Record Transformation Language

Joe Justesen

January 28, 2014

The Record Transformation Language (RTL) provides an easy way to validate and transform your data files. Data files consisting of coma separated values (CSV) can be check for data errors, used to validation or transform of other CSV files or transformed into new data files.

Data Types

RTL will support the following data simple types: int, real, string, date, time, datetime and bool; plus the following container types: set, map and array. These containers can hold any simple data type as well as other containers. The data types in a container must be of a uniform type.

No implicit data type conversion is supported. All data conversions must be performed using the built in functions like: int_to_string, string_to_float, char_to_string.

Operators

Supported operators are the assignment =; standard mathematical operators -, +, /, *; and comparison operators <, >, <=, >=, ==, !=.

Control Statements

Supported control statements will be if-else; while and for loops.

Variables

Instance variables must be declared, specifying the data type they contain, before being used.

Functions

There are three types of functions in RTL, generic, filter, and transform functions.

A generic function, denoted by the *function* keyword which can contain any number and type of parameters and a single return type. The return data type is specified after the parameter list, if present.

A filter function, denoted by the *filter* keyword, has only a single parameter, a map representing a single record and returns a boolean. Filter functions can only be used in a *check* process. Returning true sends the record to the next stop of the process, false rejects the record, no further processing is performed on that record. Any modification of the record map lost upon return of the function.

A transform function, denoted by the *transform* keyword, has only a single parameter, a map representing a single record, its return value is ignored. Transform functions can only be used in a *trans* process. Transform functions provide the ability to change the contents of a record. All modifications to the record map parameter will be retained.

In all functions, the last expression evaluated is the value returned by the function. The *return* statement causes the function to exit after evaluating the expression after the return. All expressions after a *return* statement in a function must evaluate to the same data type.

Files

Files that will be used as input or output to the processing, can be specified with *file* statement. There are several types of files: input, output, filter, transform and log files. Input and output files are, as their name implies, ordinary CSV files used to read and write data. Filter and transform files are used as inputs to filter and transform processes respectively and must conform to some simple structural rules. Filter files must contain a set of records with a single element. Transform files must contain a set of records with two elements, creating a map between the first elements and the second elements in the file.

The map records created for each row, all map the header name to the field's constant as a string. There is no attempt to example the data in the field and determine the correct data type. A header is required for all input files.

Processing Records

A process in RTL is a chain of *check* and *trans* code blocks or functions that are performed in the sequence defined by process list. c*heck* and *trans* statements can be intermingled as needed and the output for this sequence of processing will be written to an output file.

check

In RTL, there are two types of check blocks. First, in line code blocks act as a filter function defined in the *check* statement. The code in the block is executed as if it was as separate function. Second, a filter function can be named. The filter function would be called for each record in the input file.

If any check rejects a record, the sequence of processing for that record is halted and the processing starts again with the next record in the input file.

to

The results of sending the input record through all the check and trans processes is written to the file listed in the *to* statement.

Runtime Environment

RTL will run on the Java Virtual Machine. Source files will compile to Java byte code and be stored in jar files for execution. There will be no support for explicit Java interoperability besides that supported by the language. Data types in the language will be mapped to Java data types were appropriate.

Example

Here is a small example with a single input file, a filter file, a transform file and the resulting output file. The program in example.rtl will take the customers.csv file and verify that the Customer is set and that the Product is LIN. The Notes field is set to an empty string. The output is written to the results.csv file using '|' to delimit the fields.

customers.csv

```
Customer,DeliveryDate,Product,Type,Amount,UOM,Notes
10889,2014-01-21T14:23:00,LIN,SDAY,5000,GAL,
10289,2014-01-21T16:00:00,LIN,SDAY,1000,GAL,
10453,2014-01-21T17:12:00,BHY,SDAY,2500,M3,
10351,2014-01-21T18:56:00,LIN,MDAY,3000,LBS,Bill by weight
```

example.rtl

```
file customers = "./customers.csv"
file results = "./results.csv" sep="|" output

function PassThrough( int value ) int {
      return value;
}

filter ValidCustomer {
      rec[Customer] =
          int_to_string(PassThrough(string_to_int(rec[Customer])); # does nothing
      return rec["Customer"] > 0;
}

process customers
    check { rec["Product"] == "LIN" }                    # inline validiation
    check ValidCustomer                                   # function validation
    to results                                            # resulting file
```

results.csv

```
Customer|DeliveryDate|Product|Type|Amount|UOM|Notes
10889|2014-01-21T14:23:00|LIN|Single Day|5000|GAL|
10289|2014-01-21T16:00:00|LIN|Single Day|1000|GAL|
10351|2014-01-21T18:56:00|LIN|Multi Day|3000|LBS|
```

RTL Tutorial

As the name implies, RTL is designed to modify records, specifically, records stored in a character separated values (CSV) file. This tutorial will walk you through writing scripts to transform your data files starting with simple reading and writing of your data. By the end, you will know how to validate your data and change existing values based on other data files.

The best way to learn about RTL is to try it out. I will show you several examples, each one adding additional features. You are encouraged to run each RTL script for yourself. Change the data and rerun the script, see what happens. Experiment for yourself.

Example 1 – Hello World

This is the simplest RTL script that does anything. It copies the `e1_in.csv` to `e1_out.csv`. You start by specifying the files you are reading or writing to using the `file` statement. All files are by default read only, you let RTL know that the file is used for writing by appending the `output` property to the file statement. This will create a named data stream that can be read or written to using the `process` statement.

example1.rtl

```
file file_in  = "./in.csv"
file file_out = "./out.csv" output

process file_in to file_out
```

in.csv

```
Col1,Col2
Hello,World
```

out.csv

```
Col1,Col2
Hello,World
```

Example 2 – Change Separation character

This copies the files just like example 1, but changes the output separator character. Notice the `sep` property on the `file_out` data stream.

example2.rlt

```
file file_in  = "./in.csv"
file file_out = "./out.csv" output sep=|
process file_in to file_out
```

out.csv

```
Col1|Col2
Hello|World
```

Example 3 – Simple Validation

Most times when you are handling data, you want to make sure the data values are what you expect them to be. You want to stop the "garbage in, garbage out" syndrome as soon as you can. Example 3 shows how to verify that the data in the Product column is correct.

example3.rtl

```
file file_in  = "./customers.csv"
file file_out = "./out.csv" output

process file_in
    check { rec["Product"] = "LIN"; }
    to file_out
```

customers.csv

```
Number,Name,ZipCode,Product,OrderNum,Notes
10001,Alson Inc,18062,LIN,9902,
10034,Fast Freeze,18087,LIN,1009,
10282,Slow Welds,18109,BOX,8931,Do not deliver!
10198,Macungie Hospital Supply,18062,LOX,,Not currently active
10019,Allentown Blast Steel,18109,LIN,3879,Call Bob (610) 482-8932
```

out.csv

```
Number,Name,ZipCode,Product,OrderNum,Notes
10001,Alson Inc,18062,LIN,9902,
10034,Fast Freeze,18087,LIN,1009,
10019,Allentown Blast Steel,18109,LIN,3879,Call Bob (610) 482-8932
```

Example 4 – Filter Based Validations

Lets redo example 3, moving the validation checks to its own function. Use this technique when you have more than a simple check will make your process easier to understand at a later date. We will reuse the data files from example 3 for this example.

example4.rtl

```
file file_in  = "./customers.csv"
file file_out = "./out.csv" output

filter ValidateProduct {
    return rec["Product"] = "LIN";
}

process file_in
    check ValidateProduct
    to file_out
```

Example 5 – Changing data

If we wanted to prepend the string "AN-" to all the customer account numbers, we can use the check statement as shown in this example.

example6.trl

```
file file_in  = "./customers.csv"
file file_out = "./out.csv" output
 /
process file_in
    check {
      rec["Number"] = str_concat("AN-", rec["Number"]);
      return true;
    }
    to file_out
```

an_out.csv

```
Number,Name,ZipCode,Product,OrderNum,Notes
AN-10001,Alson Inc,18062,LIN,9902,
AN-10034,Fast Freeze,18087,LIN,1009,
AN-10198,Macungie Hospital Supply,18062,LOX,,Not currently active
AN-10019,Allentown Blast Steel,18109,LIN,3879,Call Bob (610) 482-8932
```

RTL Reference

1. JVM

Record Transformation Language (RTL) runs on the JVM. Runtime support for operations on data types are all implemented in Java. This constrains the data types to be compatible to Java's data types.

2. Lexical Conventions

Only the ASCII character set is supported, characters outside of that character set in source files will cause a error.

Regular expressions as defined in flex are used to add clarity. These expression follow the rules of regular expression as used by the flex package. These expressions will be in `mono spaced font`.

2.1 Blanks

```
blank = [ \t\n\r]+
```

Unless blanks are inside a string literal, they are ignored and provide no meaning to the program except as separators for program elements.

2.2 Comments

```
comment = "#".*$
```

Outside of a string literal, all characters following a `#` character on the same line of code are considered comments and are ignored by the compiler.

2.3 Identifiers

```
letter     = [a-zA-Z]

digit      = [0-9]

identifier = ({letter}|"_")({letter}|"_"|{digit})*
```

All characters in an identifier are significant. Identifiers are limited 256 characters. Identifiers are case sensitive, upper and lower case characters are considered different.

2.4 Integer Literals

```
integer = ({digit})+("_"+{digit}+)*
```

For readability, all integers can have the underscore character embedded in the number, the underscore is ignored when evaluating the value of the literal. Note, the integer literal cannot have a trailing underscore.

2.5 Floating Point Literals

```
float = {integer}"."{integer}([eE][-+]?{digit}+)?
```

Just as in integer literals, underscore characters can be embedded in the number for readability. Floats in RTL conform to double precision 64 bit IEEE 754 floating point standard. Note, all floating point literals require a digit before and after the decimal point. `.34` and `124.` are not validate floating point numbers.

2.5 Character Literals

```
character = [ -~]
```

RTL supports all ASCII printable characters as literals.

2.6 String Literals

```
String = "\"".*"\""
```

To embed a double qoute in a string, you can prefix the double quote with a backslash. `"This \"string\" is quoted."` is an example where the word string is surrounded by double quotes.

2.7 semicolon

The semi colon is used to delimit expression and statements. All expression must end with a semi colon, but not all statements have this requirement. Any statement that ends with a code block, which are a list of one or more statements delimited by `'{'` and `'}'` brackets, does not require a semi colon.

2.8 Keywords

These identifiers are keywords, and as such, cannot be used as program identifiers.

```
bool       break     char      check     else      file
filter     float     for       function  if        int
process    print     rec       return    string    to
while
```

3. Objects and names

All objects must be declared, and there fore named, before they can be accessed. Static scoping is used to limit the scope of a name either the file level or to a block delimited by `{}`. Binding the same name to a different object in an inner scope will hide the out binding.

4. Expressions

Expressions are evaluated in the order they are listed.

4.1 (*expression*)

Provides grouping for expression so that the expression inside the brackets must be evaluated before any expression before or after the grouped expressions.

4.2 *expression*$_1$ [*expression*$_2$]

$expression_1$, must be a map; $expression_2$ must evaluate to a string.

4.3 – *expression*

Reverses the numeric sign of the expression.

4.4 Binary expressions

$expression_1 * expression_2$ - multiplication

$expression_1 / expression_2$ - division

$expression_1 + expression_2$ - addition

$expression_1 - expression_2$ - subtract

Evaluates to numerical result of the normal mathematical operations. These operators are only valid for numerical data types int and float.

$expression_1 < expression_2$ - less than

$expression_1 > expression_2$ – greater than

$expression_1 <= expression_2$ - less than or equal

$expression_1 >= expression_2$ – greater than or equal

$expression_1 == expression_2$ - equal

$expression_1 != expression_2$ – not equal

$expression_1 = expression_2$ – assignment

Assign the values of $expression_2$ to the object in $expression_1$. The the data types of $expression_1$ and $expression_2$ must be the same.

4.5 Operator precedence

Operators are listed in order of precedence, lower precedence operators are listed below those with a higher precedence.

Operator Type	Operator	Associativity
Primary	[]	Left to right
Binary	* /	Left to Right
	+ -	
	< > <= >= == !=	

Operator Type	Operator	Associativity
Assignment	=	Right to left

5. Declarations

A declaration creates an object and gives it a name. These objects are usually variables or functions. The object can then be referenced by name as long as the name is in scope. When the name goes out of scope, the object can no longer be referenced and for all purposes, does not exist any more.

Name scoping uses static scoping rules, and names declared in an inner block will hide names declared outside that block. Functions can only be declare at the file level, nested function declarations are not supported.

6. Statements

A *statement* is a single language construct, such as an expression. A *statement_list* is an grouping of statements that are evaluated in the order they are listed. All statements must end

6.1 Control Statements

`if`, `while` and `for` provide a way to control the execution of a *statement_list*.

6.1.1 If statements

`if (` *expression* `) {` *statement_list*$_1$ `}`

`if (` *expression* `) {` *statement_list*$_1$ `} else {` *statement_list*$_2$ `}`

If the *expression* evaluates to true, then the *statement_list*$_1$ is executed, otherwise, *statement_list*$_2$ is executed if present. The open and closing brackets are always required around either *statement_list*.

6.1.2 While statement

`while (` *expression* `) {` *statement_list* `}`

The *expression* is evaluated first, and as long as the *expression* evaluates to true, the *statement_list* will be execute repeatedly. The open and closing brackets are always required around either *statement_list*.

6.2 Break statement

```
Break;
```

The `break` statement only has meaning in the body of a looping statements `while`, `for`. It will terminate the loop and continue execution after the loop body.

6.3 Return statements

`return` *expression* `;`

This will cause any function to stop executing the statements in it's statement_list and return the value of the *expression* to the caller.

7. Data types

These are the supported primitive data types.

Name	Type	Values
`bool`	Truthness	`true` or `false`
`char`	A single character.	Any printable ASCII character.
`string`	A sequence of characters	
`int`	An integral value.	-2^{63} to $2^{63} - 1$
`float`	A floating point number.	-2

7.1 Built in data conversion functions

There are a set of supplied functions that take a single expression and convert the data from the expression into another type. These functions all have the name {*data_type*}`_to_`{*data_type*}. For example, to convert a string into an integer, you would call the function `string_to_int()`. Conversion from a char to a float or a float to a char is not supported.

7.2.2 Map interface

A map's contents can be accessed by using the `[]` operator. The `[]` operator can be used to update the value at a position or retrieve its value in an expression. The key for the key-value pair must be specified inside the operator's brackets as a string.

8. Strings

Strings are objects just like any other data type. They are not arrays of characters. String manipulations are all performed using built in functions.

