PB & J: Parallel Boxes and Jam

Final Report

Hahn Chong - hc2361
Fred Clark Jr. - fc2413
Rotem David - rd2499
Robert Tolda - rmt2131
Jose Rodriguez - jgr2128 - Team Leader

Table of Contents

Section 1: Introduction

1.1 Project Overview

Distributed computing is the use of multiple autonomous computers which work in concert to achieve a desired output. PB&J is a programming language designed to allow developers to distribute a job amongst multiple computers/processors using a minimal amount of code. In PB&J, developers define a program which is run on both the master and slave servers. While a portion of the program is only run on the master, the slave waits for messages from the master as to what functions to run on what data sets. It's the responsibility of the master server to distribute the job to the slave servers. A developer making a program in PB&J can use the spread & jam statements to perform these actions.

1.2 Background

We define a distributed system, minimally, as a system with multiple processors. Distributed computing is often used to harness the processing power of multiple machines that share a network connection. However, using our basic definition, it is possible to demonstrate the feasibility of implementing this programming language using a single machine with multiple

processors.

The primary motivation for developing PB & J was increase the speed in which a computation could be made through parallelism. Certain computations are inherently suited to be performed more quickly by distributed computing. Specifically, computations that have multiple stages, and where one stage is not dependant on any other stage. Hackers like to use distributed computing to brute force passwords.

Distributed computing has other applications besides parallelism. Distributed computing is particularly advantageous in critical applications where the partial failure property can be realized. Partial failure is somewhat of a redundancy in processing so that execution of the task is not lost if one processor fails, or in other words, fault -tolerant execution.

1.3 Related Work

There were almost 100 programming languages that had been adapted to address distributed computing according to “Programming Languages For Distributed Computing Systems,” a paper by Henri E. Bal et. al. This was in 1989, so we imagine there are several many more than that by now. To name a few of the notable languages discussed in the Bal paper: CSP, Concurrent C, Concurrent PROLOG, Linda.

Section 2: Language Tutorial

The following section gives a brief overview of language usage. To see more examples of code refer to the Demo Testing section.

2.1 Types and Declarations

PBJ supports primitive types of long, double, boolean, and string. When a variable is declared it begins with the data type and is assigned an alphanumeric identifier, while the rest of the identifier allows numbers it must start with a letter. Upon declaration each variable is assigned a default value (see section 3) unless it is explicitly assigned a value.

Example of a default initialization of string:

```
string myString;
```

Example of an explicit declaration of string:

```
string myString <- “Hello World”;
```

2.2 Functions

Functions, similar to type variables, require an identifier that starts with a letter and a data type for each argument. The most common function found in all PBJ files is master. Master unlike other functions in PBJ requires a specific signature as follows:

```
master(map slaves, array args) {
}
```

A function is allowed a return type but is not required. In the case of the previous example, master has no return type so it is omitted. If we were to declare a new function that returns a value, the signature is prepended with the return type.

```
string getString() {
        -> “Hello World”;
}
```

2.3 Compilation

PBJ source is compiled using the “pbjc” utility. This utility can be created by using the Makefile in compiler within our project source. pbjc does not only compile the program into java source but can also provide an ast printout which it does by default.

2.4 Packaging

Packaging is completed by the package script found in the root directory of our project. When uses the script a source file to compile is given as an argument.

For example:

```
./package.sh my_test.pbj
```

Once the provided source is compiled, the resulting Java source is then compiled along with the backend code of our project and any other needed dependencies.

These are then compressed into a runnable jar file.

2.5 Execution

Execution occurs by running the packaged jar in either a slave state or a master state. First all the slaves with a copy of the jar are initialized with the slave parameter and an optional port name (See section 3 for more details). With all the necessary slaves running we can them initialize the master with the slaves being used for the program as an argument.

Example of slave initialization:

```
java -jar PBJ.jar -slave
java -jar PBJ.jar -slave 9002
```

Example of master initialization:

java -jar PBJ.jar 127.0.0.1;127.0.0.1:9002

Section 3: Language Reference Manual

3.1 Types

3.1.1 Primitive Types

PB&J supports the following primitive types:

long	Basic numeric type.
double	Floating point value and is declared by adding a decimal point to an integer.
boolean	True or false expression.
string	Sequence of characters.

3.1.2 Collection Types

PB&J also supports more complex data types, known as collection types, that follow the syntactical structure of JSON.

Array	A fixed position of sequential data types.
Map	A key/value based data structure. Where keys can be any of the defined data types.

3.1.3 Special types

PB&J contains the following special values:

null	A special type that can be assigned to any variable type in place of it’s allowed value range.

3.2 Lexical Structure

3.2.1 Comments

In PB&J there are only single line comments. Comments are generated one line at a time, by placing three consecutive periods (...).

Example:
…This is a comment in PB&J.
… This is also a comment in PB&J.
.. . This is not a valid comment in PB&J

3.2.2 Operators

3.2.2.1 PB&J Operator (@)

Spread: @ is used to identify an explicit argument that is a collection to spread amongst the slave servers with
Jam: @ is used to identify the parameter in which to deliver the spread result.
Refer to section 5 for more details on spread and jam.

3.2.2.2 Assignment Operator: identifier <- expr

The assignment operator consists of <- to represent you are injecting the left identifier to the right expression.

3.2.2.3 Return Operator: -> expr

The return operator (->) returns the value of the expression on the right of the operator.

3.2.3 Literal Types

3.2.3.1 Literal Primitive Types

The following are examples of literal primitive types:

long	0 10 1
double	0.0 10.0 1.1
boolean	true, false

string	“Hello World”

3.2.3.2 Literal Data Types

DCC also supports more complex data types following the syntactical structure of JSON.

array	An example of an Array of longs: [1, 2, 3, 4]
map	An example of a map with keyed by longs (similar to the list above): { 1 : 1, 2 : 2, 3 : 3, 4 : 4} An example of a map keyed by strings: { “one” : 1, “two” : 2, “three” : 3, “four” : 4}

3.2.4 Keywords

The following keywords, formed from ASCII characters, are reserved and can not be used as identifiers:

jam spread long double boolean string array map null if else while for print global

3.2.5 Arrays and Maps

3.2.5.1 Arrays

Arrays a series of random access values that have a fixed length and are accessed by a number from 0 to length - 1.

