"))) l
 in S_ListLit(s_l, t), t

and check_call (f: string) (es: expr list) (env: translation_environment) =
 let sexpr_l_typ = List.map (fun e -> check_expr e env) es in
 let ret_funcs = find_func env.funcs f in
 (* Need to loop through all functions of name f to see if any signatures map caller
     *)
 match ret_funcs with
 [] -> raise (Error("No functions named " ^ f))
|   _ ->
 let (sexpr_l, fdec) = find_func_signature f sexpr_l_typ ret_funcs in
  S_Call(f, sexpr_l), fdec.ret_type

and find_func_signature (f: string) (opts: (s_expr * dataType) list) (ret_funcs:
   s_func_decl list) =
 (* Check function opts *)
 try
  match ret_funcs with
      [] -> raise (Error("No signature matches for function call "^f))
     | hd::tl -> let forms = hd.formals in
         let sexpr = List.map2 (fun (opt: s_expr * dataType) (form: s_var_decl) ->
         let opt_typ = snd opt in
         let form_typ = (match form with
              S_BasicDecl(d,_)
             |   S_ListDecl(d,_)
             |   S_LayoutDecl(d,_) -> d) in
          if opt_typ = form_typ then
           fst opt
          else
           S_Noexpr) opts forms in
         let matched = List.exists (fun e -> e = S_Noexpr) sexpr in
         if matched then
          find_func_signature f opts tl
         else
          sexpr , hd
 with Invalid_argument(x) ->
  raise (Error("Incorrect number of arguments in call to function "^f))

and get_ref_return (e1: expr) (r: ref) (e2: expr) (env: translation_environment) =
 let (e1,typ1) = check_expr e1 env in
 match typ1 with
 List(t) ->  get_list_ref_return e1 r e2 env t
|   Layout(name) -> get_layout_ref_return e1 r e2 env name
|   _ -> raise (Error("Must reference a type List or Layout"))
```

```
and get_list_ref_return (e1: s_expr) (r: ref) (e2: expr) (env:
    translation_environment) (t: dataType) =
 let (e2, typ2) = check_expr e2 env in
 if typ2 = Int then
  S_Ref(e1,r,e2,t,env.inSetBuild), t
   else
  raise (Error("List reference index must be integer valued"))

and get_layout_ref_return (e1: s_expr) (r: ref) (e2: expr) (env:
    translation_environment) (name: string) =
 let (_, mems) = find_layout env.scope name in
 (* Need to add layout members to temporary scope *)
 let env' = { env with scope = {env.scope with variables = []}} in
 List.iter (fun v_dec -> add_layout_mems v_dec env') mems;
 let (e2, typ2) = check_expr e2 env' in
 match e2 with
 (* Can be either a member name, or a numeric value *)
 S_Id(x, _) -> S_Ref(e1, r, e2, typ2, env.inSetBuild), typ2
|    _ -> if typ2 = Int then
   S_Ref(e1, r, e2, typ2, env.inSetBuild), typ2
   else
    raise (Error("Layout reference must either be an element name or an index"))

and add_layout_mems (v: s_var_decl) (env: translation_environment) =
 match v with
  S_ListDecl(d,e) ->  (match e with
         S_Id(x,_) -> env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
        |   _ -> raise (Error ("Should not be reachable")))
 |   S_LayoutDecl(d,e) -> (match e with
         S_Id(x,_) -> env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
        |   _ -> raise (Error ("Should not be reachable")))
 |   S_BasicDecl(d,e) -> (match e with
         S_Id(x,_) -> env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
        |   _ -> raise (Error ("Should not be reachable")))

and check_slice (e1: expr) (e2: expr) (env: translation_environment) =
 let (e1, typ1) = check_expr e1 env
 and (e2, typ2) = check_expr e2 env in
 if typ1 = Int && typ2 = Int then
  S_Slice(e1,e2), Int
 else if e1 = S_Noexpr && e2 = S_Noexpr then
  raise (Error("Slice must have at least one valid index"))
 else if e1 = S_Noexpr || e2 = S_Noexpr then
  S_Slice(e1,e2), Int
 else
  raise (Error("Slice indexes must be integers"))

and check_layout_lit (typ: dataType) (e_list: expr list) (env:
    translation_environment) =
 (* Need to check whether every expr in e_list has the same type as the
     corresponding types in typ *)
 match typ with
  Layout(name) -> let (_,lay_var) = find_layout env.scope name in
      let lay_d_list = List.map (fun e' -> match e' with
               S_BasicDecl(t, _) -> t
               |   S_ListDecl(t, _) -> t
               |   S_LayoutDecl(t, _) -> t) lay_var in
```

```
      let lit_list = List.map (fun e' -> check_expr e' env) e_list in
      let lit_e_list = List.map (fun e' -> let (e,_) = e' in e) lit_list in
      let lit_d_list = List.map (fun e' -> let (_,t) = e' in t) lit_list in
      let correct = compare_lists lay_d_list lit_d_list in
      if correct then
       S_LayoutLit(Layout(name), lit_e_list, env.inSetBuild), Layout(name)
      else
       raise (Error("Layout instance is not the correct format"))
 |    _ -> raise (Error("Can only create a layout instance of a layout type"))

and compare_lists l1 l2 = match l1, l2 with
| [], [] -> true
| [], _
| _, [] -> false
| x::xs, y::ys -> x = y && compare_lists xs ys