`substring( string, pos, len ) string` – returns the inner string of the specified `string` start at `pos` (string positions start at 0) for len characters. If the length requested extents past the end of the string, only those characters are returned.

`concat(` $\texttt{string}_1$`,` $\texttt{string}_2$ `) string` – concatenates $\texttt{string}_1$ and $\texttt{string}_2$ to form a new string which is returned.

9. Data streams

`file` *name* = `"`*path*`" sep=`*ch* `output`

There are three types of data streams, input, output and logging. Input data streams are the default. The path can be an absolute path or relative to the current process's directory. Only the *name* and *path* of a `file` statement are required, `sep` and `output` are optional attributes for a data stream.

All data stream except the log data stream operate on Character Separated Value or CSV files. Each line in the file is treated as a record to be processed.

The `sep=`*ch* attribute changes the default separation character from a `','` to *ch*. This can be specified for both input and output data streams, but has no meaning for a log data stream.

The `output` attribute changes the data stream from the default input to an output stream.

Any attempt to read or write to a data stream that fails, will result in the termination of the application with an OS appropriate error message written to standard error.

10. Functions

10.1 Generic functions

> `function` *expression* `(` *param_list* `)` *return_type* `{` *statement_list* `}`

Functions create grouping of statements that can be executed by name. The parameter_list defines a list of variables that have local scope to the function whose values must be provided by the caller. The value of the *return_type* is either the value of the last expression in the *statement_list* or the value of the expression in a `return` statement. The data type of the returned value from the function must match the data type declared in the *return_type*.

10.2 Filter functions

> `filter` *expression* `{` *statement_list* `}`

A filter function has an implicit return_type of `bool`. These functions are used to filter out records that should not be included in the result set from processing the input stream. The filter is called for each record in the input stream. It has a single implicit parameter `rec`, that is a map containing the value of the current record. The elements of the map can be accessed by name or column number. If the function returns false, the record is not included in the output stream.

11. Processing

> `process` *input check_trans_list* `to` *output*

This is the main part of the application. The `process` statement will run each record in the input stream through the list of data checks and transformations in the *check_trans_list*. If the record evaluates to true for all `check` statements it will be written to the output stream with any modifications made by the `trans` statements in the *check_trans_list.* The `check` and `trans` statements are executed in the order they are declared.

11.1 rec map

To access the contents of the current record, an implicit map is created. The keys for the map are the names of the columns in the header row of the input file. If the input file has a `noheader` attribute, then the keys are the column numbers starting with 0 for the first column. The values in the map are the read in values in the CSV row.

11.2 Check statement

> `check {` *expression* `}`

If the expression evaluates to true, then the record passes the check. The code block for the expression is the same as the code block for a `filter` function. A rec map is implicitly defined in the code block.

```
check filter_function
```

The *filter_function* must be a `filter` function. It will be called for each record in the input stream and if the `filter` function returns false, the record will not be included in the output stream.

RTL Project Plan

Process

The design process was adhoc. My understanding of what was needed between the modules that I had defined, was driven more by need as I coded and tested, then from an upfront analysis. This was mainly caused by a lack of understanding of what concrete constructs were needed in each phase of the process. Now that I have written a compiler, I have a road map, crude thou it may be, to guide me in laying out the parts and interfaces to drive the project.

Again, as a single person writing an application, there was no driving force to make an upfront design a requirement. Exploration of the project space by code was more the rule. This caused some issues as modules further down the compiler chain were not explored until the current module was fully coded. In some cases, rework of prior modules was required because my understanding of the needs for the next module changed as I coded. And example would be symbol.ml, it went through many iterations and is still not satisfactory.

Style Guide

Because this project was coded by a single programmer, there was no defined style guide. Code was written in a manner that seemed nature. As my understanding of the ocaml grew, my style of programming changed. But my basic ideas are:

1. Use space not tabs
2. All indents are two spaces
3. Keep line length to under 132 characters
4. Use blank lines to seperate lines of code that work together
5. Prefer "= function" to "match with" constructs
6. 1 to 2 blank lines between functions defined at file scope

Tools

Tool	**Notes**
Vim	Text editiong
Ocaml	Code generation and debugging
Make	Automated build
Python	Automated test scripts
Git	Source control, code stored on BitBucket
Java	To run the resulting jar file

Javap	Dis-assemble javac generated classes to understand how the JVM works.
Javac	Compile runtime classes
Jasmin	Assemble JVM instructions into a java class.
Libreoffice	Documentation
Gs	Conversion of product documents and files into a single pdf file.
Ocaml	Programming language and tool set
Opam	Installing modules
Getopt	Ocaml module installed via opam

Log

Date	Notes
Feb 10	Lex and yacc language definitions started
Mar 05	Setup project plan
Mar 06	Language reference manual
Mar 10	Started coding using microc as the template.
Mar 17	Language tutorial
	Python testing framework created
…	Complete lack of log updates, the rest of the dates are generated from my memory so they are best guesses
April ?	AST dump for test files, this was used to validate my AST creation
April ?	Intermediate representation (byte code) generation
May ?	JVM byte code generation
April to May 14	More testing

RTL Design

The compiler is broken up into independent modules that, for the most part, take an input list of data and proesss that list, providing a new list as input to the next module. The scanner and parser modules don't work this way, they work more in concert to provide the input for the `ir.ml` module.

The final output is an executable jar file. The jvm module does not directly create the jar file, but generates a jvm assembly file.

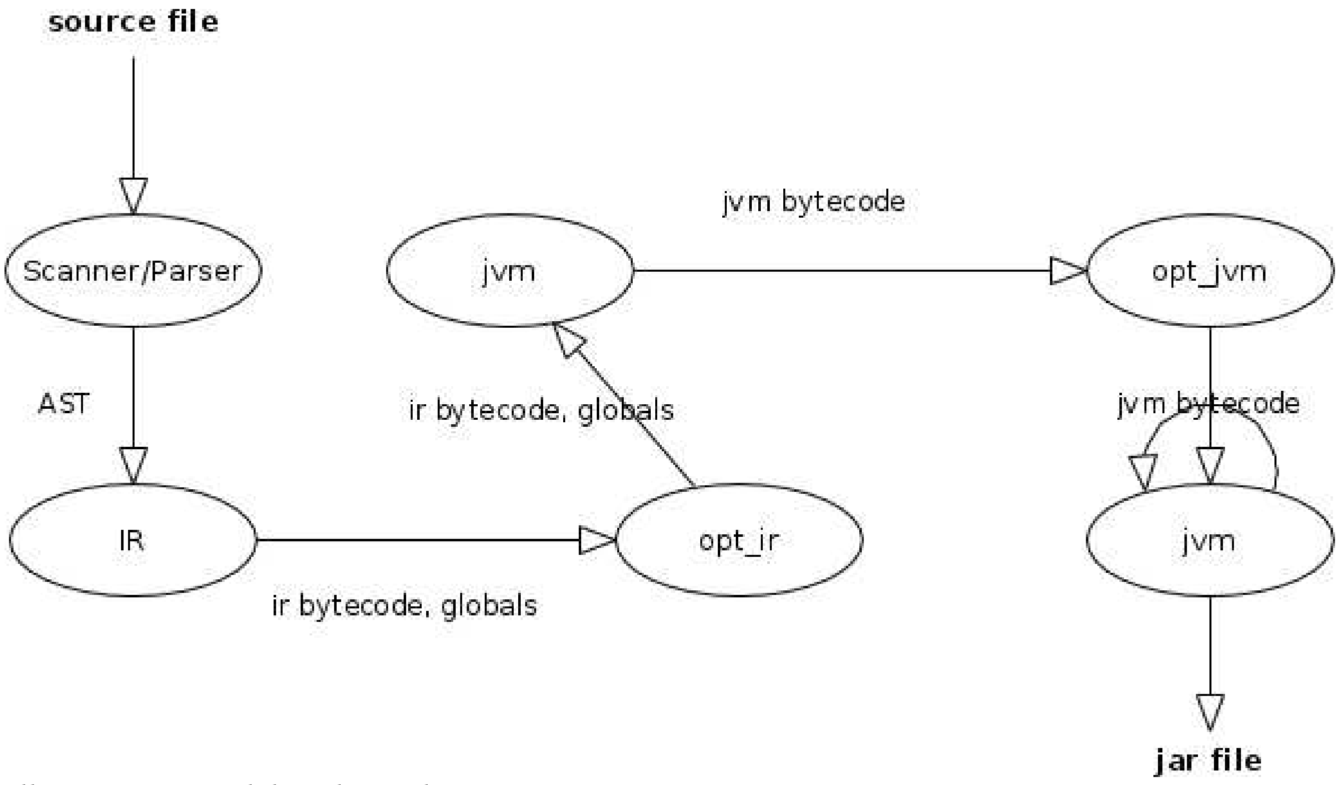

Illustration 1: Module Relationship

Ast

This module defines the object that make up the AST along with some functions that help in using the AST.

Bytecode

This module defines the objects that make up the intermediate representation of the AST along with functions to convert the byte code into a string representation.

Error

This module provides functions to display error messages in a uniform manner.

Symbol

This module defines the what information is in the symbol table and provides the functions to manipulate the symbol table.

Scanner/Parser

The scanner.mll and parser.ly work work together to take the input source file and generate an AST (Abstract Syntax Tree) from the source file. This AST is the output of the scanner/parser module.

The out put from this module is a list of statements; `Ast.stmt list`.

Ir

The input for this module `Ast.stmt list.`

This module takes an AST and converts it into an intermediate byte code representation using the byte codes defined in bytecode.ml. Most of the work going from source code to assembly is done here.

Before any work is done, all the functions are extracted from the program statement list, with the non-function statements becoming the body of the main function. Any variables defined in the main function will become global variables, so that references to them in any of the defined functions will work in the assembled code.

The output from this module is a tuple of intermediate code and a list of globals;

`(Byecode.bstmt list * Symbol.sym_var list).`

Opt_ir

The input and output for this module is `(Byecode.bstmt list * Symbol.sym_var list).` It is setup to perform machine independent optimization. Currently, it is a pass through.

Jvm

This module performs several functions, depending on the application parameters. It can either compile the code in to a runnable jar file, or stop after producing a jvm assembler file suitable for assembling with jasmin.

The jvm code is are all organized into single class with the same name as the input file minus the file extension. Any global variables are setup as fields in the class. All defined functions become static functions for the class. Code not in a function makes up the main function, which is the entry point to the application.

The input to the module is `(Byecode.bstmt list * Symbol.sym_var list).` The `Symbol.sym_var list` is a list of global variables that are setup as fields in the application class. The Bytecode.bstmt list is converted from an intermediate byte code into JVM byte code and written to a temporary file with a .j extension. This file is then compiled into a java class file and a jar file is created with the application class file and the various runtime classes.

Opt_jvm

The input and output for this module is `jvm.jcode list`. It is setup to perform JVM bytecode optimization.   Currently, it is a pass through.

Runtime

The runtime library for RTL is a set of four classes. These classes provide the functionality to the RTL application while hiding the complexity of the java runtime.

Rtl

This class provides functions to create the input and output files, data conversion and map variable access.

RtlException

This is the exception throw by the `RtlInFile` and `RtlOutFile` when create operations fail.

RtlInFile

This contains class encapsulates the java File that is read from in a process statement, and creates the Map record that is passed around in that `process` statement. It also keeps track of the head fields read from the first line of the input file.

RtlOutFile

This contains class encapsulates the java File that is written to in a `process` statement. It will write out the Map record used in the `process` statement after the Map record has been processed..

RTL Test Plan

Example1

The first program that compiled and worked.

test_helloworld.rtl

```
print "Hello World!";
```

test_helloworld.j

```
.source test_helloworld.j
.class public test_helloworld
.super java/lang/Object

.method public <init>()V
      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return
.end method

.method public static main([Ljava/lang/String;)V
      .limit stack 4
      .limit locals 1
      ldc "Hello World!"
      invokestatic Rtl/print(Ljava/lang/String;)V

      return
.end method
```

Example2

This example computes the gcd.

test_gcd.rtl

```
function gcd(int a, int b) int {
  while( a != b ) {
    if ( a < b ) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  return a;
}

print gcd(2, 14);
print gcd(3, 15);
print gcd(99, 121);
```

test_gcd.j

```
.source test_gcd.j
.class public test_gcd
.super java/lang/Object

.method public <init>()V
      aload_0
      invokenonvirtual java/lang/Object/<init>()V
      return
.end method

.method public static gcd(II)I
      .limit stack 32
      .limit locals 2
      ; begin while loop
      goto Label2
Label1:
      iload 0
      iload 1
      if_icmplt Label4
      ; setting variable a
      iload 0
      iload 1
      isub
      istore 0
      goto Label5
Label4:
      ; setting variable b
      iload 1
      iload 0
      isub
      istore 1
Label5:
Label2:
      iload 0
      iload 1
      if_icmpne Label1
Label3:
      ; end while loop
      iload 0
      ireturn
.end method

.method public static main([Ljava/lang/String;)V
      .limit stack 14
      .limit locals 1
      ; setting up call to gcd
      ldc 2
      ldc 14
      invokestatic gcd(II)I
      invokestatic Rtl/int_to_string(I)Ljava/lang/String;
      invokestatic Rtl/print(Ljava/lang/String;)V
      ; setting up call to gcd
      ldc 3
      ldc 15
      invokestatic gcd(II)I
      invokestatic Rtl/int_to_string(I)Ljava/lang/String;
      invokestatic Rtl/print(Ljava/lang/String;)V
      ; setting up call to gcd
      ldc 99
      ldc 121
      invokestatic gcd(II)I
      invokestatic Rtl/int_to_string(I)Ljava/lang/String;
      invokestatic Rtl/print(Ljava/lang/String;)V

      return
.end method
```

Test Suite

```
fail_assignment0.rtl
fail_assignment1.rtl
fail_assignment2.rtl
fail_assignment3.rtl
fail_function0.rtl
fail_function1.rtl
fail_function2.rtl
fail_function3.rtl
fail_function4.rtl
fail_function5.rtl
fail_function_params0.rtl
fail_function_params1.rtl
fail_function_params2.rtl
fail_function_params3.rtl
fail_function_params4.rtl
fail_function_params5.rtl
fail_function_params6.rtl
test_assignment.rtl
test_bare_types.rtl
test_break.rtl
test_comments.rtl
test_example1.rtl
test_example2.rtl
test_example3.rtl
test_example8.rtl
test_file_output.rtl
test_file.rtl
test_file_sep.rtl
test_filter.rtl
test_function_empty.rtl
test_function_params.rtl
test_gcd.rtl
test_global.rtl
test_helloworld.rtl
test_print.rtl
test_process.rtl
test_string_concat.rtl
test_string_substr.rtl
```

`test_types_value.rtl`

`test_var_strings.rtl`

`test_loop.rtl`

A test file was created contains a set of tests for either a single statement or expression or more than one related statements or expressions.