Where “valuen” is value in the array, “index” is a number which corresponds to an address in the array, and “size” is the wanted size of the list as a long, and a is the identifier for the array.

Create an empty array	array a <- []
Create with values	array a <- [value1, value2, valuen]
Get the value for a given index	a[index]
Add or replace a value	a[index] <- value
Get length of the array	|a|

3.2.5.2 Maps

Where “keyn” is a wanted key in the form of a string or long and “valuen” is the wanted value for that key and m is the identifier for a wanted map.

Create an empty map	map m <- {}
Create with values	map m <- { key1 : value1, key2 : value2, keyn : valuen}
Get the value for a given key	m{key}
Add or replace the value for a key	m{key} <- value
Remove a key and it’s value	m{key} <- null
Get all the keys as an array	m*
Get all the values as an array	m{*}
Get length of map	|m|

3.3 Expressions

3.3.1 Operators

All Mathematical Operators follow traditional order of precedence with remainder ordered on the same level as division and multiplication.

3.3.1.1 Multiplicative Operators

Operators * and / are known as multiplicative operators. Multiplicative operators are allowed on numeric primitive data types and show described as follows:

3.3.1.1.1 Multiplication Operator

The * operator is used for multiplication of all numerical primitive data types. The following is a typical format for a multiplication operator:

*expr * expr2*

Multiplication Examples:

Expression	**Result**

1 * 2	2
1.0 * 2.5	2.5

3.3.1.1.2 Division Operator

The / operator is used for division of all numerical primitive data types. The following is a typical format for a division expression:

expr / expr2

Division Examples:

Division of longs rounds towards 0.

Expression	Result
1 / 2	0
3 / 2	1
3.0 / 2.0	1.5

3.3.1.1.3 Remainder Operator

The % operator is used for finding the remainder of all numerical primitive types. The following is a typical format for a remainder expression:

expr % expr2

Remainder Examples:

Division of integers rounds towards 0.

Expression	Result
4 % 2	0
4 % 3	1
2.5 % 2.0	0.5

3.3.1.2 Additive Operators

The + and - are known as additive operators.

3.3.1.2.1 Addition Operator

The + operator is used for addition of two expressions:

expr + expr2

Addiction Examples:

Expression	Result
1 + 2	3
1.0 + 0.5	1.5

3.3.1.2.2 Subtraction Operator

The - operator is used for addition of two expressions:

expr - expr2

Subtraction Examples:

Expression	Result
1 - 2	-1
1.0 - 0.5	0.5

3.3.1.3 Comparison Operators

We are dropping the == operator as to not create confusion.

=	Structural comparison
===	Physical comparison

3.3.1.4 Inequality Operators

Inequality operators can be used on the numeric types: long and double.

PB&J supports the following inequality operators:

>	Greater than
>=	Greater than or equal to
<	Less than

<=	Less than or equal to

3.3.1.5 Logical Operators

Since our language does not contain bitwise operators, we can simplify our logical operators to one character. Operators are still short-circuit operators.

&&	And operator
||	Or operator

3.3.1.6 String Concatenation

The ~ (tilde) is used to concatenate data types into a string.

Examples:

Expression	*Evaluation*
4 ~ 2	“42”
“Hello “ ~ 0.0	“Hello 0.0”

3.4 Blocks and Statements

3.4.1 Blocks

The start of a block is defined by open brace ({) where { is not following the identifier for a map.
The end of a block is defined by close brace (}) where } is not preceded by an unclosed hash’s {. In the case of nested blocks } closes the last opened block.
Blocks for functions and statements are defined using braces { }

3.4.2 Spread Statement

spread: *func*
The spread statement is used to distribute a collection type amongst registered slave machines. The function acts as a callback returning each time a slaving reports results, but only blocks the call for the first result. *func* is a reference to the function that runs on the spread collection.

3.4.3 Jam Statement

jam: *func*

The jam statement is used to block on a spread statement until each slave reports a result. It will then return the results of the spread or the result of the given function. *func* is an optional reference to a function, which is run on the resulting collection set.

Note that jam: always precedes a spread statement with only an optional function reference in between.

3.4.4 Conditional Statement

The conditional statement has two forms:

if(*expr*) *statement1*

if(*expr*) *statement1* else *statement2*

In both cases *expr* is a boolean expression that is evaluated first and *statement1* will be executed if *expr* evaluates true. In the second case, *statement2* is executed if expr evaluates false.

3.4.5 While Statement

while(*expr*) *statement*

The while statement repeatedly executes *statement* until *expr* evaluates false. *expr* is a boolean expression that is evaluated before the statement is executed.

3.4.6 For Statement

for(*expr1*; *expr2*; *expr3*) *statement*

The for statement first evaluates *expr1* once. *expr2* is a boolean expression that is evaluated before each iteration. *statement* is executed whenever *expr2* evaluates true and *expr3* is evaluated after each time *statement* is executed. If *expr2* evaluates false, the for statement is terminated. Any or all of the expressions may be left empty. If *expr2* is left empty, it will be evaluated as true.

3.4.7 Print Statement

print(*string*)

The print statement will print *string*. Other types will be converted to a string and printed.

3.5 Declarations & Identifiers

Both function declaration and variable declarations require a valid identifier. A valid identifier must start with an alphabetic unicode character followed by any sequence of alphanumeric

unicode characters. An identifier may not be one of the reserved keywords. Two identifiers are the same, if and only if, they have identical unicode characters for each letter of digit.

Examples of identifiers
long a … a is a valid identifier
long b123 … b123 is a valid identifier
long 1ba … 1ba is NOT a valid identifier as it does not start with an alphabetic character

3.5.1 Function Declarations

A function is a body of executable code which is passed a specific number of parameters. To declare a function into a program specify a unique identifier. The identifier is used to refer to the function for the duration of the program. Function declarations must follow the following format:

[type] *identifier*(**type** param [,...]) {
… body of function
}

When defining a function, it is optional to provide a type before the identifier to define the return type of the function. It is also acceptable to leave out the return type if the function has no return.

3.5.1.1 Master Function

For each program to run on a master node, a function with the identifier **"master"** is required. A **"master"** function must take two parameters. The first parameter must have a map type. The map corresponds to a map of server objects. The second parameter being an array of strings contains any additional command line arguments when ran. A **"master"** function must follow the following format:

```
master(map servers, array params) {
        … body of master function
}
```

Where the identifiers, servers and params, are interchangeable.