(* Need to check e1 is Layout type *)
(* Need to check e2 is an ID FROM (TABLE TYPE) expression *)
(* Need to check e3 is a boolean type expression *)
and check_set_build e1 id e2 e3 env : (s_expr * dataType) =
 match e2 with Id(tbl_name) -> let (tbl_typ,_) = find_table env.scope tbl_name in
 let env' = { env with inSetBuild = tbl_name } in
   env'.scope.variables <- (tbl_typ, id, S_Noexpr)::env'.scope.variables;
 let (e1, t1) = check_expr e1 env' in
 let (e2, t2) = check_expr e2 env' in
 let (e3, t3) = check_expr e3 env' in
 (match t1 with
 Layout(name) ->
  if t3 = Bool then
   S_SetBuild(e1,id,e2,e3), Table
  else
   raise (Error("Third section of set build must be boolean-valued"))
 |    _ -> raise (Error("First section of set build must be a layout type")))


let rec check_stmt (s: Ast.stmt) (env: translation_environment) = match s with
  Block(ss) ->
 let scope' = { parent = Some(env.scope); variables = []; layouts = env.scope.
    layouts; tables = env.scope.tables; } in
      let env' = { env with scope = scope'} in
      let ss = List.map (fun s -> check_stmt s env') (List.rev ss) in
      scope'.variables <- List.rev scope'.variables;
      S_Block(scope', ss)
| Expr(e) -> let (e,t) = check_expr e env in S_Expr(e,t)
| Return(e) -> check_return e env
| If(elif_l, s) -> let elif_l' = check_cond elif_l env in
     let s_else = check_stmt s env in
     S_If(elif_l', s_else)
| For(e1, e2, s) -> check_for e1 e2 s env
| While(e, s) -> check_while e s env
| VarDeclS(v) -> check_var_decl v env
| LayoutCreation(name, v_list) ->  check_layout_creation name v_list env

and check_layout_creation (name: string) (v_list: var_decl list) (env:
   translation_environment) =
 (* Need to check whether this layout is already defined *)
 let exist = List.exists (fun (s, _) -> s = name) env.scope.layouts in
```

```
 if exist then
  raise (Error("Layout " ^ name ^ " already defined."))
 else
  (* Then need to check that every expr in the var_decl list is an ID expr *)
  let s_v_list = List.map ( fun v' ->
   let scope' = {env.scope with variables = []} in
         let env' = {env with scope = scope' } in
   let v' = check_var_decl v' env' in
   let e = (match v' with
   S_VarDecl(v) -> (match v with
    S_BasicDecl(_, e) -> e
   |   S_ListDecl(_, e) ->  e
   |   S_LayoutDecl(_, e) -> e)) in
   match e with
    S_Id(x, typ) -> (match typ with
     Layout(name) -> S_LayoutDecl(Layout(name), e)
    | List(typ) -> S_ListDecl(List(typ), e)
    |    _ -> S_BasicDecl(typ, e))
   | _ -> raise (Error("Layout creation statement must only contain datatype
      identifier pairs"))) v_list in
  env.scope.layouts <- (name, s_v_list)::env.scope.layouts;
  env.funcs <- { fname = "Write";
     ret_type = Void;
     formals = [S_BasicDecl(String, S_Id("o_file", String)); S_LayoutDecl(Layout(
        name), S_Id("output_str", Layout(name)))];
     body = [S_Expr(S_Noexpr, Void)];}::env.funcs;
  env.funcs <- { fname = "Append";
     ret_type = Void;
     formals = [S_BasicDecl(Table, S_Id("tbl", Table)); S_LayoutDecl(Layout(name),
        S_Id("layout_rec", Layout(name)))];
     body = [S_Expr(S_Noexpr, Void)];}::env.funcs;
  (* Layouts created are kept track of in the symbol table *)
  S_Expr(S_Noexpr, Void)

(* Need to check every expr in the elif_l is boolean valued *)
(* Need to check stmt is valid *)
and check_cond (elif: (expr * stmt) list) (env: translation_environment) =
 List.map (fun elif' -> let (e, t) = check_expr (fst elif') env in
 if t = Bool then
  let s = check_stmt (snd elif') env in (e,s)
 else
  raise (Error("Condition must be boolean-valued"))) elif

and check_return (e: expr) (env: translation_environment) =
 if env.in_func then
 let (e,t) = check_expr e env in
  if t = env.return_type then
   S_Return(e, t)
  else
   raise (Error("Must return compatible type. Expected type " ^ data_type_s env.
      return_type ^ ", found type " ^ data_type_s t ))
 else
  raise (Error("Cannot return outside of a function"))

(* e1 must be an identifier; e2 must be a list type *)
and check_for (e1: expr) (e2: expr) (s: stmt) (env: translation_environment) =
```

```
match e1 with
Id(x) -> let scope' = { parent = Some(env.scope); variables = [(Void, x, S_Noexpr)
   ]; layouts = env.scope.layouts; tables = env.scope.tables; } in
   let env' = { env with scope = scope' } in
      let (_, _) = check_expr e1 env' in
   let (e2, t2) = check_expr e2 env' in
    let t1 = (match t2 with
       List(typ) -> typ
     |  _ -> t2) in
    env'.scope.variables <- (t1, x, S_Noexpr)::env.scope.variables;
    match e2 with
     S_Id(name, _) -> let (typ,_,_) = find_variable env'.scope name in
     (match typ with
      List(_) -> let s = check_stmt s env' in S_For(S_Id(x, t2), e2, s)
     |    _ -> raise (Error("Must iterate over a list type. Type " ^ data_type_s typ
         ^ " found")))
    | S_ListLit(_,_) -> let s = check_stmt s env' in S_For(S_Id(x, t2), e2, s)
    | S_ListGen(_,_) -> let s = check_stmt s env' in S_For(S_Id(x, t2), e2, s)
    | _ -> raise (Error("How did you get here?"))