The output from the test was compared against a reference document and if they were the same, the test was considered successful. Tests files that begin with “fail_” should not compile, they contain a syntax error that should be caught by the compiler.

The test suite was run by a python script, `rtl_check.py`. The shell script `rtl_check` setup and ran the python script.

RTL Lessons Learned

Lesson 1: Writing a compiler is a battle against the nastiest type of bugs found in computer science.

Lesson 2: Java is full of unexplained nuances, documentation and google don't always agree or have an answer.

Lesson 3: Given lessons 1 and 2, your planned start date is not early enough.

Writing a compiler turned out to be a lot of fun, just not at the end. This is one of those programs where sitting down and thinking for a while at each turn is needed to get things done in a manner that is both correct and elegant. This project did not have that kind of time.

Additionally, working by myself, with no fellow students to bounce idea's off of what a major hindrance, especially in the beginning. I should have attempted to reach out to my fellow CVN students early and make connections. This being my first CVN class, I was not sure what to expect on a inter student level. And that led me to take a wait and see approach.

If I had to do this project again, I would have not used the JVM as my runtime platform. Jasmin's documentation was not really adequate to learning how to use the tool, except through trial and error for the most part, which took a large chunk of time. Time better spend on testing the compiler.

I learned a lot in this class, too bad there is not a PLT 2 for an other semester to follow up with the more interesting stuff, the optimization techniques and the like.

I was not sure, based on the syllabus, if you wanted the actual source code and project files, in addition to this document. So I have uploaded a tar file with the project files as project 2.

Cheers and thanks for a great class. I just wish I could have done it in person.

```
(************************************************************************
 *
 * File: scanner.mll
 *
 * Purpose: language lex specification
 *
 *)

{
open Parser

let update_linenum lexbuf =
  let pos = lexbuf.Lexing.lex_curr_p in
    lexbuf.Lexing.lex_curr_p <- { pos with
      Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
      Lexing.pos_bol = pos.Lexing.pos_cnum;
    }
}

let white_space = [' ' '\012' '\t' '\n' '\r']
let digit       = ['0'-'9']
let exponent    = "e" ['-''+']? digit+
let int_lit     = '-'? digit+
let float_lit   = (int_lit "." digit+) exponent?
let string_lit  = '"' ('\\' '"' | [^ '"'])* '"'
let char_lit    = '\'' _ '\''
let true_lit    = "true"
let false_lit   = "false"
let id_start    = ['a'-'z' 'A'-'Z' '_' ]
let id_other    = ['a'-'z' 'A'-'Z' '_' '0'-'9' ]
let id          = id_start id_other*
let comment     = '#' [^ '\n']*

rule token = parse
  [' ' '\t' '\r'] { token lexbuf } (* Whitespace *)
| '\n'          { update_linenum lexbuf; token lexbuf }
| comment       { update_linenum lexbuf; token lexbuf }           (* Comments *)
| '('           { LPAREN }
| ')'           { RPAREN }
| '{'           { LBRACE }
| '}'           { RBRACE }
| '['           { LBRACK }
| ']'           { RBRACK }
| ';'           { SEMI }
| ':'           { COLON }
| ','           { COMMA }
| '+'           { PLUS }
| '-'           { MINUS }
| '*'           { TIMES }
| '/'           { DIVIDE }
| '='           { ASSIGN }
| "=="          { EQ }
| "!="          { NEQ }
| '<'           { LT }
| "<="          { LEQ }
| ">"           { GT }
| ">="          { GEQ }
| "break"       { BREAK }
| "check"       { CHECK }
| "else"        { ELSE }
| "file"        { FILE }
| "filter"      { FILTER }
| "for"         { FOR }
| "function"    { FUNCTION }
| "if"          { IF }
| "output"      { OUTPUT }
| "print"       { PRINT }
| "process"     { PROCESS }
| "sep"         { SEP }
| "return"      { RETURN }
| "to"          { TO }
| "while"       { WHILE }

| "char"        { CHAR }
| "float"       { FLOAT }
| "int"         { INT }
| "string"      { STRING }

| float_lit as value             { LIT_FLOAT(value) }
| int_lit as value               { LIT_INT(value) }
| string_lit as value            { LIT_STR(value) }
| char_lit as value              { LIT_CHAR(value) }
| true_lit as value              { LIT_BOOL(value) }
| false_lit as value             { LIT_BOOL(value) }
| id as value                    { ID(value) }
| eof                            { EOF }

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

```
/**************************************************************************
 *
 * File: parser.mly
 *
 * Purpose: language yacc definition
 *
 */

%{ open Ast %}

%token SEMI COLON LPAREN RPAREN LBRACK RBRACK MBEGIN MEND LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN EQ NEQ LT LEQ GT GEQ LAND LOR
%token FILE OUTPUT SEP PRINT BREAK CHECK ELSE FILE FILTER FOR FUNCTION IF LOG
PROCESS RETURN TO WHILE
%token CHAR FLOAT INT STRING BOOL

%token <string> LIT_FLOAT
%token <string> LIT_INT
%token <string> LIT_STR
%token <string> LIT_CHAR
%token <string> LIT_BOOL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN
%left LAND LOR
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

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

process_list:
    /* empty */                    { [] }
  | process_list process_stmt      { $2 :: $1 }

process_stmt:
    CHECK check_stmt               { $2 }

check_stmt:
  | ID                             { Check_Function($1) }

formal_list:
    /* nothing */                  { [] }
  | formal_item                    { [$1] }
  | formal_list COMMA formal_item { $3 :: $1 }
```

```
formal_item:
  | BOOL ID                     { Variable(Bool, $2, Noexpr) }
  | CHAR ID                     { Variable(Char, $2, Noexpr) }
  | FLOAT ID                    { Variable(Float, $2, Noexpr) }
  | INT ID                      { Variable(Int, $2, Noexpr) }
  | STRING ID                   { Variable(String, $2, Noexpr) }

vdecl:
  | BOOL ID SEMI                { Variable(Bool, $2, Noexpr) }
  | BOOL ID ASSIGN expr SEMI    { Variable(Bool, $2, $4) }
  | CHAR ID SEMI                { Variable(Char, $2, Noexpr) }
  | CHAR ID ASSIGN expr SEMI    { Variable(Char, $2, $4) }
  | FLOAT ID SEMI               { Variable(Float, $2, Noexpr) }
  | FLOAT ID ASSIGN expr  SEMI  { Variable(Float, $2, $4) }
  | INT ID SEMI                 { Variable(Int, $2, Noexpr) }
  | INT ID ASSIGN expr SEMI     { Variable(Int, $2, $4) }
  | STRING ID SEMI              { Variable(String, $2, Noexpr) }
  | STRING ID ASSIGN expr SEMI  { Variable(String, $2, $4) }

variable_type:
  | BOOL                        { Char }
  | CHAR                        { Char }
  | FLOAT                       { Float }
  | INT                         { Int }
  | STRING                      { String }

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

file_opts:
    /* nothing */               { [] }
  | fopt_list                   { List.rev $1 }

fopt_list:
    fopt                        { [$1] }
  | fopt_list fopt              { $2 :: $1 }

fopt:
    OUTPUT                      { Output }
  | SEP ASSIGN LIT_STR          { Sep($3) }

stmt:
    expr SEMI                   { Expr($1) }
  | BREAK SEMI                  { Break }
  | RETURN expr SEMI            { Return($2) }
  | LBRACE stmt_list RBRACE     { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE
                                { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt
                                { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
                                { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

  | FUNCTION ID LPAREN formal_list RPAREN variable_type LBRACE stmt_list RBRACE
                                { Function($2, List.rev $4, $6, Block(List.rev
$8)) }
```

```
  | FILTER ID LBRACE stmt_list RBRACE
                                  { FilterFunc($2, Block(List.rev $4)) }
  | PROCESS ID process_list TO ID SEMI
                                  { Process($2, List.rev $3, $5) }
  | vdecl                         { Declare($1) }
  | LBRACE error RBRACE           { Expr(Noexpr) }
  | FILE ID ASSIGN LIT_STR file_opts SEMI
                                  { File($2, $4, $5) }
  | PRINT expr SEMI               { Print($2) }

/*
str_format:
    expr                          { [$1] }
  | str_format COMMA expr         { $3 :: $1 }
*/

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

expr:
    LIT_INT                             { Literal_Int($1) }
  | LIT_FLOAT                           { Literal_Float($1) }
  | LIT_STR                             { Literal_String($1) }
  | LIT_CHAR                            { Literal_Char($1) }
  | LIT_BOOL                            { Literal_Bool($1) }
  | ID                                  { Id($1) }
  | expr PLUS   expr                    { Binop($1, Add,   $3) }
  | expr MINUS  expr                    { Binop($1, Sub,   $3) }
  | expr TIMES  expr                    { Binop($1, Mult,  $3) }
  | expr DIVIDE expr                    { Binop($1, Div,   $3) }
  | expr EQ     expr                    { Binop($1, Equal, $3) }
  | expr NEQ    expr                    { Binop($1, Neq,   $3) }
  | expr LT     expr                    { Binop($1, Less,  $3) }
  | expr LEQ    expr                    { Binop($1, Leq,   $3) }
  | expr GT     expr                    { Binop($1, Greater, $3) }
  | expr GEQ    expr                    { Binop($1, Geq,   $3) }
  | expr LAND   expr                    { Binop($1, LAnd,  $3) }
  | expr LOR    expr                    { Binop($1, LOr,   $3) }
  | expr ASSIGN expr                    { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN        { Call($1, $3) }
  | LPAREN expr RPAREN                  { $2 }
  | ID LBRACK expr RBRACK               { MapAccess($1, $3) }

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

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

%%

let parse_error s =
  print_endline ("error: {" ^ s ^ "} TODO")
```

```
(**************************************************************************
 *
 * File: sat.ml
 *
 * Purpose: defines the objects in the abstract syntax tree.  Also provides
 * utility methods to convert the ast objects into a string representation.
 *
 *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | LAnd
| LOr | NotNull

type expr =
    Literal_Int of string
  | Literal_Float of string
  | Literal_String of string
  | Literal_Char of string
  | Literal_Bool of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of expr * expr
  | Call of string * expr list
  | MapAccess of string * expr
  | Noexpr

and map_entry = MapEntry of expr * expr
and file_options = Sep of string | Output

type expr_type = Bool | Char | Float | Int | Map | String | Void | StringArray |
FileIn | FileOut

(* Data Type * ID * value *)
type variable =
    Variable of expr_type * string * expr

type process_steps = Check_Function of string

and stmt =
    Block of stmt list
  | Break
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Function of string * variable list * expr_type * stmt
  | FilterFunc of string * stmt
  | Process of string * process_steps list * string
  | Declare of variable
  | File of string * string * file_options list
  | Print of expr

type program = stmt list

let var_list_to_type_list lst =
  let extract_type = function
    | Variable(Char, n, _) -> Char
    | Variable(Float, n, _) -> Float
    | Variable(Int, n, _) -> Int
    | Variable(String, n, _) -> String
    | Variable(Bool, n, _) -> Bool
    | Variable(Map, n, _) -> Map
    | Variable(Void, n, _) -> Void
    | Variable(StringArray, n, _) -> StringArray
    | Variable(FileIn, n, _) -> FileIn
    | Variable(FileOut, n, _) -> FileOut
  in
  List.map extract_type lst


let string_of_op = function
    Add     -> "+"
  | Sub     -> "-"
  | Mult    -> "*"
  | Div     -> "/"
  | Equal   -> "=="
  | Neq     -> "!="
  | Less    -> "<"
  | Leq     -> "<="
  | Greater -> ">"
  | Geq     -> ">="
  | LAnd    -> "&&"
  | LOr     -> "||"
  | NotNull -> "notnull"

let string_of_type = function
  | Char -> "char"
  | Float -> "float"
  | Int -> "int"
  | String -> "string"

  (* internal types *)
  | Bool -> "bool"
  | Map -> "map"
  | Void -> "void"
  | StringArray -> "String[]"
  | FileIn -> "FileIn"
  | FileOut -> "FileOut"

let string_of_type_list l =
  String.concat ", " (List.map string_of_type l)


let string_of_formal = function
  | Variable(Char, n, _) -> "char " ^ n
  | Variable(Float, n, _) -> "float " ^ n
  | Variable(Int, n, _) -> "int " ^ n
  | Variable(String, n, _) -> "string " ^ n
  | Variable(Bool, n, _) -> "string " ^ n

  (* internal types, not used in rtl*)
  | Variable(Map, n, _) -> "internal-type-map " ^ n
  | Variable(Void, n, _) -> "internal-type-void " ^ n
  | Variable(StringArray, n, _) -> "internal-type-string[] " ^ n
  | Variable(FileIn, n, _) -> "internal-type-filein " ^ n
  | Variable(FileOut, n, _) -> "internal-type-fileout " ^ n
```