3.5.2 Variable and Constant Declarations

Variables and constants are declared by a unique *identifier* with the scope (in the case of constants) and data type, respectfully, specified before the *identifier*. The scope determines where the variable or constant is accepted by the compiler. Declare a variable without the global keyword the context of a block to specify a local variable. Use the specifier *global* before the type declaration to declare a global constant which cannot be changed and is accessible by the

master and all servers. The global specifier can only be used outside of the block scopes.

Note that this means the only way how to declare something as constant defines it as global and visa versa.

[specifier] type ***identifier***

3.5.2.1 Variable and Constant Initialization

Variables and constants in PB&J can be initialized with a given literal value or another already declared variable or constant that can be seen within the scope of the initializing variable. Any variable or constant that is not provided a value with the declaration is initialized one of the following default values:

Data Type	Default Value
long, double	0, 0.0
boolean	false
string	“” (empty string)
array	[] (empty array)
map	{} (empty map)

3.6 Scoping

3.6.1 Global Scope

The global constants are part of the global scope and can be accessed in all of the functions in the file. defined by the keyword “global” followed by the type and name of the constant.

Global Scope Usage:
global double newresult <- 10.0;

master(double d) { ... d is a local variable.
print("Result: " ~ newresult); ... newresult can be accessed in
... the function because it’s a global variable
}

3.6.2 Block Scope

Each block defines it's own local scope. Blocks have access to their parent block's local variables that have already been defined, as well as global variables.

Block Scope Usage:

```
foo( double a) { ... a is a local variable.
        double b <- a;  ...b is a local variable
        if(true) {
        double d <- a; ... Legal, the if block has access
... to it's parent block's variables.
        }
        b <- d; ... Illegal - d is not in the scope of this block.
}

master(double c){ ... c is a local variable.
        c <- b + 3.0;  ... ILLEGAL - b is a local variable
... in the foo function and thus cannot be
... accessed in the master function
}
```

3.7 Execution

Master and slave nodes of PB&J both execute the same version of compiled code to run. Execution of a PB&J compiled application happens in two phases. The first phase is to initialize the program as a slave daemon on all available slaves. The second is to initialize the master program with the desired slaves as an argument.

3.7.1 Slave Daemon

To initialized the slave daemon, you simply run the compiled version of your program with the *-slave* argument with an optional port number. When initialized without a port number the slave defaults to TCP port 35000.

```
slave$ programName -slave [port_number]
```

3.7.2 Master Execution

When a program is executed without the -slave argument then by default it'll run the **"master"** function of the program with a list of acceptable slaves

```
master$ programName ip[:PORT];ip2[:PORT];...
```

Section 4: Project Plan

4.1 Planning

The PB & J team met Tuesday afternoons to give updates on progress and to plan out course of action for the upcoming week. We also sent out emails to the whole group to meet additional times as needed (meeting almost every day for the last week). For developing the compiler, the guiding principle which we used to design our planning process was inspired by Prof. Edwards: to try to implement all the functions that we could in the allotted time. We created a list of the most essential features and then we assigned ourselves to them. Then we further fleshed out the compiler by assigning ourselves more aspects of it.

4.2 Specification

For specification, members of PBJ referred to the LRM for each feature being developed. During our planning stages each feature was worked out in detail for the LRM so it was easy to reference it as a specification. The specification was only changed when we approached features that caused ambiguities and conflicts in the language.

4.3 Development

Each contributor pulled the original *master* compiler files that Jose created from the GitHub repository, and then started a branch of their own. Additional functionalities were coded and tested independently before being merged with the master branch. Each phase of the development cycle consisted of a pull from the master branch, and a subsequent merge. We did encounter some conflicts when code was merged that was too far behind the current master branch. To resolve this we had to reconcile the files manually.

4.4 Project Timeline

The following timeline was created at the beginning of the semester. We were basically able to stick to this .

Milestone	**Date**
PB & J Proposal Submission	September 26, 2012
PB & J Language Reference Manual Submission	October 31, 2012

PB & J Compiler Preliminary Functionality	November 11, 2012
PB & J Compiler Advanced Functionality	December 15, 2012
PB & J Final Report Submission and Presentation	December 19, 2012

4.5 Project Log

Implementation of our project was managed in features. Each member would be assigned a feature that they would work on in their own branch. Once the feature is completed it is them merged into the master repository.

The initial version of features was tracked with the following README.md found in our repository at:

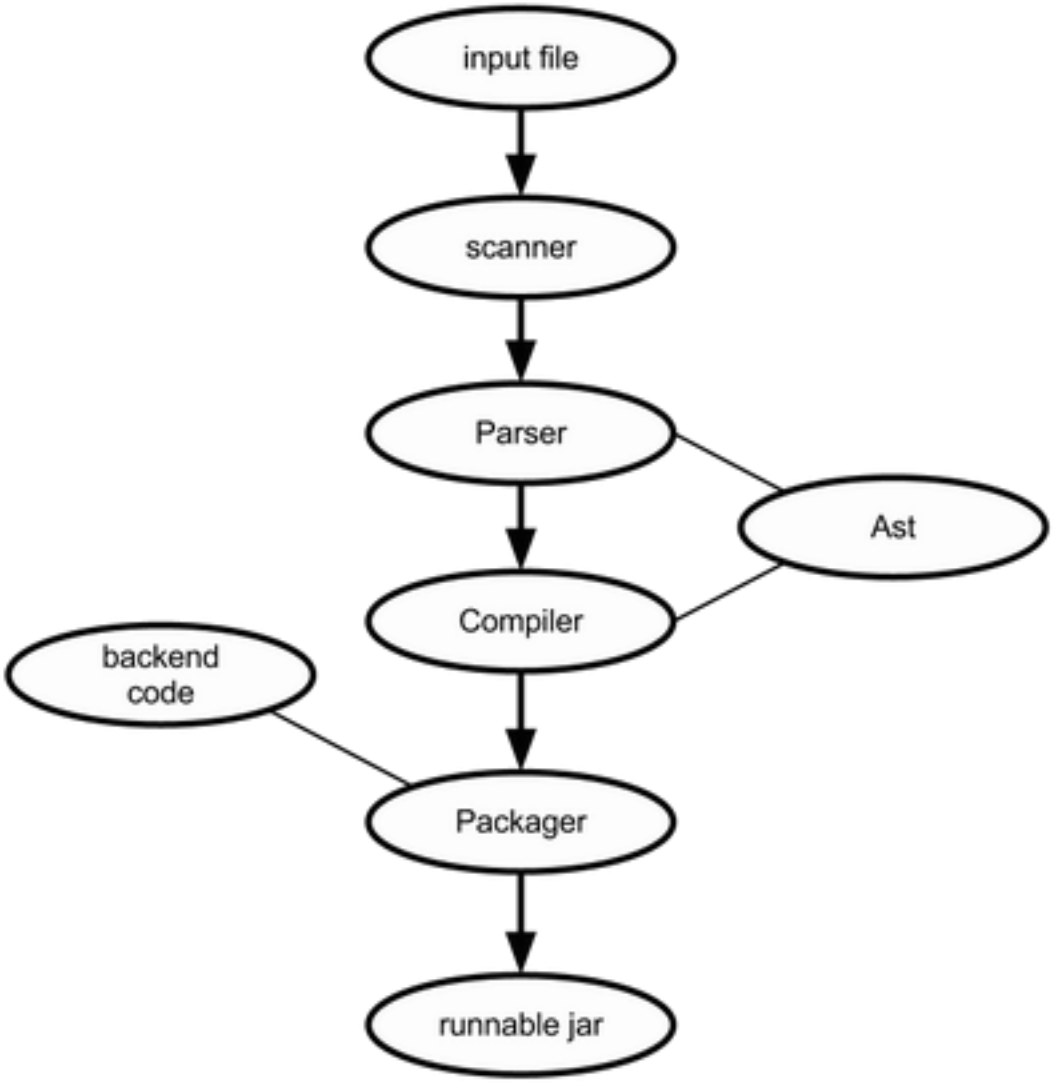
- Types
  - Primitive Types
    - long (Fred)
    - double (Fred)
    - string(Rotem)
    - boolean (Rotem)
    - null (Robert)
  - Collection Types
    - Map (Jose)
    - Array (Robert)
  - Globals (Rotem)
- Lexical
  - Comments (Jose)
  - Operators
    - PBJ Op (Robert)
    - Assignment operator (Jose)
    - Return (Hahn)
  - Literal Types
    - Literal primitives - Apart of types
  - Arrays (Robert)
  - Maps (Jose)
- Expressions
  - Operators
    - Arithmetic (Hahn)
      - Compiler checking for binop. (Hahn)
    - Comparison (Hahn)
    - Inequalities (Hahn)

        - Logical operators (Hahn)
        - String concat (Jose)
    - Assignment (Jose)
    - Func Call (Rotem)
- Blocks & Statments
    - Blocks (Jose)
    - Spread (Robert)
    - Jam (Robert)
    - Conditional (Robert)
    - While (Robert)
    - For (Robert)
    - Print(Rotem)

After the basic implementation of each feature, available members would then implement the semantic analysis and code generation, both are found in compile.ml. It was very common in our project for one team member to implement the basics of a feature and another member implement the semantic analysis when other features were advanced enough to do so.

Section 5: Architectural Design

5.1 High Level Architectural Design



5.2 Component Interface Interaction

5.2.1 Scanner (scanner.mll)

Takes the Input file and converts it into a series of tokens used by the parser.

5.2.2 Parser (parser.mly)

Takes the tokens generated by scanner and assembles them into an Abstract Syntax Tree using the type definitions in ast.ml.

5.2.3 AST (ast.ml)

Contains the type definitions for constructing the AST and functions to print it out. Accessed by the parser and the compiler.

5.2.4 Java Code Generator\Semantic Checker (compile.ml)

Utilizes the AST produced from parser.mly along with the type definitions in ast.ml to produce java code.

5.2.5 Java Backend code

Code that is necessary to perform spreading and jamming (as those require communication over a port and procedural functions such as “slicing” an array or map into smaller pieces to be worked on by each slave)

5.2.6 Packager

Unites the backend code with the compiled .pbj code to create a jar file that executes the pbj program with the functionality provided by the backend.

Section 6: Test Plan

In our development workflow, it was expected that each member run our test scripts accompanying the compiler and ensure all tests passed. By design, the master repository at all times would succeed on all tests. If a team member’s code failed the test, it was their responsibility to resolve the issue before committing.

6.1 Unit Test cases

Below is a listing of all the test cases included in the project. These tests can also be seen in our repo location at: https://github.com/josebezme/pb-j/tree/master/compiler/test

Our project contained both success and failure tests. Success cases are typically easily rolled up into single files such as valid_operations.pbj where all the operations are used. Failure cases required a test per failing error.

Test cases were ran with the run-test.sh script and also ran with the run-ast-test.sh script to detect parsing errors in failure tests.

array_assign_fail.pbj
array_assign_id.pbj
array_declare.pbj
array_get.pbj
array_get_id.pbj
array_get_id_fail.pbj
array_get_literal_fail.pbj

array_get_size.pbj
array_literal.pbj
array_literal_in_literal.pbj
array_put.pbj
bad_assign_bool_lit_string_fail.pbj
bad_assign_map_id_string_fail.pbj
bad_assign_map_lit_string_fail.pbj
bad_assign_string_id_map_fail.pbj
bad_assign_string_lit_map_fail.pbj
bad_declare_assign_string_lit_map_fail.pbj
bad_map_get_fail.pbj
bad_map_put_fail.pbj
bool.pbj
bool_init.pbj
comments.pbj
concat_literal_str.pbj
control_do_while.pbj
control_for.pbj
control_for_expr_fail.pbj
control_for_stmt_fail.pbj
control_if.pbj
control_if_else.pbj
control_while.pbj
demo_factorial.pbj
demo_factorization.pbj
demo_prime.pbj
demo_tree.pbj
double_tests.pbj
func_assign_string.pbj
func_main.pbj
func_two.pbj
func_type_all.pbj
func_type_long.pbj
func_type_long_fail.pbj
func_type_long_type_fail.pbj
func_type_string.pbj
func_type_string_fail.pbj
globals.pbj
long_assign_binop.pbj
long_init.pbj
long_lit.pbj
map_declare.pbj
map_entries.pbj
map_entry.pbj

map_get.pbj
map_literal_of_literal.pbj
map_put.pbj
null.pbj
op_arith_fail.pbj
op_comp_fail.pbj
op_logic_fail.pbj
op_peq_fail.pbj
pbj_jam_spread.pbj
pbj_jam_spread_default.pbj
pbj_slice_spread.pbj
pbj_spread.pbj
return_order_fail.pbj
size_assign.pbj
stmt_block.pbj
string_assign.pbj
string_assign_concat.pbj
string_assign_return.pbj
string_init.pbj
valid_operations.pbj

6.2 Demo Test Cases

Once the project became advanced enough it was important to put together larger functional tests to see the compiler in action along with the source code creation. These tests were typically prefixed with demo and allowed us to see how our languages elements worked when they were all combined compared to the simple unit test cases of features.