and check_while (e: expr) (s: stmt) (env: translation_environment) =
 (* check expr is boolean valued *)
 let (e,t) = check_expr e env in
 if t = Bool then
  let s' = check_stmt s env in S_While(e,s')
 else
  raise (Error("While expression must be boolean-valued"))

and check_var_decl (v: var_decl) (env: translation_environment) =
 match v with
 VarDecl(d,e) ->
  (match d with
  List(typ) ->   (match e with
         Id(x) -> let exist = List.exists (fun (_, s, _) -> s = x) env.scope.
            variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
            env.scope.variables <- (List(typ), x, S_Noexpr)::env.scope.variables;
            S_VarDecl(S_ListDecl(List(typ), S_Id(x, typ)))
        |   Assign(x,e) ->  let exist = List.exists (fun (_, s, _) -> s = x) env.
           scope.variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
           env.scope.variables <- (List(typ), x, S_Noexpr)::env.scope.variables;
            let (e',_) = check_expr e env in
             env.scope.variables <- (List(typ), x, e')::env.scope.variables;
            S_VarDecl(S_ListDecl(List(typ), S_Assign(x, e')))
        |  _ -> raise (Error ("Not a valid assignment")))
 |   Layout(name) -> (match e with
         Id(x) -> let exist = List.exists (fun (_, s, _) -> s = x) env.scope.
            variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
```

```
            env.scope.variables <- (Layout(name), x, S_Noexpr)::env.scope.variables;
            S_VarDecl(S_LayoutDecl(Layout(name), S_Id(x, Layout(name))))
        |   Assign(x,e) ->  let exist = List.exists (fun (_, s, _) -> s = x) env.
           scope.variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
           env.scope.variables <- (Layout(name), x, S_Noexpr)::env.scope.variables;
            let (e',_) = check_expr e env in
             env.scope.variables <- (Layout(name), x, e')::env.scope.variables;
            S_VarDecl(S_LayoutDecl(Layout(name), S_Assign(x, e')))
        |  _ -> raise (Error ("Not a valid assignment")))
 |   Table -> (match e with
        Id(x) -> let exist = List.exists (fun (_, s, _) -> s = x) env.scope.
            variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
            env.scope.variables <- (Table, x, S_Noexpr)::env.scope.variables;
            S_VarDecl(S_BasicDecl(Table, S_Id(x,Table)))
        |   Assign(x,e) ->  let exist = List.exists (fun (_, s, _) -> s = x) env.
           scope.variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
           env.scope.variables <- (Table, x, S_Noexpr)::env.scope.variables;
            let (e',_) = check_expr e env in
           (match e' with
            S_TableInit(typ) -> env.scope.tables <- (typ, x)::env.scope.tables
           |   _ -> env.scope.variables <- (Table, x, e')::env.scope.variables);
           S_VarDecl(S_BasicDecl(Table, S_Assign(x, e')))
        |  _ -> raise (Error ("Not a valid assignment")))
 |    _ -> (match e with
        Id(x) -> let exist = List.exists (fun (_, s, _) -> s = x) env.scope.
            variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
            env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
            S_VarDecl(S_BasicDecl(d, S_Id(x, d)))
        |   Assign(x,e) ->  let exist = List.exists (fun (_, s, _) -> s = x) env.
           scope.variables in
           if exist then
            raise (Error("Identifier already declared"))
           else
           env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
           let (e',_) = check_expr e env in
            S_VarDecl(S_BasicDecl(d, S_Assign(x, e')))
        |  _ -> raise (Error ("Not a valid assignment"))))

let check_formals (decl: var_decl) (env: translation_environment) =
 match decl with
 VarDecl(d,e) ->
 (  match d with
  List(typ) -> (match e with
        Id(x) -> env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
           S_ListDecl(d, S_Id(x, d))
```

```
        |   _ -> raise (Error("Function formals must be identifiers")))
 | Layout(name) -> (match e with
         Id(x) -> env.scope.variables <- (Layout(name), x, S_Noexpr)::env.scope.
             variables;
             S_LayoutDecl(Layout(name), S_Id(x, Layout(name)))
        |   _ -> raise (Error("Function formals must be identifiers"))
      )
 |  _ -> (match e with
         Id(x) -> env.scope.variables <- (d, x, S_Noexpr)::env.scope.variables;
             S_BasicDecl(d, S_Id(x, d))
        |   _ -> raise (Error("Function formals must be identifiers"))))

let check_fdecl (func: Ast.func_decl) (env: translation_environment) : (s_func_decl)
    =
 if env.in_func then
  raise (Error ("Cannot nest function declarations"))
 else
  let env' = { env with scope = {parent = Some(env.scope); variables = [(String, "
      stdout", S_Noexpr); (String, "stderr", S_Noexpr)]; layouts=env.scope.layouts;
      tables=env.scope.tables;};
  return_type = func.ret_type; in_func = true} in
  let formals = (List.rev (List.map (fun x -> check_formals x env') func.formals))
      in
  let f = { Sast.fname = func.fname; Sast.ret_type = func.ret_type; Sast.formals =
      formals; Sast.body = (List.map (fun x -> check_stmt x env') func.body );} in
  env.funcs <- f::env.funcs; f