```
let string_of_formal_list l =
  String.concat ", " (List.map string_of_formal l)

let rec string_of_expr = function
    Literal_Int(l) -> l
  | Literal_Float(l) -> l
  | Literal_String(l) -> l
  | Literal_Char(l) -> l
  | Literal_Bool(l) -> l
  | Id(s) -> s
  | Binop(lhs, o, rhs) -> string_of_expr lhs ^ " " ^ string_of_op o ^ " " ^
string_of_expr rhs
  | Assign(lhs, rhs) -> string_of_expr lhs ^ " = " ^ string_of_expr rhs
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | MapAccess(name, prop) -> name ^ "[" ^ string_of_expr prop ^ "]"
  | Noexpr -> ""

let string_of_expr_list l =
  String.concat ", " (List.map string_of_expr l)

let string_of_declare = function
    Variable(Bool, n, Noexpr) -> "bool " ^ n ^ ";"
  | Variable(Bool, n, e) -> "bool " ^ n ^ " = " ^ string_of_expr e ^ ";"
  | Variable(Char, n, Noexpr) -> "char " ^ n ^ ";"
  | Variable(Char, n, e) -> "char " ^ n ^ " = " ^ string_of_expr e ^ ";"
  | Variable(Float, n, Noexpr) -> "float " ^ n ^ ";ocaml toplevel load file"
  | Variable(Float, n, e) -> "float " ^ n ^ " = " ^ string_of_expr e ^ ";"
  | Variable(Int, n, Noexpr) -> "int " ^ n ^ ";"
  | Variable(Int, n, e) -> "int " ^ n ^ " = " ^ string_of_expr e ^ ";"
  | Variable(String, n, Noexpr) -> "string " ^ n ^ ";"
  | Variable(String, n, e) -> "string " ^ n ^ " = " ^ string_of_expr e ^ ";"
  | Variable(Void,_,_) -> raise (Invalid_argument("Void type not supported"))
  | Variable(Map,_,_) -> raise (Invalid_argument("Map type not supported"))
  | Variable(StringArray,_,_) -> raise (Invalid_argument("String[] type not
supported"))
  | Variable(FileIn,_,_) -> raise (Invalid_argument("File type not supported"))
  | Variable(FileOut,_,_) -> raise (Invalid_argument("File type not supported"))
  (*
  | Variable(Map, n, Noexpr) -> "map " ^ n ^ ";"
  | Variable(Map, n, e) -> "map " ^ n ^ " = " ^ string_of_expr e ^ ";"
  *)
let string_of_file_option = function
    Output -> "Output"
  | Sep(s) -> "Sep(" ^ s ^ ")"

let rec string_of_file_options = function
    [] -> ""
  | [v] -> " " ^ string_of_file_option v
  | hd :: tl -> " " ^ string_of_file_option hd ^ string_of_file_options tl


let rec string_of_stmt = function
    Expr(expr) -> "Expr(" ^ string_of_expr expr ^ ")"
  | Break -> "Break"
  | Return(expr) -> "Return(" ^ string_of_expr expr ^ ")";
  | Block(stmts) -> "Block(" ^ string_of_stmt_list stmts ^ ")"
  | If(e, s1, s2) -> "If(" ^ string_of_expr e ^ ", " ^ string_of_stmt s1 ^ ", " ^
atring_of_stmt s2 ^ ")"
  | For(e1, e2, e3, s) ->
      "For(" ^ string_of_expr e1  ^ ", " ^ string_of_expr e2 ^ ", " ^
string_of_expr e3 ^ ", " ^ string_of_stmt s ^ ")"
  | While(e, s) -> "While(" ^ string_of_expr e ^ ", " ^ string_of_stmt s ^ ")"
  | Function(n,vl,rt,b) ->
      "Function(" ^ n ^ ", " ^ string_of_formal_list vl ^ ", " ^ string_of_type rt
^ ", " ^ string_of_stmt b ^ ")"
  | FilterFunc(n,b) -> "Filter(" ^ n ^ ", " ^ string_of_stmt b ^ ")"
  | Process(f,p,t) -> "Process(" ^ f ^ ", " ^ string_of_process_steps p ^ ", " ^ t
^ ")"
  | Declare(v) -> "Declare(" ^ string_of_declare v ^ ")"
  | File(n,p,o) -> "File(" ^ n ^ ", " ^ p ^ ", [" ^ string_of_file_options o ^
"])"
  | Print(expr) -> "Print(" ^ string_of_expr expr ^ ")"

and string_of_process_step = function
  | Check_Function(n) -> "Check_Function(" ^ n ^ ")"

and string_of_process_steps l =
  "[" ^ String.concat ", " (List.map string_of_process_step l) ^ "]"

and string_of_stmt_list stmts =
  "[" ^ String.concat ", " (List.map string_of_stmt stmts) ^ "]"

let string_of_program stmt_list  =
  List.map string_of_stmt stmt_list
```

```
(***************************************************************************
 *
 * File: error.ml
 *
 * Purpose: display error messages in a uniform format
 *
 *)


let report msg =
  prerr_endline ("error: " ^ msg)


(*
 * Display an error message and return the stmt error state
 *)
let stmt_error msg =
  prerr_endline ("error: " ^ msg)
  (*[Bytecode.Halt]*)

(*
 * Display an error message and return the expr error state
 *)
let expr_error msg =
  prerr_endline ("error: " ^ msg)
  (*([Bytecode.Halt], Ast.Void)*)

(*
 * Something happened that should never happen, exit and start debugging...
 *)
let internal_error msg =
  prerr_endline ("internal error: " ^ msg);
  ignore (exit 999)
```

```
(*************************************************************************
 *
 * File: asmbol.ml
 *
 * Purpose: defines and contains the multiple symbol tables:
 *  functions, globals, local variables
 *
 *)

type sym_var  = {vtype: Ast.expr_type; address: int; scope: int}
type sym_func = {args : Ast.expr_type list; rtn: Ast.expr_type; fid: string}
type environment = {
  scope: int;
  label: int;
  mutable vars: string list;
  mutable address: int;
  mutable rec_address: int;
  rtype: Ast.expr_type
}

exception Duplicate_id of string

(*
 * Seperate namespace for functions and variables
 *  All variable are stack variables, there are no global variables
 *)

let functions = Hashtbl.create 109
let global_vars = Hashtbl.create 109
let symbol_table = Hashtbl.create 109

(*************************************************************************
 * Return the next address for stored variables
 *)
let next_address env =
  let addr = env.address in
  ignore(env.address <- succ env.address);
  addr

let init () =
  let var = {vtype = Ast.Int; address = 0; scope = 0} in
  Hashtbl.add symbol_table "__force_table_setup__" var;
  Hashtbl.remove symbol_table "__force_table_setup__";

  Hashtbl.add functions "int_to_string"
    {args=[Ast.Int]; rtn=Ast.String; fid = "Rtl/int_to_string"};
  Hashtbl.add functions "int_to_float"
    {args=[Ast.Int]; rtn=Ast.Float; fid = "Rtl/int_to_float"};
  Hashtbl.add functions "int_to_char"
    {args=[Ast.Int]; rtn=Ast.Char; fid = "Rlt/int_to_char"};
  Hashtbl.add functions "char_to_string"
    {args=[Ast.Char]; rtn=Ast.String; fid = "Rtl/char_to_string"};
  Hashtbl.add functions "char_to_int"
    {args=[Ast.Char]; rtn=Ast.Int; fid = "Rtl/char_to_int"};
  Hashtbl.add functions "string_to_int"
    {args=[Ast.String]; rtn=Ast.Int; fid = "Rtl/string_to_int"};
  Hashtbl.add functions "string_to_float"
    {args=[Ast.String]; rtn=Ast.Float; fid = "Rtl/string_to_float"};
  Hashtbl.add functions "string_to_char"
    {args=[Ast.String]; rtn=Ast.Char; fid = "Rtl/string_to_char"};
  Hashtbl.add functions "float_to_string"
    {args=[Ast.Float]; rtn=Ast.String; fid = "Rtl/float_to_string"};
  Hashtbl.add functions "float_to_int"
    {args=[Ast.Float]; rtn=Ast.Int; fid = "Rtl/float_to_int"};
  Hashtbl.add functions "substr"
    {args=[Ast.String; Ast.Int; Ast.Int]; rtn=Ast.String; fid = "Rtl/substr"};
  Hashtbl.add functions "concat"
    {args=[Ast.String; Ast.String]; rtn=Ast.String; fid = "Rtl/concat"};
  Hashtbl.add functions "print"
    {args=[Ast.String]; rtn=Ast.Void; fid = "Rtl/print"}


(*
 *  We have a list of symbol tables(Hash tables), we add a new one each time we
enter
 *  a new block of code, which creates a new scope.  When we leave the that
 *  scope, we remove the added symbol table
 *)

(****************************************************************************
 * Return an empty environment
 *)
let create_initial_env () =
  init ();
  {scope = -1; vars = []; label = -1; address = 0; rec_address = -1; rtype =
Ast.Void}


(****************************************************************************
 * Create an empty environment at the top of the list of environments
 *)
let push_env env label naddr rtype =
  {
    scope = succ env.scope;
    vars = [];
    label = env.label;
    address = naddr;
    rec_address = env.rec_address;
    rtype = rtype
  }

let pop_env env =
  ignore(List.iter (fun elt -> Hashtbl.remove symbol_table elt) env.vars)


let add_local_var env name typ =
  try
    let var = Hashtbl.find symbol_table name in
    if var.scope = env.scope then
      raise (Duplicate_id name)
    else
      var
  with Not_found -> begin
    let addr = next_address env in
    let var = {vtype = typ; address = addr; scope = env.scope} in
    ignore(env.vars <- name :: env.vars);
    Hashtbl.add symbol_table name var;
    var
  end


(****************************************************************************
 * Add an variable to the curent environment
 *)
let add_var env name typ =
  try
    let var = if env.scope = 0 then
      begin
        ignore(Hashtbl.find global_vars name);
        raise (Duplicate_id name)
      end
    else
      Hashtbl.find symbol_table name in
    if var.scope = env.scope then
      raise (Duplicate_id name)
    else
      var
  with Not_found -> begin
    if env.scope = 0 then
      let var = {vtype = typ; address = 0; scope = env.scope} in
      Hashtbl.add global_vars name var;
      var
    else
      let addr = next_address env in
      let var = {vtype = typ; address = addr; scope = env.scope} in

      ignore(env.vars <- (name :: env.vars));
      Hashtbl.add symbol_table name var;
      var
  end

(****************************************************************************
 * Return the variable if it exists in the immediate scope only
 *)
let find_local_var env name =
  let var = if env.scope = 0 then
    Hashtbl.find global_vars name
  else
    Hashtbl.find symbol_table name in

  if var.scope = env.scope then
    var
  else
    raise Not_found

(****************************************************************************
 *  Return variable if it exists some where in the layers of scope
 *)
let find_var env name =
  if env.scope = 0 then
    Hashtbl.find global_vars name
  else
    Hashtbl.find symbol_table name
```

```
(****************************************************************************
 * find the function record
 *)
let find_function name =
  Hashtbl.find functions name

let add_func name args rtn =
  try
    ignore(Hashtbl.find functions name);
    raise (Duplicate_id name)
  with Not_found ->
    Hashtbl.add functions name {args = args; rtn = rtn; fid = name}
```

```
(***************************************************************************
 *
 * File: rytecode.ml
 *
 * Purpose: defines the intermediate representation operations and serveral
 * utility functions.
 *
 *)

type bstmt =
  | Dup
  | Swap
  | Pushb of string (* Push a literal boolean *)
  | Pushc of string (* Push a literal char *)
  | Pushf of string (* Push a literal float *)
  | Pushs of string (* Push a literal string index *)
  | Pushi of string (* Push a literal int *)
  | Pop             (* Discard a value *)
  | Binop of Ast.op * Ast.expr_type (* Perform arithmetic on top of stack *)
  | Cmp of Ast.op * Ast.expr_type * int   (* Perform comparison on top of stack *)
  | Load of int * Ast.expr_type
  | Store of int * Ast.expr_type
  | Call of string * bool (* bool is flag to signify need for fixup *)
  | CallV of string * bool (* bool is flag to signify need for fixup *)

    (* name, param list, return type, stack size, local var count *)
  | Bfunc of string * Ast.expr_type list * Ast.expr_type * int * int
  | Efunc of string
  | Return of Ast.expr_type
  | Beq of int      (* Branch relative if top-of-stack is zero *)
  | Bne of int      (* Branch relative if top-of-stack is non-zero *)
  | Jump of int     (* Branch relative *)
  | Label of int    (* Jump/Branch to location *)
  | Halt            (* Terminate *)
  | Comment of string (* Comment *)
  | Nop
  | GetGlobal of string * Ast.expr_type
  | PutGlobal of string * Ast.expr_type

type field =
  | Int of string * string
  | Char of string * string
  | Float of string * string
  | String of string * string
  | FileIn of string * string
  | FileOut of string * string

let store_of name (var : Symbol.sym_var) =
  if var.Symbol.scope = 0 then
    PutGlobal(name, var.Symbol.vtype)
  else
    Store(var.Symbol.address, var.Symbol.vtype)

let load_of name (var : Symbol.sym_var) =
  if var.Symbol.scope = 0 then
    GetGlobal(name, var.Symbol.vtype)
  else
    Load(var.Symbol.address, var.Symbol.vtype)
```

```
let push_of = function
  | Ast.Bool -> Pushb("false")
  | Ast.Char -> Pushc(" ")
  | Ast.Float -> Pushf("0.0")
  | Ast.Int -> Pushi("0")
  | Ast.Map -> Halt  (* not supported *)
  | Ast.String -> Pushs("")
  (* internal types *)
  | Ast.Void -> Nop
  | Ast.StringArray -> Nop
  | Ast.FileIn -> Nop
  | Ast.FileOut -> Nop


let string_of_instr = function
  | Dup -> "dup"
  | Swap -> "swap"
  | Pushb(s) -> "pushb " ^ s
  | Pushc(s) -> "pushc " ^ s
  | Pushf(s) -> "pushf " ^ s
  | Pushs(s) -> "pushs " ^ s
  | Pushi(s) -> "pushi " ^ s
  | Pop -> "pop"
  | Binop(Ast.Add, typ) -> "add"
  | Binop(Ast.Sub, typ) -> "sub"
  | Binop(Ast.Mult, typ) -> "mul"
  | Binop(Ast.Div, typ) -> "div"
  | Binop(op, _) -> "internal error, Binop used with unsupported op " ^
Ast.string_of_op op
  | Cmp(Ast.Equal, typ, l) -> "cmpEq " ^ string_of_int l
  | Cmp(Ast.Neq, typ, l) -> "cmpNeq " ^ string_of_int l
  | Cmp(Ast.Less, typ, l) -> "cmpLt " ^ string_of_int l
  | Cmp(Ast.Leq, typ, l) -> "cmpLeq " ^ string_of_int l
  | Cmp(Ast.Greater, typ, l) -> "cmpGt " ^ string_of_int l
  | Cmp(Ast.Geq, typ, l) -> "cmpGeq " ^ string_of_int l
  | Cmp(Ast.NotNull, typ, l) -> "notNull " ^ string_of_int l
  | Cmp(op, typ, _) -> "internal error, Cmp used with unsupported op " ^
Ast.string_of_op op
  | Load(i, _) -> "load " ^ string_of_int i
  | Store(i, _) -> "store " ^ string_of_int i
  | PutGlobal(name, typ) -> "putglobal " ^ name ^ ", " ^ Ast.string_of_type typ
  | GetGlobal(name, typ) -> "getglobal " ^ name ^ ", " ^ Ast.string_of_type typ
  | Call(s,_) -> "callstatic " ^ s
  | CallV(s,_) -> "callvirtual " ^ s
  | Bfunc(n,p,r,s,l) ->
      Printf.sprintf "function %s(%s) %s <stack=%d, locals=%d>" n
        (Ast.string_of_type_list p) (Ast.string_of_type r) s l
  | Efunc(s) -> "end " ^ s ^ "\n"
  | Return(typ) -> "return " ^ Ast.string_of_type typ
  | Bne(i) -> "bne L" ^ string_of_int i
  | Beq(i) -> "beq L" ^ string_of_int i
  | Jump(i) -> "jump L" ^ string_of_int i
  | Label(i) -> "L" ^ string_of_int i ^ ":"
  | Halt    -> "halt"
  | Comment(s) -> "; " ^ s
  | Nop -> "Nop"
```