6.2.1 Prime Factorization

Below is an example of a demo test case named: demo_factorization.pbj which will return the prime factors of a bi-prime number.

```
...This Program breaks the job of finding the factors of a number up
...and uses multiple slaves to solve it

master(map slaves, array args){ ... Runtime argument.
        long n <- args[0];
        array searchStarts;

... get the place for each slave to start
        long iterations <- (n / |slaves|) - 1; ... size of slaves
        long start <- 1;
        for(long m <- 0; m < |slaves|; m <- m + 1) {
```

```
                searchStarts[m] <- start + (m * 2);
        }

... spread the starting points to the slaves
        array result <- jam: spread: factor(@searchStarts, n, |slaves|);
        print("Result: " ~ result);
}

array factor(array starts, long n, long slaves) {
        long start <- starts[0];

        array factors;
        ... Check special case for 2.
        if(start = 1) {
                if(n % 2 = 0) {
                        factors[|factors|] <- 2;
                }
        }

        for(long i <- start; i <= n / 2; i <- i + (2 * slaves)) {
                print("Trying " ~ i);
                if( i > 1 && n % i = 0) { ...it is not prime
                        factors[|factors|] <- i;
                }
        }

        if(|factors| > 0) {
                -> factors;
        }
        -> null;
}
```

6.2.2 Depth First Search

...Here is an example of a program that performs Depth First Search on a ternary tree to find the parent node of the “goal.” in the file demo_tree.pbj
...it takes advantage of pbj to split the tree into three subtrees to be searched separately

```
master(map slaves, array args){
	map tree <- {"start": ["b", "c", "d"], "b": ["e", "f", "g"], "f": ["h", "i", "j"], "i":
["l", "m", "goal"], "c": ["p", "q", "r"], "d": ["s", "t", "u"], "t": ["v", "w", "x"] };...the tree
	print(spread: search(@tree{"start"}, tree ));
}

string search(array a, map tree){
	string out <- null;
	array currentNodesArray <- tree{a[0]};
	if ( currentNodesArray === null ){
		-> null;
	}...base case , we are on a leaf
	for (long i <- 0; i < |currentNodesArray|; i <- i + 1){
			if( currentNodesArray[i]  = "goal"){
				->  a[0];
			} else{
				string t  <- search( [currentNodesArray[i]], tree );
				boolean b <- (t === null);
				if(b === false) -> out; ...we found the parent node
			}
	}
	-> null; ...we didn’t find it return null
}
```

Section 7: Lessons Learned

7.1 Jose

7.1.1 Lessons Learned

The largest lesson I’ve learned from this is that it’s better to implement an additional filter containing all the semantic checking. Currently this happens in our compiler and it’s very much coupled with our java source generation. The source generation in itself is hard to read so it

makes tasks of adding more semantic checking even more difficult.

While I enjoyed the way we implemented the language feature by feature, If I could do this project over again, I would definitely add a layer that's designed specifically for semantic checking.

I would have also liked to work more with an SAST. Since we started our compiler before they were covered in class, we didn't implement a SAST. It would have been nice if this would covered sooner.

7.1.2 Advice for Future Teams

Developing by feature is a great way to get the project completed but also a great way to create a lot of conflicts. It's important to have a testing system and to have each member of the team understand the testing requirements before committing code.

7.2 Rotem

7.2.1 Lessons Learned

Working on this project and taking PLT this semester was a great learning experience for me. I've learned a lot about compilers, scanners, and parsers and most importantly, how to fit them all together to create a programming language. In addition, I got to play with functional programming and understand its significance. I have to admit, at first I was reluctant to try programming in anything that's different than java. However, after coding in Ocaml, I began to enjoy writing recursive functions as well as writing compact code. What's more, the error handling messages were very useful and to the point. Furthermore, working on this project with my group-members enabled me to learn the fundamentals of Git since we had to collaborate and to delegate tasks to one another. We each worked on our own branches and had to merge everything to master once we finished a task. Moreover, merging branches resulted in a lot of conflicting commits (since we were oftentimes working on the same file), and it was a crucial lesson to learn; one that helped me to better understand Git and that will prepare me for working in groups in the real-world.

7.2.2 Advice for Future Teams

My advice for future teams is to first make sure that everyone in the group knows how to use Git/version-control before starting to code. This is crucial since a person might work on a task and not push it properly and that could affect everyone elses code since it affects the remote branch that everyone is sharing. Also, make sure that your teammate's schedule does not conflict since this it is a big project and you are expected to meet a lot throughout the semester. Lastly, make sure not to miss the lectures that cover Microc as it is the fundamental building blocks for creating your team's programming language. Other than that, Good luck!

7.3 Robert

7.3.1 Lessons Learned

- You can start to feel comfortable with something as complex as writing a compiler in OCaml if you do it enough.
- Well developed initial ideas are important.
- Although I learn this in every semester: it is extremely important to learn how to use the tools (such as the language and the version control) well, as early in the project as possible. Unfortunately the only way how to learn how to use the tools well (for me at least) is to use them for a long time. I am still not quite sure what the solution is...