(* Need to initialize reserved keywords and built-in funcs *)
let init_env : (translation_environment) =
 let func_i = [{ fname = "Write";
     ret_type = Void;
     formals = [S_BasicDecl(String, S_Id("o_file", String)); S_BasicDecl(String,
         S_Id("output_str", String))];
     body = [S_Expr(S_Noexpr, Void)];};
     { fname = "Write";
     ret_type = Void;
     formals = [S_BasicDecl(String, S_Id("o_file", String)); S_BasicDecl(Int, S_Id("
         output_str", Int))];
     body = [S_Expr(S_Noexpr, Void)];};
     { fname = "Write";
     ret_type = Void;
     formals = [S_BasicDecl(String, S_Id("o_file", String)); S_BasicDecl(Float, S_Id
         ("output_str", Float))];
     body = [S_Expr(S_Noexpr, Void)];};
     { fname = "Write";
     ret_type = Void;
     formals = [S_BasicDecl(String, S_Id("o_file", String)); S_BasicDecl(Bool, S_Id(
         "output_str", Bool))];
     body = [S_Expr(S_Noexpr, Void)];};
     { fname = "Write";
     ret_type = Void;
     formals = [S_BasicDecl(String, S_Id("o_file", String)); S_BasicDecl(Table, S_Id
         ("output_str", Table))];
     body = [S_Expr(S_Noexpr, Void)];};
      { fname = "Read";
       ret_type = Table;
```

```
       formals = [S_BasicDecl(String, S_Id("in_file", String));S_BasicDecl(String,
          S_Id("delim", String))];
       body = [S_Expr(S_Noexpr, Void)]; };
      { fname = "Column";
       ret_type = List(String);
       formals = [S_BasicDecl(Table, S_Id("tbl", Table));S_BasicDecl(String, S_Id("
          col_name", String))];
       body = [S_Expr(S_Noexpr, Void)]; };
      { fname = "Column";
       ret_type = List(String);
       formals = [S_BasicDecl(Table, S_Id("tbl", Table));S_BasicDecl(Int, S_Id("
          col_id", Int))];
       body = [S_Expr(S_Noexpr, Void)]; };] in
 let scope_i = { parent = None;
        variables = [(String, "stdout", S_Noexpr); (String, "stderr", S_Noexpr)];
           layouts=[]; tables=[];} in
 { funcs = func_i ; scope = scope_i; return_type = Void; in_func = false; inSetBuild
     = "NOT"; }

let check_prgm (prog: Ast.program ) : (Sast.s_program) =
 let env = init_env in
 let layout_stmts = List.filter (fun stmt -> match stmt with LayoutCreation(_,_) ->
    true | _ -> false) prog.stmts in
 let rest_stmts = List.filter (fun stmt -> match stmt with LayoutCreation(_,_) ->
    false | _ -> true) prog.stmts in
 let layout_stmts = List.map (fun x -> check_stmt x env) (List.rev layout_stmts) in
 let env = {env with scope = {env.scope with variables = [(String, "stdout",
    S_Noexpr); (String, "stderr", S_Noexpr)]}} in
 { Sast.stmts = layout_stmts@(List.map (fun x -> check_stmt x env) (List.rev
    rest_stmts) );  Sast.funcs = (List.map (fun x -> check_fdecl x env) prog.funcs)
    ; Sast.syms = env.scope}
```

FRYLayout.java

```
package fry;

import java.lang.reflect.Field;
import java.util.Hashtable;

public class FRYLayout{

 public Hashtable<Integer,String> layout; // Maps the value names to their position
    number
 public Hashtable<Integer, String> datatype; // Maps the position value to their
    datatype
 public int numFields;

 public FRYLayout(){
  buildHashtable();
 }

 /**
  * Creates an anonymous layout (no field names)
  * @param numFields
  */
 public FRYLayout(int numFields){
  layout = new Hashtable<Integer, String>();
  datatype = new Hashtable<Integer, String>();
```

```
  for (int i = 0; i < numFields; i++){
   layout.put(i, (new Integer(i)).toString());
  }
 }

 public FRYLayout(Object[] record){

 }

 public void buildHashtable(){
  layout = new Hashtable<Integer, String>();
  datatype = new Hashtable<Integer, String>();
  int i = 0;
  for (Field field : this.getClass().getDeclaredFields()){
   if ( field.getName().equals("layout") ||
     field.getName().equals("datatype")){
    continue;
   }
   layout.put(i, field.getName());
   datatype.put(i, "String");
   i++;
  }
  numFields=i;
 }

 public Field[] getFields(){
  Field[] fields = new Field[this.getClass().getDeclaredFields().length];
  int i = 0;
  for (Field field : this.getClass().getDeclaredFields()){
   if ( field.getName().equals("layout") ||
     field.getName().equals("datatype")){
    continue;
   }
   fields[i] = field;
   i++;
  }
  return fields;
 }

 public Field getField(int index) throws NoSuchFieldException, SecurityException{
  return this.getClass().getDeclaredField(layout.get(index));
 }

 public Integer getIdByName(String fieldName){
  /**
   * This is inefficient but don't have enough time to fix things
   */
  for(int i = 0; i < layout.size(); i++){
   if(layout.get(i).equals(fieldName)){
    return i;
   }
  }
  return null;
 }