```
let string_of_program (ir, globals) =
  List.map (fun instr -> "\t" ^ string_of_instr instr) ir
```

```
(*********************************************************************
 *
 * File: ir.ml
 *
 * Purpose: convert the abstract syntax tree representation of the program
 * into bytecode.  Syntax checking and type checking are also done at this
 * stage.
 *
 *)

let dump_program = function
  | Ast.Expr(expr) -> "Expr"
  | Ast.Break -> "Break"
  | Ast.Return(expr) -> "Return"
  | Ast.Block(stmts) -> "Block"
  | Ast.If(e, s1, s2) -> "If"
  | Ast.For(e1, e2, e3, s) -> "For"
  | Ast.While(e, s) -> "While"
  | Ast.Function(n,vl,rt,b) -> "Function"
  | Ast.FilterFunc(n,b) -> "FilterFunc"
  | Ast.Process(f,p,t) -> "Process"
  | Ast.Declare(v) -> "Declare"
  | Ast.File(n,p,o) -> "File"
  | Ast.Print(el) -> "Print"



(*********************************************************************
 * Get the name of the statement, used for error messages
 *)
let string_of_stmt = function
  | Ast.Block(_) -> "code block {...}"
  | Ast.Break -> "break"
  | Ast.Expr(_) -> "expression"
  | Ast.Return(_) -> "return"
  | Ast.If(_,_,_) -> "if"
  | Ast.For(_,_,_,_) -> "for"
  | Ast.While(_,_) -> "while"
  | Ast.Function(_,_,_,_) -> "function"
  | Ast.FilterFunc(_,_) -> "filter"
  | Ast.Process(_,_,_) -> "process"
  | Ast.Declare(_) -> "declare"
  | Ast.File(_,_,_) -> "file"
  | Ast.Print(_) -> "print"

let string_of_expr_eval (code, typ) =
  let module BC = Bytecode in
  match typ with
    | Ast.String -> code
    | Ast.Int -> code @ [BC.Call("rtl/int_to_string(I)Ljava/lang/String;", false)]
    | Ast.Float -> code @ [BC.Call("rtl/float_to_string(F)Ljava/lang/String;",
false)]
    | Ast.Char -> code @ [BC.Call("rtl/char_to_string(C)Ljava/lang/String;",
false)]
    | _ as typ ->
        (Error.internal_error ("cannot convert a '" ^ Ast.string_of_type typ ^ "'
to a string");
        [BC.Halt])
```

```
let calc_stack_limit code =
  let module BC = Bytecode in
  let instr_stack_usage = function
    | BC.Dup -> 1
    | BC.Pushb(_) -> 1
    | BC.Pushc(_) -> 1
    | BC.Pushf(_) -> 1
    | BC.Pushi(_) -> 1
    | BC.Pushs(_) -> 1
    | BC.Pop -> -1
    | BC.Binop(_,_) -> 2
    | BC.Cmp(_,_,_) -> 2
    | BC.Load(_,_)  -> 1
    | BC.Store(_,_) -> -1
    | BC.GetGlobal(_,_)  -> 1
    | BC.PutGlobal(_,_) -> -1
    | BC.Call(_,_) -> 0 (* this should use pop values from the stack, but for now
ignore that *)
    | BC.CallV(_,_) -> 0 (* this should use pop values from the stack, but for now
ignore that *)
    | BC.Bfunc(_,_,_,_,_) -> 0
    | BC.Efunc(_) -> 0
    | BC.Return(_) -> -1
    | BC.Beq(_) -> -1
    | BC.Bne(_) -> -1
    | BC.Jump(_) -> 0
    | BC.Label(_) -> 0
    | BC.Halt -> 0
    | BC.Comment(_) -> 0
    | BC.Swap -> 0
    | BC.Nop -> 0
  in
  List.fold_left (fun a e ->
      let v = (fst a) + e in
      ((v + e), (max (v + e) (snd a)))
      ) (0,0) (List.map instr_stack_usage code)

let calc_local_limit code =
  let module BC = Bytecode in
  let local_index = function
    | BC.Load(i,_)  -> i
    | BC.Store(i,_) -> i
    | _ -> 0
  in
  List.fold_left max 0 (List.map local_index code) + 1

(*****************************************************************************
 * Keep track of the next lable number, and return it incrementing the
 * counter.
 *)
let label_cntr = ref 0

let next_label () =
  label_cntr := succ !label_cntr;
  !label_cntr

(*****************************************************************************
 * Return the instruction to push the value onto the stack by var type
 *)
let push_var_instr typ value =
  let module BC = Bytecode in
  match typ with
  | Ast.Char    -> BC.Pushc(value)
  | Ast.Float   -> BC.Pushf(value)
  | Ast.Int     -> BC.Pushi(value)
  | Ast.String  -> BC.Pushs(value)

  (* internal types *)
  | Ast.Map     ->
    Error.internal_error "unable to push type map";
    BC.Halt
  | Ast.Void ->
    Error.internal_error "unable to push type void";
    BC.Halt
  | Ast.StringArray ->
    Error.internal_error "unable to push type string[]";
    BC.Halt
  | Ast.FileIn ->
    Error.internal_error "unable to push type FileIn";
    BC.Halt
  | Ast.FileOut ->
    Error.internal_error "unable to push type FileOut";
    BC.Halt
  | Ast.Bool    ->
    Error.internal_error "unable to push type bool";
    BC.Halt


let int_of_bool_str = function
  | "true" -> 1
  | _ -> 0

let bool_of_bool_str = function
  | "true" -> true
  | _ -> false

(***************************************************************************
 * Return the default value for the variable by type
 *)
let var_default = function
  | Ast.Char -> " "
  | Ast.Float -> "0.0"
  | Ast.Int -> "0"
  | Ast.String -> ""

  (* internal types *)
  | Ast.Map -> ""
  | Ast.Bool -> "0"
  | Ast.Void -> ""
  | Ast.StringArray -> ""
  | Ast.FileIn -> ""
  | Ast.FileOut -> ""

let params_to_types params =
  List.map (fun p -> match p with Ast.Variable(et,_,_) -> et) params
```

```
let rec has_return code =
  let module BC = Bytecode in
  match code with
    | [] -> false
    | [BC.Return(_)] -> true
    | hd :: tl -> has_return tl


(****************************************************************************
 * Check if the variable has been defined in the local scope
 *)
let is_already_local env name =
  try
    ignore(Symbol.find_local_var env name);
    Error.report ("'" ^ name  ^ "' already defined in local scope");
    true
  with Not_found -> false

(****************************************************************************
 * Add a variable to the symbol table and return the instructions to
 * store the default value for the variable
 *)
let add_variable_default env typ name =
  let module BC = Bytecode in
  if is_already_local env name then
    [BC.Halt]
  else
    let value = var_default typ in
    let var = Symbol.add_var env name typ in
    [push_var_instr var.Symbol.vtype value; BC.store_of name var]

(****************************************************************************
 * Add the variable to the local symbol table and return the instruction
 * to store the value on the stack
 *
 *  IMPORTANT: It is expected that the value for the variable is on
 *  top of the stack.
 *)
let add_variable env typ name =
  let module BC = Bytecode in
  if env.Symbol.scope > 0 && (is_already_local env name) then
    [BC.Halt]
  else
    let var = Symbol.add_var env name typ in
    [BC.store_of name var]

(****************************************************************************
 *  Convert the AST (abstract syntax tree) into a stack based
 *  IR (intermediate representation)
 *)
let translate program =
  let module BC = Bytecode in
  let env = Symbol.create_initial_env () in

  (*
   * Converts the expression into a list of ir codes, and returns the ir code
   * list with the type of the expression
   *
   * Return: ([Bytecode.bstmt], Ast.expr_type)
     *)
  let rec translate_expr lenv = function
    | Ast.Literal_Int(v)    -> ([BC.Pushi(v)], Ast.Int)
    | Ast.Literal_Float(v)  -> ([BC.Pushf(v)], Ast.Float)
    | Ast.Literal_Char(v)   -> ([BC.Pushc(String.sub v 1 1)], Ast.Char)
    | Ast.Literal_String(v) -> ([BC.Pushs(v)], Ast.String)
    | Ast.Literal_Bool(v)   -> ([BC.Pushi(string_of_int (int_of_bool_str v))],
Ast.Bool)

    | Ast.Id(s) ->
        (try
          let var = Symbol.find_var lenv s in
          ([BC.load_of s var], var.Symbol.vtype)
        with Not_found ->
          Error.report ("'" ^ s ^ "' is undefined");
          ([BC.Halt], Ast.Void))

    (* an Ast.Binop can be either a math op or a comparison *)
    | Ast.Binop(e1,op,e2) ->
        let expr1 = translate_expr lenv e1 in
        let expr2 = translate_expr lenv e2 in

        if (snd expr1) != (snd expr2) then
          let type1 = Ast.string_of_type (snd expr1) in
          let type2 = Ast.string_of_type (snd expr2) in
          let () = Error.report ("cannot convert type '" ^ type1 ^ "' to '" ^
type2 ^ "'") in
          ([BC.Halt], Ast.Void)
        else
          let code = match op with
            | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div -> BC.Binop(op, (snd expr1))
            | _ ->
                let () = Error.internal_error "attempt to evaluate Cmp in
generation translate_expr" in
                BC.Cmp(op, Ast.Void, 0) in

          ((fst expr1) @ (fst expr2) @ [code], (snd expr1))

    | Ast.Noexpr -> ([], Ast.Void)
    | Ast.Call(name, params) -> begin
        try
          let func = Symbol.find_function name in
          let params' = List.map (translate_expr lenv) params in
          let params_code = List.map fst params' in
          let params_type = List.map snd params' in
          let param_code = List.fold_left (@) [] params_code in

          let param_match = try List.for_all2 (=) params_type func.Symbol.args
            with Invalid_argument(_) -> begin
              false
            end in

          if not param_match then
            begin
              Error.report ("number or type of parameters in call to '" ^ name ^
"' don't match");
```

```
              ("BC.Halt], Ast.Void)
            end
          else
            (BC.Comment("setting up call to " ^ name) ::
              param_code @ [BC.Call(name, true)], func.Symbol.rtn)
        with Not_found -> begin
          Error.report ("cannot find function '" ^ name ^ "'");
          ([BC.Halt], Ast.Void)
        end
      end

    (*| Ast.Assign(e1,Ast.Assign(e2,e4)) -> *)
    | Ast.Assign(e1,e2) -> begin
        match e1 with
          | Ast.Id(s) ->
            begin
              try
                let var = Symbol.find_var lenv s in
                let expr2 = translate_expr lenv e2 in

                if var.Symbol.vtype != (snd expr2) then
                  let type1 = Ast.string_of_type var.Symbol.vtype in
                  let type2 = Ast.string_of_type (snd expr2) in
                  begin
                    Error.report ("cannot convert type '" ^ type1 ^ "' to '" ^
type2 ^ "'");
                    ([BC.Halt], Ast.Void)
                  end
                else
                  (BC.Comment("setting variable " ^ s) ::
                    (fst expr2) @ [BC.store_of s var], var.Symbol.vtype)
              with Not_found ->
                Error.report ("'" ^ s ^ "' is undefined");
                ([BC.Halt], Ast.Void)
            end
          | Ast.MapAccess(id,col) ->
            begin
              try
                let var = Symbol.find_var lenv id in
                let expr2 = translate_expr lenv e2 in
                let colexpr = translate_expr lenv col in

                if var.Symbol.vtype != Ast.Map then
                  let typ = Ast.string_of_type var.Symbol.vtype in
                  begin
                    Error.report ("cannot convert type '" ^ typ ^ "' to
'record'");
                    ([BC.Halt], Ast.Void)
                  end
                else if (snd colexpr) != Ast.String then
                  let typ = Ast.string_of_type (snd colexpr) in
                  begin
                    Error.report ("cannot convert type '" ^ typ ^ "' to 'string'
for record key");
                    ([BC.Halt], Ast.Void)
                  end
                else if (snd expr2) != Ast.String then
                  let typ = Ast.string_of_type (snd expr2) in
                  begiu
                    Error.report ("cannot convert type '" ^ typ ^ "' to 'string'
for record value");
                    ([BC.Halt], Ast.Void)
                  end
                else
                  (BC.Comment("setting map " ^ id) ::
                    [BC.load_of id var] @ (fst colexpr) @ (fst expr2) @

[BC.Call("java/util/Map/put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object
;", false)], Ast.Map)
              with Not_found ->
                Error.report ("'" ^ id ^ "' is undefined");
                ([BC.Halt], Ast.Void)
            end
          | Ast.Literal_Int(_) | Ast.Literal_Float(_)| Ast.Literal_String(_) |
            Ast.Literal_Char(_) | Ast.Literal_Bool(_) ->
            begin
              Error.report "cannot assign a value to a literal";
              ([BC.Halt], Ast.Void)
            end
          | Ast.Binop(_,_,_) ->
            begin
              Error.report "cannot assign a value to the result of an operation";
              ([BC.Halt], Ast.Void)
            end
          | Ast.Call(_,_) ->
            begin
              Error.report "cannot assign a value to the result of a function
call";
              ([BC.Halt], Ast.Void)
            end
          | Ast.Noexpr ->
            begin
              Error.report "assigning value to nothing ?";
              ([BC.Halt], Ast.Void)
            end
          | Ast.Assign(_,_) ->
            begin
              Error.report "assignment chaining error ??";
              ([BC.Halt], Ast.Void)
            end
      end

    | Ast.MapAccess(id,col) -> begin
        try
          let var = Symbol.find_var lenv id in
          let colexpr = translate_expr lenv col in

          if var.Symbol.vtype != Ast.Map then
            let typ = Ast.string_of_type var.Symbol.vtype in
            begin
              Error.report ("cannot convert type '" ^ typ ^ "' to 'record'");
              ([BC.Halt], Ast.Void)
            end
          else if (snd colexpr) != Ast.String then
            let typ = Ast.string_of_type (snd colexpr) in
            begin
```

```
              Error.report ("cannot convert type '" ^ typ ^ "' to 'string' for
record key");
              ([BC.Halt], Ast.Void)
            end
          else
            (BC.Comment("getting map " ^ id) ::
              [BC.load_of id var] @ (fst colexpr) @
              [BC.Call("java/util/Map/get(Ljava/lang/Object;)Ljava/lang/Object;",
false)], Ast.Map)
        with Not_found ->
          Error.report ("'" ^ id ^ "' is undefined");
          ([BC.Halt], Ast.Void)
      end
  in

  let reverse_cmp_op = function
    | Ast.Equal -> Ast.Neq
    | Ast.Neq -> Ast.Equal
    | Ast.Less -> Ast.Geq
    | Ast.Geq -> Ast.Less
    | Ast.Leq -> Ast.Greater
    | Ast.Greater -> Ast.Leq
    | _ -> begin
        Error.internal_error "attempt to reverse arithmetic op";
        Ast.Equal
      end
  in