I also learned how to use version control. I was surprisingly not taught this previously. I also learned that version control is both a blessing and a curse: it can make certain actions such as organizing separate copies of the same file and merging the copies easier through branching, but is can make small changes take a long time since there are often a number of steps that need to be taken. (pulling, adding, committing, pushing)

7.3.2 Advice for Future Teams

- Comment your code. The time it takes for you to comment the code will be dramatically less than the extra time it takes for other people to try to otherwise figure it out.
- Keep an eye on your teammates since they may forget something you implemented and you may not have time to make it work with their stuff when you find out.
- Don't just rely on small tests only. Writing real programs reveals all sorts of important cases you would otherwise miss.
- Be clear about what each person is doing so there is no overlap.

7.4 Hahn

7.4.1 Lessons Learned

Through working on the project I was able to become quite familiar with OCaml and in the end I think it is a nice language to know how to program in. Also because we split up our tasks by feature I was able to learn a lot about implementing every part of the compiler. I also learned about version control with GitHub and will definitely be using it in future projects.

7.4.2 Advice for Future Teams

Main advice would be to consider splitting up the work by feature rather than module. I felt that I was able to learn more by having to code each section of the compiler, than I would have if I only wrote one part of the compiler and having the others explain how their parts work. Also, remember to commit and merge your changes on GitHub as soon as you can. I eventually abandoned my branch and decided to work straight from the master branch because I fell too far behind in commits, and resolving all the conflicts would have been too tedious to do.

7.5 Fred

7.5.1 Lessons Learned

Throughout the semester PLT has taught me functional programming through Ocaml. Functional programming is interesting as it is based on the Lambda Calculus. I think that it is really neat how recursion and pattern matching is done in this language. I learned how the files that make up the compiler work and I became familiar with GitHub.

7.5.2 Advice for Future Teams

I would recommend that future PLT students meet regularly and keep up with the project throughout the semester. You definitely don't want to try to put something like this together over night. Also Version Control can't be overstated. Make sure everyone knows how to use it, maybe even make your first meeting all about learning how to use it.

Appendix: Code Listing

The following code listings can be found in a much more readable format at our public repo located here:
https://github.com/josebezme/pb-j

scanner.mll

```
{ open Parser }


rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "..."    { comment lexbuf }
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '|'      { PIPE }
| "**"             { STARSTAR }
| '~'      { CONCAT }
```

```
| "for"             { FOR }
| "while"          { WHILE }
| "do"             { DO }
| "if"     { IF }
| "else"           { ELSE }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'              { MOD }
| '='      { SEQUAL }
| "==="            { PEQUAL }
| '>'      { GT }
| ">="             { GTE }
| '<'      { LT }
| "<="             { LTE }
| "&&"             { AND }
| "||"     { OR }
| "!="             { NEQ }
| "spread:"   { SPREAD }
| "jam:"           { JAM }
| '@'              { AT }
| "map"          { MAP }
| "array"          { ARRAY }
| "print"          { PRINT }
| "string"         { STRING }
| "<-"             { ASSIGN }
| "boolean"      { BOOLEAN }
| "true"           { BOOLEAN_LITERAL(true) }
| "false"          { BOOLEAN_LITERAL(false) }
| "long"           { LONG }
| "double"       { DOUBLE }
| "null"           { NULL }
| "global"         {GLOBAL}
| (['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id) "[" { ARRAY_BEGIN(id) }
| '"' ([^'"']+ as s) '"'   { STRING_LITERAL(s) }
| ['0'-'9']* ['.'] ['0'-'9']+ as lxm { DUB_LITERAL(lxm) }
| ['0'-'9']+ as lxm { LONG_LITERAL(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof              { EOF }
| "->"             { RETURN }
| _  as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
```

```
  '\n' { token lexbuf }
  | eof { EOF }
  | _ { comment lexbuf }
```

parser.mly

```
%{ open Ast

let parse_error s = (* Called by the parser function on error *)
  print_endline s;
  flush stdout
%}

%token SEMI COLON LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
COMMA BAR
%token MAP ARRAY STRING LONG DOUBLE BOOLEAN
%token IF WHILE FOR ELSE DO
%token NULL
%token STARSTAR PIPE CONCAT
%token PLUS MINUS TIMES DIVIDE MOD
%token SEQUAL PEQUAL GT GTE LT LTE AND OR NEQ
%token COMMENT
%token ASSIGN
%token RETURN
%token PRINT
%token AT
%token SPREAD
%token JAM
%token <string> ARRAY_BEGIN
%token <bool> BOOLEAN_LITERAL
%token <string> STRING_LITERAL
%token <string> ARRAY_LITERAL
%token <string> ID
%token <string> LONG_LITERAL
%token <string> DUB_LITERAL
%token GLOBAL
%token EOF

%right JAM NOJAMFUN SPREAD
%nonassoc NOELSE
%nonassoc ELSE
%left ID LBRACKET RBRACKET CONCAT
%right RETURN
%right ASSIGN
```

```
%left AND OR
%left SEQUAL PEQUAL GT GTE LT LTE NEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD

%start program
%type <Ast.program> program

%%

program:
  { [], [] }
  | program fdecl { fst $1, ($2 :: snd $1)}
  | program GLOBAL vdecl ASSIGN prim_literal SEMI { ( ($3,$5) :: fst $1), snd $1 }


fdecl:
    ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    {{ fname = Void($1); formals = $3; body = List.rev $6 }}
  | vdecl LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
          {{ fname = $1; formals = $3; body = List.rev $6 }}

formals_opt:
  { [] }
  | formal_list   { List.rev $1 }

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

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

actuals_list:
          expr              { [$1] }
  | AT expr                 { [At($2)] }
  | actuals_list COMMA AT expr { At($4) :: $1 }
  | actuals_list COMMA expr { $3 :: $1 }

map_entry_list:
  { [] }
  | map_entry { [$1] }
  | map_entry_list COMMA map_entry { $3 :: $1 }
```

```
map_entry:
  | prim_literal COLON expr { ($1, $3) }

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

prim_literal:
  STRING_LITERAL { StringLiteral($1) }
  | LONG_LITERAL { LongLiteral($1) }
  | DUB_LITERAL {DubLiteral($1)}
  | BOOLEAN_LITERAL { BooleanLiteral($1) }

stmt_expr_opt:
  { NoExpr }
  | stmt_expr { $1 }

stmt_expr:
  ARRAY_BEGIN expr RBRACKET ASSIGN expr { ArrayPut($1, $2, $5) }
  | ID LBRACE expr RBRACE ASSIGN expr { MapPut($1, $3, $6) }
  | ID ASSIGN expr   { Assign($1,$3) }
  | ID LPAREN actuals_opt RPAREN { FunctionCall($1,$3) }
    | SPREAD stmt_expr         { Spread($2) }
    | JAM stmt_expr SPREAD stmt_expr { JamSpread($2, Spread($4)) }
    | JAM %prec NOJAMFUN stmt_expr { JamSpread( NoExpr, $2) }

expr:
  stmt_expr { StmtExpr($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 MOD expr { Binop($1, Mod, $3) }
  | expr SEQUAL expr { Binop($1, Seq, $3) }
  | expr PEQUAL expr { Binop($1, Peq, $3) }
  | expr NEQ expr { Binop($1, Neq, $3) }
  | expr GT expr { Binop($1, Greater, $3) }
  | expr GTE expr { Binop($1, Geq, $3) }
  | expr LT expr { Binop($1, Less, $3) }
  | expr LTE expr { Binop($1, Leq, $3) }
  | expr AND expr { Binop($1, And, $3) }
```

```
  | expr OR expr { Binop($1, Or, $3) }
  | NULL          { Null }
  | prim_literal { Literal($1) }
  | ARRAY_BEGIN expr RBRACKET { ArrayGet($1, $2) }
    | LBRACKET array_list RBRACKET  { ArrayLiteral(List.rev $2) }
  | LBRACE map_entry_list RBRACE { MapLiteral($2) }
  | ID LBRACE expr RBRACE { MapGet($1, $3) }
  | ID STARSTAR { MapKeys($1) }
  | ID LBRACE STARSTAR RBRACE { MapValues($1) }
  | PIPE ID PIPE { Size($2) }
  | expr CONCAT expr { Concat($1, $3) }
  | LPAREN expr RPAREN { $2 }

vdecl:
  MAP ID { Map($2) }
  | LONG ID {Long ($2) }
  | DOUBLE ID {Double ($2)}
  | ARRAY ID { Array($2) }
  | STRING ID { String($2) }
  | BOOLEAN ID { Boolean($2) }

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

stmt_opt:
  SEMI { NoStmt }
  | stmt { $1 }

stmt:
  vdecl SEMI { Declare($1) }
  | stmt_expr SEMI { ExprAsStmt($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | vdecl ASSIGN expr SEMI { DeclareAssign($1, $3) }
  | PRINT LPAREN expr RPAREN SEMI { Print($3) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt          { If($3, $5, $7) }
  | FOR LPAREN stmt_opt expr SEMI stmt_expr_opt RPAREN stmt
          { For($3, $4, $6, $8) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | DO stmt WHILE LPAREN expr RPAREN SEMI { DoWhile($2, $5) }
```