}
```

FRYListFactory.java

```
package fry;

public class FRYListFactory{

 public static FRYList<Integer> getGeneratedFryList(int min, int max){
  FRYList<Integer> list = new FRYList<Integer>(max-min);
  for(int i = min; i <= max; i++){
   list.add(new Integer(i));
  }
  return list;
 }

}
```

FRYList.java

```
package fry;

import java.util.ArrayList;
import java.util.Collection;

public class FRYList<E> extends ArrayList<E> {

 /**
  *
  */
 private static final long serialVersionUID = 1L;

 public FRYList(){
  super();
 }

 public FRYList(Collection<? extends E> c){
  super(c);
 }

 public FRYList(int initialCapactity){
  super(initialCapactity);
 }


 @Override
 public E get(int index){
  try{
   return super.get(index);
  } catch (IndexOutOfBoundsException e){
   return null;
  }
 }


}
```

FRYTable.java

```
package fry;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.regex.Pattern;

public class FRYTable {
 public FRYLayout layout; // Maps the value names to their position number
 public boolean isLayoutSet;
 private int numFields; //Number of fields in a record
 private ArrayList<String[]> data; //data stored in table
 int recIndex, numRecords;

 public FRYTable(){
  layout = new FRYLayout();
  isLayoutSet = false;
  recIndex = 0;
  data = new ArrayList<String[]>();
 }

  public ArrayList<String[]> getData() {
  return data;
 }

 public FRYTable(ArrayList<String[]> data, FRYLayout layout){
  this(layout);
  this.data = data;
  numRecords = data.size();
 }

 public FRYTable(FRYLayout layout){
  this.layout = layout;
  isLayoutSet = true;
  recIndex = 0;
  data = new ArrayList<String[]>();
 }

 public void readInput(InputFile i) throws IOException{
  String line = null;
  String[] record = null;
  BufferedReader buf = i.buf;
  String delim = i.delim;
  data = new ArrayList<String[]>();
  numRecords = 0;
  while ( (line = buf.readLine()) != null){
   record = parseRecord(line, delim);
   data.add(record);
   numRecords++;
  }
  numFields = record.length;
  recIndex = 0;
  if(layout == null){
   this.layout = new FRYLayout(numFields);
  }
 }
```

```java
public String[] parseRecord(String line, String delim){
 String[] record = line.split(Pattern.quote(delim));
 return record;
}
/**
 * Returns next record in the dataset
 *
 * @return
 * Returns next record in the FRYTable if it exists, returns null if there are no
    more
 * records
 */
public String[] readRecord(){
 if(recIndex == numRecords){
  return null;
 }
 String[] dat = data.get(recIndex);
 recIndex++;
 return dat;
}

public FRYList<String> getColumn(int i){
 String[] col = new String[data.size()];
 for(int j = 0; j < data.size(); j++){
  col[j] = data.get(j)[i];
 }
 return new FRYList<String>(Arrays.asList(col));
}


public FRYList<String> getColumn(String fieldName){
 return getColumn(layout.getIdByName(fieldName));
}

public void append(FRYLayout rec){
 Field[] fields = rec.getFields();
 String[] record = new String[fields.length];
 int i = 0;
 for (Field field : fields){
  try {
    if (field.get(rec) == null){
     record[i] = "";
    }
    else {
     record[i] = field.get(rec).toString();
    }
  } catch (IllegalArgumentException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  } catch (IllegalAccessException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
  i++;
 }
 data.add(record);
 numRecords++;
```

```
 }

 public void resetIndex(){
  recIndex = 0;
 }

}
```

InputFile.java

```
package fry;

import java.io.BufferedReader;

public class InputFile {
 public BufferedReader buf;
 public String delim;

 public InputFile(BufferedReader buf, String delim) {
  this.buf = buf;
  this.delim = delim;
 }
}
```

IOUtils.java

```
/**********************************

IOUtils.java contains functions for
reading and writing to input/output
streams

***********************************/
package fry;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.util.ArrayList;


public class IOUtils {

 public static String stdout = "stdout";
 public static String stderr = "stderr";
 public static String stdin = "stdin";

 public static PrintWriter getPrintWriter(String outputSpec){
  PrintStream o = null;
  // OutputSpec should be stdout, stdin, or string path to file
  if ( outputSpec.equals("stdout") ){
   o = System.out;
  }
  else if ( outputSpec.equals("stderr") ){
```

```
   o = System.err;
  }
  else {
   try {
    o = new PrintStream(new File(outputSpec));
   }
   catch (FileNotFoundException e) {
    System.err.println("File " + outputSpec + " not found.\n" + e.getMessage());
    System.exit(1);
   }
  }
  return new PrintWriter(o);
 }


 public static void Write(String outputSpec, int expr){
  Write(outputSpec, Integer.toString(expr));
 }

 public static void Write(String outputSpec, float expr){
  Write(outputSpec, Float.toString(expr));
 }

 public static void Write(String outputSpec, boolean expr){
  if (expr)
   Write(outputSpec, "true");
  else
   Write(outputSpec, "false");
 }