  (**************************************************************************
   * Generate the code to evaluate the left and right expressions for
   * the comparision, and then do the compare op with a jump supplied
   * for the false value
   *)
  let translate_cmp lenv label = function
    | Ast.Binop(e1,op,e2) ->
        let expr1 = translate_expr lenv e1 in
        let expr2 = translate_expr lenv e2 in
        if (snd expr1) != (snd expr2) then
          let type1 = Ast.string_of_type (snd expr1) in
          let type2 = Ast.string_of_type (snd expr2) in
          let () = Error.report ("cannot convert type '" ^ type1 ^ "' to '" ^
type2 ^ "'") in
          ([BC.Halt], Ast.Void)
        else
          (match op with
            (* error message for invalid bin op not generated here *)
            | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div ->
              ([BC.Halt], Ast.Void)
            | _ ->
              let code = (fst expr1) @ (fst expr2) @ [BC.Cmp(reverse_cmp_op op,
(snd expr1), label)] in
              (code, Ast.Bool))

    | Ast.Literal_Bool(value) ->
        if bool_of_bool_str value then
          ([BC.Jump(label)], Ast.Bool)
        else
```

```
          ("^, Ast.Void)

    | _ as expr ->
      let () = Error.internal_error ("attempting to translate binop when expr is a
'" ^
        (Ast.string_of_expr expr) ^ "'") in
      ([BC.Halt], Ast.Void)
  in

  let translate_cmp_rev lenv label = function
    | Ast.Binop(e1,op,e2) ->
        let expr1 = translate_expr lenv e1 in
        let expr2 = translate_expr lenv e2 in
        if (snd expr1) != (snd expr2) then
          let type1 = Ast.string_of_type (snd expr1) in
          let type2 = Ast.string_of_type (snd expr2) in
          let () = Error.report ("cannot convert type '" ^ type1 ^ "' to '" ^
type2 ^ "'") in
          ([BC.Halt], Ast.Void)
        else
          (match op with
            (* error message for invalid bin op not generated here *)
            | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div ->
              ([BC.Halt], Ast.Void)
            | _ ->
              let code = (fst expr1) @ (fst expr2) @ [BC.Cmp(op, (snd expr1),
label)] in
              (code, Ast.Bool))

    | _ as expr ->
      let () = Error.internal_error ("attempting to translate binop when expr is a
'" ^
        (Ast.string_of_expr expr) ^ "'") in
      ([BC.Halt], Ast.Void)
  in

  let rec translate_stmt lenv = function
    | Ast.For(e1, e2, e3, s) ->
        let label1 = next_label () in
        let label2 = next_label () in
        let label3 = next_label () in
        let expr1 = translate_expr lenv e1 in
        let expr2 = translate_cmp lenv label1 e2 in
        let expr3 = translate_expr lenv e3 in
        let nenv = Symbol.push_env lenv label3 lenv.Symbol.address
lenv.Symbol.rtype in
        let stmts = translate_stmt nenv s in
        let () = Symbol.pop_env nenv in

        if (snd expr2) != Ast.Bool then
          begin
            Error.report " test expresion in 'while' statement must evaluate to a
'boolean'";
            [BC.Halt]
          end
        else
          [BC.Comment("begin while loop")] @ (fst expr1) @
[BC.Jump(label2);BC.Label(label1)] @ stmts @
```

```
            (fst expr3) @ [BC.Label(label2)] @ (fst expr2) @ [BC.Label(label3);
BC.Comment("end while loop")]

    | Ast.While(e, s) ->
        let label1 = next_label () in
        let label2 = next_label () in
        let label3 = next_label () in
        let expr = translate_cmp lenv label1 e in
        let nenv = Symbol.push_env lenv label3 lenv.Symbol.address
lenv.Symbol.rtype in
        let stmts = translate_stmt nenv s in
        let () = Symbol.pop_env nenv in

        if (snd expr) != Ast.Bool then
          begin
            Error.report "expression for 'while' statement must evaluate to a
'boolean'";
            [BC.Halt]
          end
        else
          [BC.Comment("begin while loop");
           BC.Jump(label2);
           BC.Label(label1)]
          @ stmts @
          [BC.Label(label2)]  @ (fst expr) @ [BC.Label(label3); BC.Comment("end
while loop")]

    | Ast.If(e, s1, s2) ->
        let label1 = next_label () in
        let label2 = next_label () in
        let expr = translate_cmp lenv label1 e in
        let nenv = Symbol.push_env lenv lenv.Symbol.label lenv.Symbol.address
lenv.Symbol.rtype in
        let stmts1 = translate_stmt nenv s1 in
        let () = Symbol.pop_env nenv in
        let nenv = Symbol.push_env lenv lenv.Symbol.label lenv.Symbol.address
lenv.Symbol.rtype in
        let stmts2 = translate_stmt nenv s2 in
        let () = Symbol.pop_env nenv in

        if (snd expr) != Ast.Bool then
          let () = Error.report "expression for 'if' statement must evaluate to a
'boolean'" in
          [BC.Halt]
        else
          (fst expr) @ stmts1 @
            [BC.Jump(label2); BC.Label(label1)] @ stmts2 @ [BC.Label(label2)]

    | Ast.Function(name, params, rtn_type, block) ->
        let lenv = Symbol.push_env lenv 0 0 rtn_type in
        let () =
          ignore(List.map (fun elt ->
            match elt with
              | Ast.Variable(t,n,e) -> begin
                if is_already_local lenv n then
                  Error.report ("function parameter '" ^ n ^ "' already defined in
'" ^ name ^ "'")
                else
                iguore(Symbol.add_local_var lenv n t)
              end
          ) params)
        in
        let cnt = List.length params in
        let block_code = translate_stmt lenv block in
        let stack_limit = snd (calc_stack_limit block_code) + 2 in
        let local_limit = max (calc_local_limit block_code) cnt in

        ignore(Symbol.pop_env lenv);

        BC.Bfunc(name, (params_to_types params), rtn_type, stack_limit,
local_limit) :: block_code @

        if has_return block_code then
          [BC.Efunc(name)]
        else
          [(BC.push_of rtn_type); BC.Return(rtn_type); BC.Efunc(name)]

    | Ast.FilterFunc(name, stmts) ->
      let func = Ast.Function(name, [Ast.Variable(Ast.Map, "rec", Ast.Noexpr)],
Ast.Bool, stmts) in
      translate_stmt lenv func
    | Ast.Declare(var) -> begin
        try
          trans_declare lenv var
        with Symbol.Duplicate_id(id) ->
          Error.report ("a variable named " ^ id ^ " already exists");
          [BC.Halt]
      end
    | Ast.Expr(expr) ->
      (match expr with
      | Ast.Assign(_, _) | Ast.Call(_, _) | Ast.MapAccess(_, _) ->
        fst (translate_expr lenv expr)
      | _ ->
          let () = Error.report ("'" ^ (Ast.string_of_expr expr) ^ "' is not a
valid statement") in
          [BC.Halt])

    | Ast.Block(stmt_lst) -> trans_block lenv stmt_lst
    | Ast.Print(expr) -> trans_print lenv expr

    | Ast.Return(expr) ->
      let ecode = translate_expr lenv expr in
      if snd ecode != lenv.Symbol.rtype then
        begin
          let type1 = Ast.string_of_type (snd ecode) in
          let type2 = Ast.string_of_type lenv.Symbol.rtype in
          Error.report ("return type of '" ^ type1 ^ "' does not match function
return type of '" ^ type2 ^ "'");
          [BC.Halt]
        end
      else
        (fst ecode) @ [BC.Return(snd ecode)]

    | Ast.File(name, path, opt) ->
      begin
        try
```

```
          iguore(Symbol.find_var lenv name);
          Error.stmt_error ("'" ^ name ^ "' is already defined");
          [BC.Halt]
        with Not_found -> begin
          let typ =
            List.fold_left (fun acc elt -> if elt = Ast.Output then Ast.FileOut
else acc) Ast.FileIn opt in
          let sep =
            List.fold_left (fun acc elt -> match elt with Ast.Sep(s) -> s | _ ->
acc) "\",\"" opt in
          let var = Symbol.add_var lenv name typ in
          let call =
            if typ == Ast.FileIn then

[BC.Call("Rtl/create_in(Ljava/lang/String;Ljava/lang/String;)LRtlInFile;", false)]
            else

[BC.Call("Rtl/create_out(Ljava/lang/String;Ljava/lang/String;)LRtlOutFile;",
false)]
          in
          [BC.Pushs(path);BC.Pushs(sep)] @ call @ [BC.store_of name var]
        end
      end

    | Ast.Process(fromf, steps, tof) ->
        begin
        try let f_var = Symbol.find_var lenv fromf in
        try let t_var = Symbol.find_var lenv tof in
          begin
            let label1 = next_label () in
            let label2 = next_label () in
            let label3 = next_label () in
            let map_name = "__map_" ^ (string_of_int (next_label ())) in
            let (map_var : Symbol.sym_var) = (Symbol.add_local_var lenv map_name
Ast.Map) in
            let nenv = Symbol.push_env lenv label1 lenv.Symbol.address
lenv.Symbol.rtype in
            let () = ignore(nenv.Symbol.rec_address <- map_var.Symbol.address) in
            let step_code = List.map (translate_process_step nenv map_name
map_var) steps in
            let step_code' = List.fold_left (fun acc elt -> elt @ acc) []
(List.rev step_code) in
            let () = Symbol.pop_env nenv in

            [
             BC.Label(label1);
             BC.load_of fromf f_var;
             BC.CallV("RtlInFile/read()Ljava/util/Map;", false);
             BC.Dup;
             BC.store_of map_name map_var;
             BC.Cmp(Ast.NotNull, Ast.FileIn, label2);
             BC.Jump(label3);
             BC.Label(label2)
            ]

            @ step_code' @

            [
             BC.load_of fromf f_var;
             BC.CallV("RtlInFile/getHeader()[Ljava/util/String;",false);
             BC.load_of map_name map_var;
             BC.load_of tof t_var;

BC.CallV("RtlOutFile/write([Ljava/lang/String;Ljava/util/Map;)V;",false);
             BC.Jump(label1);
             BC.Label(label3);
            ]

          end
        with Not_found ->
          (Error.stmt_error ("process from file '" ^ fromf ^ "' is not defined");
          [BC.Halt])
        with Not_found ->
          (Error.stmt_error ("process from file '" ^ tof ^ "' is not defined");
          [BC.Halt])
        end


    | Ast.Break ->
        if lenv.Symbol.label = -1 then
          begin
            Error.stmt_error "break statement used in non-loop  context";
            [BC.Halt]
          end
        else
          [BC.Jump(lenv.Symbol.label)]

  and translate_process_step lenv map_name map_var = function
    (* Call the function for filter processing *)
    | Ast.Check_Function(id) ->
        if lenv.Symbol.rec_address = -1 then
          begin
            Error.internal_error "Check_Function statement has no rec label";
            [BC.Halt]
          end
        else
          begin
            [BC.Load(lenv.Symbol.rec_address, Ast.Map);
             BC.Call(id, true);
             BC.Cmp(Ast.Equal, Ast.Bool, lenv.Symbol.label)]
          end


  and trans_block lenv stmt_lst =
    let local_env = Symbol.push_env lenv lenv.Symbol.label lenv.Symbol.address
lenv.Symbol.rtype in
    let code = List.map (translate_stmt local_env) stmt_lst in
    let () = Symbol.pop_env local_env in
    List.fold_left (fun acc elt -> elt @ acc) [] (List.rev code)

  and trans_declare lenv = function
    | Ast.Variable(typ, name, Ast.Noexpr) -> add_variable_default lenv typ name
    | Ast.Variable(typ, name, e1) ->
        let expr = translate_expr lenv e1 in
        if typ != (snd expr) then
          let type1 = Ast.string_of_type typ in
          let type2 = Ast.string_of_type (snd expr) in
          Error.stmt_error ("cannot convert type '" ^ type1 ^ "' to '" ^ type2 ^
"'");
          [BC.Halt]
        else
          (fst expr) @ (add_variable lenv typ name)

  and trans_print lenv expr =
    let module BC = Bytecode in
    (* create formatted string to pass to print *)
    let (expr_code, typ) = translate_expr lenv expr in
    let setup = match typ with
      | Ast.String -> expr_code
      | Ast.Int -> expr_code @ [BC.Call("Rtl/int_to_string(I)Ljava/lang/String;",
false)]
      | Ast.Char -> expr_code @
[BC.Call("Rtl/char_to_string(C)Ljava/lang/String;", false)]
      | Ast.Float -> expr_code @
[BC.Call("Rtl/float_to_string(F)Ljava/lang/String;", false)]
      | _ -> begin
        Error.report ("cannot print type '" ^ (Ast.string_of_type typ) ^ "'");
        [BC.Halt]
      end in

     setup @ [BC.Call("Rtl/print(Ljava/lang/String;)V", false)]
  in

  let list_globals () =
    Hashtbl.fold (fun k v acc -> (k,v) :: acc) Symbol.global_vars []
  in

  let extract_funcs acc = function
    | Ast.Function(n,p,r,_) as fb -> begin
        try
          Symbol.add_func n (Ast.var_list_to_type_list p) r
        with Symbol.Duplicate_id(id) ->
          Error.report ("a function named " ^ id ^ " already exists")
        end;
        fb :: acc
    | Ast.FilterFunc(n,_) as ff -> begin
        try
          Symbol.add_func n [Ast.Map] Ast.Bool
        with Symbol.Duplicate_id(id) ->
          Error.report ("a function named " ^ id ^ " already exists")
        end;
        ff :: acc
    | _ -> acc
  in


  (*let prog_stmt = List.map dump_program program in*)
  let func_blocks = List.rev (List.fold_left extract_funcs [] program) in

  (*
   * Remove all functions statements
   *)
  let main_stmt_list = List.fold_left
    (fun acc elt ->
      match elt with
        | Ast.Function(_,_,_,_) | Ast.FilterFunc(_,_) -> acc
        | _ as stmt -> stmt :: acc) [] program in

  let main_arg = Ast.Variable(Ast.StringArray,"args",Ast.Noexpr) in
  let main_func = [Ast.Function("main", [main_arg],Ast.Void,Ast.Block(List.rev
main_stmt_list))] in

  (*  The environment tuple = use globals * symbol tables *)
  let main_code = List.map (translate_stmt env) main_func in
  let func_code = List.map (translate_stmt (Symbol.push_env env 0 0
env.Symbol.rtype)) func_blocks in
  let globals = list_globals () in

  let ir = List.fold_left (fun acc elt -> elt @ acc) [] (List.rev main_code @
func_code) in