ast.ml

```
type op = Add | Sub | Mult | Div | Mod | Seq | Peq | Greater | Geq | Less | Leq | Neq | And | Or

type data_type =
  String  of string
  | Map   of string
  | Array of string
  | Boolean of string
  | Long of string
  | Double of string
  | Void of string

type literal =
  | StringLiteral of string
  | LongLiteral of string
  | DubLiteral of string
  | BooleanLiteral of bool

type stmt_expr =
  Assign of string * expr
  | ArrayPut      of string * expr * expr
  | MapPut of string * expr * expr
    | Spread      of stmt_expr
    | JamSpread of stmt_expr * stmt_expr
  | FunctionCall of string * expr list
  | NoExpr

and expr =
  StmtExpr of stmt_expr
  | Id of string
  | Literal of literal
  | Binop of expr * op * expr
  | ArrayGet      of string * expr
  | ArrayLiteral  of expr list
  | Null
  | MapLiteral of (literal * expr) list
  | MapGet of string * expr
  | MapKeys of string
  | MapValues of string
  | Size of string
  | Concat of expr * expr
  | At of expr
```

```
type stmt =
        Block of stmt list
  | Print of expr
  | Return of expr
  | Declare of data_type
  | DeclareAssign of data_type * expr
  | If of expr * stmt * stmt
  | For of stmt * expr * stmt_expr * stmt
  | While of expr * stmt
  | DoWhile of stmt * expr
  | ExprAsStmt of stmt_expr
  | NoStmt

type func_decl = {
        fname : data_type;
        formals : data_type list;
        body : stmt list;
  }

type program = (data_type * literal) list * func_decl list

let rec string_of_data_type = function
  String(s) -> "STRING-" ^ s
  | Map(s) -> "MAP-" ^ s
  | Array(s) -> "ARRAY-" ^ s
  | Boolean(s) -> "BOOLEAN-" ^ s
  | Long(s) -> "LONG-" ^ s
  | Double(s) -> "DOUBLE-" ^ s
  | Void(s) -> "VOID-" ^ s

let rec string_of_literal = function
  StringLiteral(s) -> "\"" ^ s ^ "\""
  | LongLiteral(l) -> "LONG_LIT: " ^ l
  | DubLiteral(l) -> "DUB_LIT: " ^ l
  | BooleanLiteral(s) -> string_of_bool s

let rec string_of_stmt_expr = function
  Assign(s, e) -> "ASSIGN " ^ s ^ " TO " ^ string_of_expr e
  | ArrayPut(id, idx, e)   -> "ARRAY-" ^ id ^ "-PUT[" ^ string_of_expr idx ^ ", " ^ string_of_expr e
^ "]"
  | MapPut(id,key,v) -> "MAP-" ^ id ^ "-PUT{" ^ string_of_expr key ^ ", " ^ string_of_expr v ^ "}"
  | FunctionCall(s,e) ->"FUNCTION CALL " ^ s ^   "{\n" ^ String.concat "," (List.map
string_of_expr e) ^ "}\n"
```

```
  | Spread(f)    -> "SPREAD AND CALL " ^ string_of_stmt_expr f
    | JamSpread(f, s) -> "JAM THE RESULTS OF " ^ string_of_stmt_expr s
                  ^ " WITH " ^ string_of_stmt_expr f
  | NoExpr -> "NoStmtExpr"

and string_of_expr = function
  StmtExpr(e) -> string_of_stmt_expr e
  | Literal(l) -> string_of_literal l
  | Id(s) -> "ID:" ^ s
  | Binop(e1, o, e2) -> "BINOP:" ^ string_of_expr e1 ^ " " ^
        (match o with
    Add -> "PLUS"
        | Sub -> "MINUS"
        | Mult -> "TIMES"
        | Div -> "DIV"
        | Mod -> "MOD"
        | Seq -> "SEQUAL"
        | Peq -> "PEQUAL"
        | Greater -> "GT"
        | Geq -> "GTE"
        | Less -> "LT"
        | Leq -> "LTE"
        | And -> "AND"
        | Neq -> "NEQ"
        | Or -> "OR") ^ " " ^ string_of_expr e2
  | ArrayGet(id,idx) -> "ARRAY-" ^ id ^ "-GET[" ^ string_of_expr idx ^ "]"
  | ArrayLiteral(al) -> "ARRAY[" ^ String.concat ","
        (List.map (fun e -> string_of_expr e ) al) ^ "]"
  | MapLiteral(ml) -> "MAP{" ^ String.concat ","
        (List.map (fun (a,b) -> string_of_literal a ^ ":" ^ string_of_expr b) ml) ^ "}"
  | MapGet(id,key) -> "MAP-" ^ id ^ "-GET{" ^ string_of_expr key ^ "}"
  | MapKeys(id) -> "MAP-KEYS-" ^ id
  | MapValues(id) -> "MAP-VALUES-" ^ id
  | Size(id) -> "SIZE-of-" ^ id
  | Concat(e1, e2) -> "CONCAT(" ^ string_of_expr e1 ^ "," ^ string_of_expr e1 ^ ")"
  | Null -> "NULL"
    | At(e)         -> "SPREADING: " ^ string_of_expr e

let rec string_of_stmt = function
        Block(stmts) ->
        "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Print(str) -> "PRINT (" ^ string_of_expr str ^ ");\n"
  | Return(e) -> "RETURN " ^ string_of_expr e ^ ";\n"
    | If (p, t, f) ->
```

```
        "IF " ^ string_of_expr p
                                    ^ " THEN DO "
        ^ string_of_stmt t
        ^ " ELSE DO " ^ string_of_stmt f ^ ";\n"
  | For (s, e, se, b) ->
                "DECLARE: " ^ string_of_stmt s
        ^ " AND DO " ^ string_of_stmt b
                                        ^ " WHILE " ^ string_of_expr e
                                        ^ " PERFORMING " ^ string_of_stmt_expr se ^ ";\n"
  | While (e, b) -> " WHILE " ^ string_of_expr e ^ " DO " ^ string_of_stmt b ^ ";\n"
  | DoWhile (b, e) -> " DO " ^ string_of_stmt b ^ " WHILE " ^ string_of_expr e ^ ";\n"
  | ExprAsStmt(e) -> "EXPR: " ^ string_of_stmt_expr e ^ ";\n"
  | Declare(dt) -> "DECLARE: " ^ string_of_data_type dt ^ ";\n"
  | DeclareAssign(dt, e) -> "DECLARE: " ^ string_of_data_type dt ^
        " AND ASSIGN: " ^ string_of_expr e ^ ";\n"
  | NoStmt -> "NoStmt"

let string_of_fdecl fdecl =
  "FUNCTION " ^ string_of_data_type fdecl.fname ^ "(" ^
        String.concat "," (List.map string_of_data_type fdecl.formals) ^
        ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_vdecl id = "long " ^ id ^ ";\n"

let string_of_globals globals =
  "GLOBALS " ^ string_of_data_type (fst globals) ^ " = " ^ string_of_literal (snd globals)

let string_of_program (vars, funcs) =
  "Vars: \n" ^ String.concat ";\n" (List.map string_of_globals vars) ^ "\n" ^
  "Funcs: \n" ^ String.concat "\n" (List.map string_of_fdecl funcs)
```