 // Writes to the output stream specified
 public static void Write(String outputSpec, String text){
  PrintWriter o = getPrintWriter(outputSpec);
  System.out.println(text);
  o.write(text);
 }

 public static void Write(String outputSpec, Object obj){
  Write(outputSpec, obj.toString());
 }

 public static void Write(String outputSpec, FRYTable tab){
  Write(outputSpec, tab, ",");
 }

 public static void Write(String outputSpec, FRYTable tab, String d){
  Object[] rec = null;
  String out = "";
  PrintWriter o = getPrintWriter(outputSpec);
  while((rec = tab.readRecord()) != null ){
   out = "";
   for(int i = 0; i < rec.length; i++){
    if ( i == (rec.length - 1)){
     out += rec[i].toString();
    }
    else {
     out += rec[i].toString()+d;
```

```
    }
   }
   o.write(out+"\n");
  }
  o.close();
 }

 public static void WriteColumn(String outputSpec, ArrayList<Object> column){
  PrintWriter o = getPrintWriter(outputSpec);
  for(Object obj : column){
   o.write(obj.toString()+"\n");
  }
  o.close();
 }

 public static void Append(FRYTable tbl, FRYLayout rec){
  tbl.append(rec);
 }

 public static InputFile Read(String inputSpec, String delim) throws IOException{
  BufferedReader i = null;
  // inputSpec should be stdin or string path to file
  if ( inputSpec.equals("stdin")){
   i = new BufferedReader( new InputStreamReader(System.in));
  }
  else {
   try{
    i = new BufferedReader( new FileReader(new File(inputSpec)));
   }
   catch (FileNotFoundException e) {
    System.err.println("File " + inputSpec + " not found.\n" + e.getMessage());
    System.exit(1);
   }
  }
  return new InputFile(i, delim);
 }

}
```

run_fry.sh

```
#!/bin/sh
ret=$(compile_fry.sh $1)
if [ "$(echo $ret | grep Error)" != "" ];then
    echo $ret
    exit 1
fi
java fry
```

compile_fry.sh

```
#!/bin/sh
fry -c < $1
fry_ret=$?
if [ "$fry_ret" -ne "0" ]; then
    echo "Error compiling $1 to java"
    exit 1
fi
javac fry.java
```

```
java_ret=$?
if [ "$fry_ret" -ne "0" ]; then
   echo "Error compiling $1"
   exit 1
fi
echo "$1 compiled to fry.class"
```

run_tests.sh

```
#!/bin/bash

## Testing Functions

print_usage(){
    echo "Usage: "
    echo "      $0 -s <sandbox_dir> -d <test_dir> [-f <comma separated .fry test in
       test_dir>] [-h]"
}

# Diffs files specified by args 1 and 2
diff_output(){
    orig=$1
    out=$2

    diff $orig $out > ${log_dir}/${test}_diff.log
    diff_ret=$?
    if [ "$diff_ret" -ne "0" ]; then
        report_error "ERROR EXECUTING DIFF"
        return 1
    fi
    if [ "$(cat ${log_dir}/${test}_diff.log | wc -l)" -ne "0" ]; then
        report_error "DIFFERENCE IN OUTPUT"
        cat ${log_dir}/${test}_diff.log
        return 1
    else
        echo "TEST  $(filename $(echo $test | sed 's/\.fry//')) PASSED"
        return 0
    fi
}

# Optionally takes a message as the first argument
report_error(){
        echo "************************"
        echo "ERROR IN TEST $(filename $(echo $test | sed 's/\.fry//'))"
        echo -e "$1"
        echo "*************************"
}

# Reads log file passed as argument, checks for any errors and reports them if they
   exist
check_and_report_errors(){
    log_file=$1

    exceptions=$(grep "exception|Exception|EXCEPTION" ${log_file})
    if [ "$(cat ${log_file} | wc -l)" -ne "0" ]; then
        report_error "ERROR IN LOG FILE $log_file\n\n$(cat $log_file)\n"
        return 1
    fi
```

```
    return 0

}

for i in $@; do
    case $i in
        "-s")
            # Sandbox dir
            sandbox_dir=$2
            shift 2
         ;;
        "-h")
            print_usage
            exit 1
         ;;
        "-f")
            tests=$( filename $2 | sed 's/,/ /')
            shift 2
         ;;
        "-d")
            test_dir=${2}
            shift 2
        ;;
    esac
done

if [ -z "${test_dir}" ]; then
    echo "Must specify test directory"
    print_usage
    exit 1
fi

if [ -z "${sandbox_dir}" ]; then
    echo "Must specify sandbox directory"
    print_usage
    exit 1
fi

if [ -z "${tests}" ]; then
    tests=$(ls ${test_dir}/*.fry)
    root_output_dir="$(pwd)/fry_tests_$(date +%m%d)"
else
    root_output_dir="$(pwd)/"
fi

export PATH=${sandbox_dir}/src/:$PATH

ulimit -t 30

root_log_dir="${root_output_dir}/logs"

mkdir -p ${root_log_dir}

cd ${root_output_dir}

for test_src in $tests; do
```

```
    test=$(filename $(echo $test_src | sed 's/\.fry//'))

    echo ""
    echo "Running test ${test}..."