(*
  let dump = String.concat "\n" prog_stmt in
  print_endline dump;
*)

  (ir, globals)
```

```
(*************************************************************************
 *
 * File: opt_ir.ml
 *
 * Purpose: perform any non-processor specific optimizations.
 *
 *)

(*
 * This is one of those, if I have time :) items
 * Currently just a pass through
 *)

let optimize (ir, globals) =
  (ir, globals)
```

```
(****************************************************************************
 *
 * File: ovm.ml
 *
 * Purpose: convert the bytecode into jvm instructions that can be compiled
 * into a java class using jasmin.
 *
 * Also, the created class file is add to the runtime class files and a
 * single jar file is produced that can be run.
 *
 *)

(* we use strings for constants so that any transformations are performed by
 * the jasmin assembler, this is mainly a concern for float constants
 *)

type jcode =
  | Dup
  | Pop
  | Swap
  | Ldc_s of string
  | Ldc_i of string
  | Ldc_f of string
  | Bipush of string

  | Iload of int
  | Istore of int
  | Isub
  | Iadd
  | Idiv
  | Imul

  | Fload of int
  | Fstore of int
  | Fsub
  | Fadd
  | Fdiv
  | Fmul
  | Fcmp

  | Astore of int
  | Aload of int

  | GetStatic of string * string
  | PutStatic of string * string

  | Invokespecial of string
  | Invokevirtual of string
  | Invokestatic of string

  | Anewarray of string
  | Aastore

  | Ifeq of int
  | Ifne of int
  | Ifgt of int
  | Ifge of int
  | Iflt of int
  | Ifle of int

  | If_icmpeq of int
  | If_icmpne of int
  | If_icmplt of int
  | If_icmple of int
  | If_icmpgt of int
  | If_icmpge of int
  | Ifnonnull of int

  | Bmethod of string * int * int
  | Emethod of string
  | Returni
  | Returnf
  | Returna
  | Return

  | Comment of string
  | Label of int
  | Goto of int

  | I2c
  | I2b
  | F2i
  | Nop

let to_jvm_type = function
  | Ast.Char    -> "C"
  | Ast.Float   -> "F"
  | Ast.Int     -> "I"
  | Ast.String  -> "Ljava/lang/String;"
  | Ast.Bool    -> "I"
  | Ast.Map     -> "Ljava/util/Map;"
  | Ast.Void    -> "V"
  | Ast.StringArray -> "[Ljava/lang/String;"
  | Ast.FileIn  -> "LRtl/FileIn;"
  | Ast.FileOut -> "LRtl/FileOut;"

let to_param_str = function
  | Ast.Void -> "" (* void parameter is an empty () *)
  | _ as typ -> to_jvm_type typ

let to_bmethod name params rtn stack locals =
  let param_names = String.concat "" (List.map to_param_str params) in
  let signature = Printf.sprintf ".method public static %s(%s)%s" name param_names
(to_jvm_type rtn) in
  Bmethod(signature, stack, locals)

(***************************************************************************
 * Make the proper jasmine function signature to make a call
 *)
let make_signature name params rtn =
  let param_names = String.concat "" (List.map to_param_str params) in
  Printf.sprintf "%s(%s)%s" name param_names (to_jvm_type rtn)

let to_return_type = function
  | Ast.Bool | Ast.Int | Ast.Char -> Returni
  | Ast.Float -> Returnf
  | Ast.Map | Ast.String | Ast.StringArray | Ast.FileIn | Ast.FileOut -> Returna
  | Ast.Void -> Return


let to_jcode ir_code =
  let module BC = Bytecode in
  (*
  let to_return_code = function
    | Ast.Bool | Ast.Int | Ast.Char -> Returni
    | Ast.Float -> Returnf
    | Ast.Void -> Return
    | _ -> Returna
  in
  *)

  let make_math_op op icode fcode = function
    | Ast.Int -> icode
    | Ast.Float -> fcode
    | _ as typ ->
      begin
        Error.internal_error ("cannot perform arithmetic on objects of type '" ^
(Ast.string_of_type typ) ^ "' with '" ^ op ^ "'");
        [Nop]
      end
  in

  let to_op_code typ = function
    | Ast.Add ->
        make_math_op "+" [Iadd] [Fadd] typ
    | Ast.Sub ->
        make_math_op "-" [Isub] [Fsub] typ
    | Ast.Mult ->
        make_math_op "*" [Imul] [Fmul] typ
    | Ast.Div ->
        make_math_op "/" [Idiv] [Fdiv] typ

    (* not supported yet *)
    | Ast.LAnd ->
        make_math_op "&&" [Nop] [Nop] typ
    | Ast.LOr ->
        make_math_op "||" [Nop] [Nop] typ

    (* sanity check *)
    | Ast.Equal | Ast.Neq | Ast.Less | Ast.Leq | Ast.Greater | Ast.Geq |
Ast.NotNull ->
      begin
        Error.internal_error "attempt to match comparison operators with
mathemcatical operators";
        [Nop]
      end
  in

  let make_cmp_op op icode fcode scode = function
    | Ast.Int | Ast.Char | Ast.Bool -> icode
    | Ast.Float -> fcode
    | Ast.String as typ ->
      if List.length scode > 0 then
        scode
      else
        begin
          Error.internal_error ("cannot compare objects of type '" ^
(Ast.string_of_type typ) ^ "' with '" ^ op ^ "'");
          [Nop]
        end
    | _ as typ ->
      begin
        Error.internal_error ("cannot compare objects of type '" ^
(Ast.string_of_type typ) ^ "' with '" ^ op ^ "'");
        [Nop]
      end
  in

  (* lbl = label number when comparision is false *)
  let to_cmp_code typ lbl = function
    | Ast.Equal ->
        make_cmp_op "==" [If_icmpne(lbl)] [Fcmp; Ifne(lbl)]
        [Invokestatic("Rtl/equals(Ljava/lang/String;Ljava/lang/String;)Z");
Ifne(lbl)] typ
    | Ast.Neq ->
        make_cmp_op "!=" [If_icmpeq(lbl)] [Fcmp; Ifeq(lbl)]
        [Invokestatic("Rtl/equals(Ljava/lang/String;Ljava/lang/String;)Z");
Ifeq(lbl)] typ
    | Ast.Less ->
        make_cmp_op "<" [If_icmpge(lbl)] [Fcmp; Ifge(lbl)] [] typ
    | Ast.Leq ->
        make_cmp_op "<=" [If_icmpgt(lbl)] [Fcmp; Ifgt(lbl)] [] typ
    | Ast.Greater ->
        make_cmp_op "<=" [If_icmple(lbl)] [Fcmp; Ifle(lbl)] [] typ
    | Ast.Geq ->
        make_cmp_op "<=" [If_icmplt(lbl)] [Fcmp; Iflt(lbl)] [] typ
    | Ast.NotNull ->
        [Ifnonnull(lbl)]
    | Ast.Add | Ast.Sub | Ast.Mult | Ast.Div | Ast.LAnd | Ast.LOr ->
      begin
        Error.internal_error "attempt to match comparison operators with
mathemcatical operators";
        [Nop]
      end
  in

  let to_load i = function
    | Ast.Int -> [Iload(i)]
    | Ast.Char -> [Iload(i)]
    | Ast.Float -> [Fload(i)]
    | Ast.String -> [Aload(i)]
    | Ast.StringArray -> [Aload(i)]
    | Ast.FileIn -> [Aload(i)]
    | Ast.FileOut -> [Aload(i)]
    | Ast.Map -> [Aload(i)]
    | Ast.Void | Ast.Bool -> [Nop]
  in

  let to_store i = function
    | Ast.Int -> [Istore(i)]
    | Ast.Char -> [Istore(i)]
    | Ast.Float -> [Fstore(i)]
```

```
      | Ast.String -> [Astore(i)]
      | Ast.StringArray -> [Astore(i)]
      | Ast.FileIn -> [Astore(i)]
      | Ast.FileOut -> [Astore(i)]
      | Ast.Map -> [Astore(i)]
      | Ast.Void | Ast.Bool -> [Nop]
    in

    let to_get_field name = function
      | Ast.Int -> [GetStatic(name,"I")]
      | Ast.Char -> [GetStatic(name,"C")]
      | Ast.Float -> [GetStatic(name,"F")]
      | Ast.String -> [GetStatic(name,"Ljava/lang/String;")]
      | Ast.StringArray -> [GetStatic(name,"[Ljava/lang/String;")]
      | Ast.FileIn -> [GetStatic(name,"LRtlFileIn;")]
      | Ast.FileOut -> [GetStatic(name,"LRtlFileOut;")]
      | Ast.Map -> [GetStatic(name,"Ljava/util/Map;")]
      | Ast.Void | Ast.Bool -> [Nop]
    in

    let to_put_field name = function
      | Ast.Int -> [PutStatic(name,"I")]
      | Ast.Char -> [PutStatic(name,"C")]
      | Ast.Float -> [PutStatic(name,"F")]
      | Ast.String -> [PutStatic(name,"Ljava/lang/String;")]
      | Ast.StringArray -> [PutStatic(name,"[Ljava/lang/String;")]
      | Ast.FileIn -> [PutStatic(name,"LRtlFileIn;")]
      | Ast.FileOut -> [PutStatic(name,"LRtlFileOut;")]
      | Ast.Map -> [PutStatic(name,"Ljava/util/Map;")]
      | Ast.Void | Ast.Bool -> [Nop]
    in

    match ir_code with
      | BC.Dup -> [Dup]
      | BC.Pop -> [Pop]
      | BC.Swap -> [Swap]
      | BC.Pushb("true") -> [Bipush("1")]
      | BC.Pushb(_) -> [Bipush("0")]
      | BC.Pushc(s) -> [Bipush(string_of_int (int_of_char (String.get s 0)))]
      | BC.Pushf(s) -> [Ldc_f(s)]
      | BC.Pushs(s) -> [Ldc_s(s)]
      | BC.Pushi(s) -> [Ldc_i(s)]
      | BC.Binop(op, typ) -> to_op_code typ op
      | BC.Cmp(op, typ, lbl) -> List.rev(to_cmp_code typ lbl op)
      | BC.Load(i,typ) -> to_load i typ
      | BC.Store(i,typ) -> to_store i typ
      | BC.GetGlobal(n,typ) -> to_get_field n typ
      | BC.PutGlobal(n,typ) -> to_put_field n typ
      | BC.Call(s,_)  -> [Invokestatic(s)]
      | BC.CallV(s,_)  -> [Invokevirtual(s)]
      | BC.Bfunc(n,p,r,s,l) -> [to_bmethod n p r s l]
      | BC.Efunc(s) -> [Emethod(s)]
      | BC.Return(typ) -> [to_return_type typ]
      | BC.Beq(i) -> [Ifeq(i)]
      | BC.Bne(i) -> [Ifne(i)]
      | BC.Jump(i) ->    [Goto(i)]
      | BC.Label(i) -> [Label(i)]
      | BC.Halt   ->     [Nop]
      | BC.Nop   ->     [Nop]
      | BC.Comment(s) -> [Comment(s)]

  let to_jvm_code (ir, globals) =
    let fixup_calls ir =
      let module BC = Bytecode in
      let fix_call = function
        | BC.Call(s,true) -> begin
            let var = Symbol.find_function s in
            BC.Call(make_signature var.Symbol.fid var.Symbol.args var.Symbol.rtn,
  false)
          end
        | BC.CallV(s,true) -> begin
            let var = Symbol.find_function s in
            BC.CallV(make_signature var.Symbol.fid var.Symbol.args var.Symbol.rtn,
  false)
          end
        | _ as instr -> instr
      in
      List.map fix_call ir
    in
    let rcode = List.map (fun elt -> to_jcode elt) (fixup_calls ir) in
    let code = List.rev rcode in

    (List.rev (List.fold_left (fun elt acc -> elt @ acc) [] code), globals)

  (*
   * Empty strings must be represented by ""
   *)
  let fix_empty_str = function
    | "" -> "\"\""
    | _ as s -> s


  (*
   * Convert the IR bytecode into equvialent jvm byte code, a single
   * byte code might require more than one jvm byte codes
   *)
  let jcode_to_string cname = function
    | Dup -> "dup"
    | Pop -> "pop"
    | Swap -> "swap"
    | Ldc_s(s) -> "ldc " ^ fix_empty_str s
    | Ldc_i(s) -> "ldc " ^ fix_empty_str s
    | Ldc_f(s) -> "ldc " ^ fix_empty_str s
    | Bipush(s) -> "bipush " ^ fix_empty_str s
    | Iload(i) -> "iload " ^ string_of_int i
    | Istore(i) -> "istore " ^ string_of_int i
    | Isub -> "isub"
    | Iadd -> "iadd"
    | Idiv -> "idiv"
    | Imul -> "imul"
    | Fload(i) -> "fload " ^ string_of_int i
    | Fstore(i) -> "fstore " ^ string_of_int i
    | Fsub -> "fsub"
    | Fadd -> "fadd"
    | Fdiv -> "fdiv"
    | Fmul -> "fmul"
    | Fcmp -> "fcmp"
    | Astore(i) -> "astore " ^ string_of_int i
    | Aload(i) -> "aload " ^ string_of_int i
    | GetStatic(n,t) -> "getstatic " ^ cname ^ "/" ^ n ^ " " ^ t
    | PutStatic(n,t) -> "putstatic " ^ cname ^ "/" ^ n ^ " " ^ t
    | Invokespecial(s) -> "invokespecial " ^ s
    | Invokevirtual(s) -> "invokevirtual " ^ s
    | Invokestatic(s) -> "invokestatic " ^ s
    | Anewarray(s) -> "anewarray"
    | Aastore -> "aastore"
    | Ifeq(i) -> "ifeq Label" ^ string_of_int i
    | Ifne(i) -> "ifne Label" ^ string_of_int i
    | Ifgt(i) -> "ifgt Label" ^ string_of_int i
    | Ifge(i) -> "ifge Label" ^ string_of_int i
    | Iflt(i) -> "iflt Label" ^ string_of_int i
    | Ifle(i) -> "ifle Label" ^ string_of_int i
    | If_icmpeq(i) -> "if_icmpeq Label" ^ string_of_int i
    | If_icmpne(i) -> "if_icmpne Label" ^ string_of_int i
    | If_icmplt(i) -> "if_icmplt Label" ^ string_of_int i
    | If_icmple(i) -> "if_icmple Label" ^ string_of_int i
    | If_icmpgt(i) -> "if_icmpgt Label" ^ string_of_int i
    | If_icmpge(i) -> "if_icmpge Label" ^ string_of_int i
    | Ifnonnull(i) -> "ifnonnull Label" ^ string_of_int i
    | Bmethod(s,stack,locals) -> s ^ (Printf.sprintf "\n\t.limit stack %d\n\t.limit
  locals %d" stack locals)
    | Emethod(s) -> ".end method\n"
    | Returni -> "ireturn"
    | Returnf -> "freturn"
    | Returna -> "areturn"
    | Return -> "return"
    | Comment(s) -> "; " ^ s
    | Label(i) -> "Label" ^ string_of_int i ^ ":"
    | Goto(i) -> "goto Label" ^ string_of_int i
    | I2c -> "i2c"
    | I2b -> "i2b"
    | F2i -> "f2i"
    | Nop -> ""