    log_dir="${root_log_dir}/${test}/"
    output_dir="${root_output_dir}/${test}/"

    mkdir -p $log_dir
    mkdir -p $output_dir

    # copy over files for test
    cp -f ${test_dir}/${test}.fry $output_dir
    cp -f ${test_dir}/${test}.out $output_dir
    cp -f ${test_dir}/${test}.in $output_dir 2>/dev/null
    error=0
    # Translate to java
    cd $output_dir
    fry -c < ${test}.fry > test.java 2>${log_dir}/${test}_trans.log

    check_and_report_errors ${log_dir}/${test}_trans.log
    error=$?
    if [ "${error}" -ne "0" ]; then
        continue
    fi

    javac fry.java 2>${log_dir}/${test}_javac.log
    error=$(cat ${log_dir}/${test}_javac.log | wc -l)
    if [ "${error}" -ne "0" ]; then
        report_error "JAVA COMPILE ERROR\n\n$(cat ${log_dir}/${test}_javac.log)\n"
        continue
    fi

    java fry > ${test}.tmp
    error=$?
    if [ "${error}" -ne "0" ]; then
        report_error "JAVA RUNTIME ERROR\n\n$(cat ${output_dir}/${test}.tmp)\n"
        continue
    fi

    diff_output ${test}.out ${test}.tmp
    error=$?
    if [ "${error}" -eq "0" ]; then
        rm -f ${log_dir}/${test}_trans.log
        rm -f ${log_dir}/${test}_javac.log
        rm -f ${log_dir}/${test}_diff.log
    fi

    echo "Finished test ${test}..."
    echo ""
done

cd ..
```

arith_test01.fry

```
/#
Tests the basic arithmetic operations
```

```
#/

Write(stdout, 20 + 25);
Write(stdout, 25 - 20);
Write(stdout, 15 * 5);
Write(stdout, 15 / 5);
```

arith_test01.out

```
45
5
75
3
```

cond_test01.fry

```
bool x = true;
str y = "Fail";

if (x) {
    y = "Success";
}

Write(stdout, y);
```

cond_test01.out

```
Success
```

cond_test02.fry

```
bool x = true;
int z = 5;
str y = "Fail";

if (not x) {
    y = "Fail";
}
elif (z < 4){
    y = "Fail";
}
else {
    y = "Success";
}
Write(stdout, y);
```

cond_test02.out

```
Success
```

func_test01.fry

```
/#

    func_test01.fry

    This tests creating a basic function

#/

str test(){
```

```
    str x = "test";
    ret x;
}

Write(stdout, test());
```

func_test01.out

```
test
```

func_test02.fry

```
/#

    func_test01.fry

    This tests creating a basic function

#/

int test(){
    int x = 5;
    int y = 6;
    ret x+y;
}

Write(stdout, test());
```

func_test02.out

```
11
```

func_test03.fry

```
/#

    func_test01.fry

    This tests creating a basic function

#/

str List test(){
    str List l_str = ["test","test2","test3\n","test4\ntest5"];
    ret l_str;
}

str List my_str = test();
for ( s <- my_str )
    Write(stdout,s);
```

func_test03.out

```
test
test2
test3

test4
test5
```

func_test04.fry

```
/#

    func_test04.fry

    This tests creating a function which takes multiple arguments

#/

int printProd(int x, int y){
    Write(stdout, x*y);
    ret 0;
}

printProd(5,8);
printProd(50,42);
```

func_test04.out

```
40
2100
```

layout_test01.fry

```
/#

    Basic layout tests

#/

Layout date = { int: mon, int: day, int: year};
Layout date today = Layout date {10, 23, 2014};

Write(stdout, today);
```

layout_test01.out

```
10|23|2014
```

layout_test02.fry

```
/#

    Basic layout tests

#/

Layout date = { int: mon, int: day, int: year};
Layout date today = Layout date {10, 23, 2014};

Layout user = { Layout date: bday, str: fname, str: lname };
Layout user tom = Layout user {Layout date {9, 19, 1991}, "Tom", "DeVoe"};

Write(stdout, tom);
```

layout_test02.out

```
9|19|1991|Tom|DeVoe
```

layout_test03.fry

```
/#

    Basic layout tests

#/

Layout flt = { int: int_part, int: frac_part };

Layout flt addFlt(Layout flt flt1, Layout flt flt2){
    int int_part = flt1.{int_part}+flt2.{int_part};
    int frac_part = flt1.{frac_part}+flt2.{frac_part};
    int diff;
    if (frac_part > 100){
        diff = frac_part - 100;
        frac_part = diff;
        int_part = int_part + 1;
    }

    ret Layout flt {int_part, frac_part};
}

Layout flt flt1 = Layout flt {4,76};
Layout flt flt2 = Layout flt {5,38};
Layout flt flt3 = addFlt(flt1,flt2);


Write(stdout, flt3);
```

layout_test03.out

```
10|14
```

list_test01.fry

```
str List l_str = ["TEST0", "TEST1", "TEST2", "TEST3" ];
str List l_str2 = l_str[1:3];
str List l_str3 = l_str[:2];
str List l_str4 = l_str[1:];

Write(stdout, "List 1:");
for (i <- l_str){
        Write(stdout, i);
}
 Write(stdout, "List 2:");
for (i <- l_str2){
        Write(stdout, i);
}

Write(stdout, "List 3:");
for (i <- l_str3){
        Write(stdout, i);
}

Write(stdout, "List 4:");
for (i <- l_str4){
    Write(stdout, i);
}
```

list_test01.out

```
List 1:
TEST0
TEST1
TEST2
TEST3
List 2:
TEST1
TEST2
List 3:
TEST0
TEST1
List 4:
TEST1
TEST2
TEST3
```

list_test02.fry

```
str List l_str = ["TEST0", "TEST1", "TEST2", "TEST3" ];
Write(stdout,l_str[0]);
Write(stdout,l_str[1]);
Write(stdout,l_str[2]);
Write(stdout,l_str[3]);
```

list_test02.out

```
TEST0
TEST1
TEST2
TEST3
```

loop_test01.fry

```
/#
This tests the while loop
#/

int x = 5;

while ( x > 0 ) {
    Write(stdout,x);
    x--;
}
```

loop_test01.out

```
5
4
3
2
1
```

loop_test02.fry

```
/#
Tests for loop with list
#/

int List l_int = [1,2,3,4];
```

```
for (i <- l_int) {
    Write(stdout, i);
}
```

loop_test02.out

```
1
2
3
4
```

loop_test03.fry

```
/#
Tests for loop with list
#/

int List l_int = [1 to 4];

for (i <- l_int) {
    Write(stdout, i);
}
```

loop_test03.out

```
1
2
3
4
```

loop_test04.fry

```
/#
Tests for loop with list
#/

for (i <- [1 to 4]) {
    Write(stdout, i);
}
```

loop_test04.out

```
1
2
3
4
```

loop_test05.fry

```
/#
Tests for loop with list
#/

for (i <- [1,2,3,4]) {
    Write(stdout, i);
}
```

loop_test05.out

```
1
2
```

```
3
4
```

table_test01.fry

```
/#
    Table test
#/
    Layout date = { int: mon, int: day, int: year };
    Table date_tbl = Table (Layout date);
    date_tbl = Read("table_test01.in",",");
    Write(stdout, date_tbl);
```

table_test01.in

```
This,is,a,test
This,is,a
test,This,is,a
a,is,This,test,a
```

table_test01.out

```
This,is,a,test
This,is,a
test,This,is,a
a,is,This,test,a
```

table_test02.fry

```
/#
    Table test
#/
    Layout user = { int: id, str: fname, str: lname };
    Table user_tbl = Table (Layout user);
    user_tbl = Read("table_test02.in","|");
    Layout username = { str: fname, str: lname};
    Table test = [ Layout username {i.{fname}, i.{lname}} | i <- user_tbl; i.{id} >
        4 ];
    Write(stdout, test);
```

table_test02.in

```
1|Tom|DeVoe
2|John|Smith
5|Eli|Manning
15|Peter|Piper
4|Barack|Obama
```

table_test02.out

```
Eli,Manning
Peter,Piper
```

table_test03.fry

```
/#
    Table test - test append
#/
    Layout user = { int: id, str: fname, str: lname };
    Table user_tbl = Table (Layout user);
    for ( i <- [1 to 100] )
        Append(user_tbl, Layout user {i, "Test" + i, "Test" + (2*i)});
    Write(stdout,user_tbl);
```

table_test03.out

```
1,Test1,Test2
2,Test2,Test4
3,Test3,Test6
4,Test4,Test8
5,Test5,Test10
6,Test6,Test12
7,Test7,Test14
8,Test8,Test16
9,Test9,Test18
10,Test10,Test20
11,Test11,Test22
12,Test12,Test24
13,Test13,Test26
14,Test14,Test28
15,Test15,Test30
16,Test16,Test32
17,Test17,Test34
18,Test18,Test36
19,Test19,Test38
20,Test20,Test40
21,Test21,Test42
22,Test22,Test44
23,Test23,Test46
24,Test24,Test48
25,Test25,Test50
26,Test26,Test52
27,Test27,Test54
28,Test28,Test56
29,Test29,Test58
30,Test30,Test60
31,Test31,Test62
32,Test32,Test64
33,Test33,Test66
34,Test34,Test68
35,Test35,Test70
36,Test36,Test72
37,Test37,Test74
38,Test38,Test76
39,Test39,Test78
40,Test40,Test80
41,Test41,Test82
42,Test42,Test84
43,Test43,Test86
44,Test44,Test88
45,Test45,Test90
46,Test46,Test92
47,Test47,Test94
48,Test48,Test96
49,Test49,Test98
50,Test50,Test100
51,Test51,Test102
52,Test52,Test104
53,Test53,Test106
54,Test54,Test108
55,Test55,Test110
```

```
56,Test56,Test112
57,Test57,Test114
58,Test58,Test116
59,Test59,Test118
60,Test60,Test120
61,Test61,Test122
62,Test62,Test124
63,Test63,Test126
64,Test64,Test128
65,Test65,Test130
66,Test66,Test132
67,Test67,Test134
68,Test68,Test136
69,Test69,Test138
70,Test70,Test140
71,Test71,Test142
72,Test72,Test144
73,Test73,Test146
74,Test74,Test148
75,Test75,Test150
76,Test76,Test152
77,Test77,Test154
78,Test78,Test156
79,Test79,Test158
80,Test80,Test160
81,Test81,Test162
82,Test82,Test164
83,Test83,Test166
84,Test84,Test168
85,Test85,Test170
86,Test86,Test172
87,Test87,Test174
88,Test88,Test176
89,Test89,Test178
90,Test90,Test180
91,Test91,Test182
92,Test92,Test184
93,Test93,Test186
94,Test94,Test188
95,Test95,Test190
96,Test96,Test192
97,Test97,Test194
98,Test98,Test196
99,Test99,Test198
100,Test100,Test200
```