  (*************************************************************************
   *  Write the code to a file
   *)
  let assemble filename (code, globals) =
    let bname = Filename.basename filename in
    let name = Filename.chop_extension bname in
    let gen_header name =
      let fmt : ('a, 'b, 'c ) format =
        ".source %s.j\n" ^^
        ".class public %s\n" ^^
        ".super java/lang/Object\n\n" in
      Printf.sprintf fmt name name
    in
    let header = gen_header name in
    let to_string code =
      List.map (fun elt ->
        let prefix =
          match elt with
```

```
          | Label(_) | Bmethod(_,_,_) | Emethod(_) -> ""
          | _ -> "\t" in
      prefix ^ (jcode_to_string name elt)) code
  in
  let body = to_string code |> String.concat "\n" in
  let _init_ =
      ".method public <init>()V\n" ^
      "\taload_0\n" ^
      "\tinvokenonvirtual java/lang/Object/<init>()V\n" ^
      "\treturn\n" ^
      ".end method\n\n" in
  let generate_field (name, var) =
    Printf.sprintf ".field private static %s %s" name (to_jvm_type
var.Symbol.vtype)
  in
  let fields = String.concat "\n" (List.map generate_field globals) in

  try
    let oname = (Filename.chop_extension filename) ^ ".j" in
    let () = if Sys.file_exists oname then Sys.remove oname else () in
    let oc = open_out_gen [Open_creat; Open_text; Open_wronly] 0o640 oname in

    Printf.fprintf oc "%s%s\n\n%s\n%s\n" header fields _init_ body;

    close_out oc;
    true
  with Failure(s) ->
    Error.report s;
    false

(**************************************************************************
 * Combine the runtime files into a single jar with the program
 *)
let link filename flag =
  if flag = false then
    false
  else begin
    let path = Filename.dirname filename in
    let bname = Filename.basename filename in
    let name = Filename.chop_extension bname in
    let rtl_path = try Sys.getenv "RTL" with Not_found -> "./" in
    let rt_classes = ["Rtl.class"; "RtlException.class"; "RtlInFile.class";
"RtlOutFile.class"] in
    let rt_files = List.map (fun elt -> rtl_path ^ "/" ^ elt) rt_classes in
    let files = String.concat " " rt_files in
    let cmd = Printf.sprintf "jar cfe %s.jar %s %s.class %s" name name name files
in
    (*let () = print_endline cmd in*)
    let rcode = Sys.command cmd in

    (* clean up class file *)
    let () = Sys.remove (name ^ ".class") in

    if rcode != 0 then
      begin
        Error.report ("linking failed, error " ^ string_of_int rcode);
        Error.report ("command [" ^ cmd ^ "]");
        false
      end
    else
      begin
        (* move jar to same directory as source *)
        Sys.rename (name ^ ".jar") (path ^ "/" ^ name ^ ".jar");
        true
      end
  end


(***************************************************************************
 * Call jasmin on the assembly file
 *)
let compile filename flag =
  if flag = false then
    false
  else
    begin
      let iname = (Filename.chop_extension filename) ^ ".j" in
      let cmd = Printf.sprintf "jasmin %s" iname in
      (*let () = print_endline cmd in*)
      let rcode = Sys.command cmd in

      if rcode = 0 then
        link filename true
      else begin
        Error.report ("assembling failed, error " ^ string_of_int rcode);
        false
      end
    end
```

```
(****************************************************************************
 *
 * File: jpt_ovm.ml
 *
 * Purpose: processor specific optimization for the JVM
 *
 *)

(*
 * Currently just a pass through
 *)

let optimize jcode =
  jcode
```

```
(**********************************************************************
 *
 * File: rtl.ml
 *
 * Purpose: main application
 *
 *)

type action_type = PrintAst | PrintBC | Compile | Disassemble | Help | Ignore

let show_help () =
  let msg = "rtl {options} files\n" ^
    "    -h   --help            this message\n" ^
    "    -a   --ast             dump abstract syntax tree\n" ^
    "    -b   --bytecode        generate bytecode listing\n" ^
    "    -S   --disassemble     generate disassembly listing\n"
  in
    print_string msg;
    exit 1

let action = ref Compile
let file_count = ref 0
let has_error = ref false

let current_line lb p =
  if String.contains_from lb.Lexing.lex_buffer p.Lexing.pos_bol '\n' then
    String.sub lb.Lexing.lex_buffer p.Lexing.pos_bol
      (String.index_from lb.Lexing.lex_buffer p.Lexing.pos_bol '\n' -
p.Lexing.pos_bol)
  else
    String.sub lb.Lexing.lex_buffer p.Lexing.pos_bol
      (String.length lb.Lexing.lex_buffer - p.Lexing.pos_bol)

let create_error_caret i =
  String.make i ' ' ^ "^"


let () =
  let specs = [
    ('a', "ast",        (Getopt.set action PrintAst), None);
        ('b', "bytecode",   (Getopt.set action PrintBC), None);
    ('S', "disasemble", (Getopt.set action Disassemble), None);
    ('h', "help",       (Getopt.set action Help), None)
  ] in

  (*
   * Attempt to open a file
   *
   * Return: option(channel)
   *)
  let open_file filename =
    try
      Some(open_in filename)
    with Sys_error(msg) ->
      Printf.fprintf stderr "error: %s\n" msg;
      None
  in
```

```
(*
 * Psrse file
 *
 * Returns: AST
 *)
let parse_file filename channel =
  let empty_prog = Ast.Expr(Ast.Noexpr) :: [] in
  let lexbuf = Lexing.from_channel channel in

  lexbuf.Lexing.lex_curr_p <- {
    Lexing.pos_fname = filename;
    Lexing.pos_lnum = 1;
    Lexing.pos_bol = 0;
    Lexing.pos_cnum = 0
  };

  try
    let program = List.rev (Parser.program Scanner.token lexbuf) in
    close_in channel;
    program

  with e ->
    close_in channel;

    match e with
    | Parsing.Parse_error ->
      let p = Lexing.lexeme_start_p lexbuf in
      let cline = current_line lexbuf p in
      let bad_ch = p.Lexing.pos_cnum - p.Lexing.pos_bol in
      let bline = create_error_caret bad_ch in
        Printf.fprintf stderr "%s\n" cline;
        Printf.fprintf stderr "%s\n" bline;
        Printf.fprintf stderr "%s(%d) error: character %d: syntax error.\n"
          p.Lexing.pos_fname p.Lexing.pos_lnum (bad_ch + 1);

        action := Ignore;
        empty_prog
    | _ -> raise e
in

let set_exit flag =
  if flag then
    exit 0
  else
    exit 1
in

(*
 * Peform *action* on the AST
 *
 * Return: ()
 *)
let do_action filename program =
  file_count := !file_count + 1;
  match ! action with
      Help -> show_help ()
    | PrintAst ->
        Ast.string_of_program program |> List.iter print_endline
      | PrintBC ->
          Ir.translate program |> Bytecode.string_of_program |> List.iter
print_endline
      | Compile ->
          ignore(Ir.translate program |> Opt_ir.optimize |> Jvm.to_jvm_code |>
Opt_jvm.optimize |>
          Jvm.assemble filename |> Jvm.compile filename |> set_exit)
      | Disassemble ->
          ignore(Ir.translate program |> Jvm.to_jvm_code |> Jvm.assemble filename)
      | Ignore -> ()
  in

  let handle_file filename =
    match (open_file filename) with
        Some(channel) -> do_action filename (parse_file filename channel)
      | None -> action := Ignore; do_action filename []

  in
    Getopt.parse_cmdline specs handle_file;
    if !file_count = 0 then begin
      print_endline "error: no source files specified\n";
      show_help ()
    end else
      exit 0
```

```
import java.util.Map;

public class Rtl {
    public static RtlInFile create_in(String path, String sep) {
        try {
            return new RtlInFile(path, sep);
        } catch ( RtlException e ) {
            System.out.println(e.getMessage());
        }

        return null;
    }

    public static RtlOutFile create_out(String path, String sep) {
        try {
            return new RtlOutFile(path, sep);
        } catch ( RtlException e ) {
            System.out.println(e.getMessage());
        }

        return null;
    }

    public static String get_map_value(Map<String, String> map, String key) {
        if ( map.containsKey(key) ) {
            return map.get(key);
        } else {
            System.out.println("record does not contain a column named '" + key +
"'");
            return "";
        }
    }

    public static void set_map_value(Map<String, String> map, String key, String
value) {
        if ( map.containsKey(key) ) {
            map.put(key, value);
        } else {
            System.out.println("record does not contain a column named '" + key +
"'");
        }
    }

    // String functions
    public static String substr(String s, int pos, int len) {
        try {
            return s.substring(pos, pos + len);
        } catch (Exception e) {
            return "";
        }
    }

    public static boolean equals(String s1, String s2) {
        try {
            return s1.equals( s2 );
        } catch (Exception e) {
            return false;
        }
    }

    public static String concat(String s1, String s2) {
        try {
            return s1 + s2;
        } catch (Exception e) {
            return "";
        }
    }

    public static String str_format(String pattern, String[] args) {
        try {
            java.text.MessageFormat mf = new java.text.MessageFormat(pattern);
            return mf.format(args, new StringBuffer(), null).toString();
        } catch (Exception e) {
            return "";
        }
    }

    //  Print
    public static void print(String s) {
        try {
            System.out.println(s);
        } catch (Exception e) {}
    }

    //  int_to_string
    //  int_to_float
    //  int_to_char
    public static String int_to_string(int value) {
        try {
            return String.valueOf(value);
        } catch (Exception e) {
            return "";
        }
    }
    public static float int_to_float(int value) {
        try {
            return (float)value;
        } catch (Exception e) {
            return 0.0f;
        }
    }
    public static char int_to_char(int value) {
        try {
            return (char)value;
        } catch (Exception e) {
            return ' ';
        }
    }

    //  char_to_string
    //  char_to_int
    public static String char_to_string(char value) {
        try {
            return String.valueOf(value);
        } catch (Exception e) {
            return "";
        }
    }
    public static int char_to_int(char value) {
        try {
            return (int)value;
        } catch (Exception e) {
            return 0;
        }
    }

    //  string_to_int
    //  string_to_float
    //  string_to_char
    public static int string_to_int(String value) {
        try {
            return Integer.parseInt(value);
        } catch (Exception e) {
            return 0;
        }
    }
    public static float string_to_float(String value) {
        try {
            return Float.parseFloat(value);
        } catch (Exception e) {
            return 0.0f;
        }
    }
    public static char string_to_char(String value) {
        try {
            return value.charAt( 0 );
        } catch (Exception e) {
            return ' ';
        }
    }


    //  float_to_string
    //  float_to_int
    public static String float_to_string(float value) {
        try {
            return String.valueOf(value);
        } catch (Exception e) {
            return "";
        }
    }
    public static int float_to_int(float value) {
        try {
            return (int)value;
        } catch (Exception e) {
            return 0;
        }
    }
}
```

```
public class RtlException extends Exception {
    public RtlException() { super(); }
    public RtlException(String msg) { super(msg); }
    public RtlException(String msg, Throwable cause) { super(msg, cause); }
    public RtlException(Throwable cause) { super(cause); }
}
```

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Map;
import java.util.HashMap;

public class RtlInFile {
    private BufferedReader  d_br;
    private String[]        d_header;
    private String          d_splitBy;
    private boolean         d_headerRead = false;

    public RtlInFile(String filename, String splitBy) throws RtlException {
        try {
            d_br = new BufferedReader(new FileReader(filename));
            d_splitBy = splitBy;
        } catch ( Exception e ) {
            throw new RtlException(e.getMessage());
        }
    }

    public String[] getHeader() {
        return d_header;
    }

    //  Returns null on EOF
    public Map<String,String> read() {
        try {
            if ( ! d_headerRead ) {
                String line = d_br.readLine();
                if ( line == null ) {
                    return null;
                }

                d_header = line.split(d_splitBy);
                d_headerRead = true;
            }

            String line = d_br.readLine();
            if ( line == null ) {
                return null;
            }

            String[] row = line.split(d_splitBy);

            Map<String,String> rec = new HashMap<String, String>();
            int count = Math.min(d_header.length, row.length);

            for ( int i = 0; i < count; ++i ) {
                rec.put(d_header[i], row[i]);
            }

            return rec;
        } catch ( Exception e ) {
            System.out.println(e.getMessage());
            return null;
        }
        }
}
```

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Map;

public class RtlOutFile {
    private boolean         d_headerWritten = false;
    private BufferedWriter  d_writer;
    private String          d_sep;

    public RtlOutFile(String filename, String sep) throws RtlException {
        try {
            d_sep = sep;
            d_writer = new BufferedWriter(new FileWriter(new File(filename)));
        } catch ( Exception e ) {
            throw new RtlException(e.getMessage());
        }
    }

    //  Write a line to the file, each element in the map seperated by d_sep
    //  string.
    //
    //  If a header has not yet been written, write one out first
    public void write(String[] header, Map<String, String> rec) {
        try {
            String line = join(header, d_sep);
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < header.length; ++i) {
                String data = "";
                if ( rec.containsKey(header[i]) ) {
                    data = rec.get(header[i]);
                }

                sb.append(data);
                if ( i + 1 < header.length ) {
                    sb.append(d_sep);
                }
            }

            if ( ! d_headerWritten ) {
                StringBuilder hd = new StringBuilder();

                for (int i = 0; i < header.length; ++i ) {
                    hd.append(header[i]);

                    if ( i + 1 < header.length ) {
                        hd.append(d_sep);
                    }
                }

                d_writer.write(hd.toString() + "\n");
                d_headerWritten = true;
            }

            d_writer.write(sb.toString() + "\n");
            d_writer.flush();
        } catch ( Exception e ) {
            System.out.println(e.getMessage());
        }
    }

    private String join(String[] strs, String sep) {
        StringBuilder sb = new StringBuilder();

        for (int i = 0; i < strs.length; ++i) {
            sb.append(strs[i]);

            if ( i + 1 < strs.length ) {
                sb.append(sep);
            }
        }

        return sb.toString();
    }
}
```