compiler.ml

```
open Ast

let imports =
  "import java.util.ArrayList;\n" ^
  "import java.util.Arrays;\n" ^
  "import java.util.List;\n" ^
  "import java.util.Map;\n" ^
  "import plt.pbj.util.PBJOp;\n"
```

```
let translate (globals, functions) =

  (* This just returns the string name for a data type *)
    let rec get_dt_name = function
      String(id) -> id
      | Map(id) -> id
      | Array(id) -> id
      | Boolean(id) -> id
      | Double(id) -> id
      | Long(id) -> id
      | Void(id) -> id

    (* Returns the java declaration of a datatype *)
  in let rec string_of_data_type with_id = function
      String(id) -> "String " ^ (if with_id then id else "")
      | Map(id) -> "Map<Object, Object> " ^ (if with_id then id else "")
      | Array(id) -> "List<Object> " ^ (if with_id then id else "")
      | Boolean(id) -> "Boolean " ^ (if with_id then id else "")
      | Long(id) -> "Long " ^ (if with_id then id else "")
      | Double(id) -> "Double " ^ (if with_id then id else "")
      | Void(id) -> "void " ^ (if with_id then id else "")

   (* Turns a literal object into a java expr *)
    in let rec string_of_literal = function
      StringLiteral(s) -> "\"" ^ s ^ "\""
      | DubLiteral(s) -> "new Double(" ^ s ^ ")"
      | LongLiteral(s) -> "new Long(" ^ s ^ ")"
      | BooleanLiteral(s) -> string_of_bool s

  in let rec check_globals id globals = match globals with
    [] -> false
    | hd :: tl ->
        if get_dt_name(fst hd) = id then
          true
        else
          check_globals id tl

    (* Translate a global *)
  in let translate_globals global =
    "public static final " ^ (string_of_data_type true (fst global)) ^ " = " ^ string_of_literal (snd
global) ^ "; \n"

  (* Translate a function with given env *)
  in let translate_helper fdecl =
```

```
(* This is a utility method for turning a map literal into a valid java expr
    The method it's self is in the backend code. *)
let into_map str =
  "plt.pbj.util.MapUtil.toMap(" ^ str ^ ")"

(* Returns the default java initialization for a data type. *)
in let rec default_init = function
  String(id) -> "\"\""
  | Array(id) -> "new ArrayList<Object>()"
  | Map(id) -> "new HashMap<Object, Object>()"
  | Long(id) -> "new Long(0L)"
  | Double(id) -> "new Double(0.0)"
  | Boolean(id) -> "new Boolean(false)"
  | _ -> raise(Failure "No default initialization for this data_type.")

in let get_dt_from_name name locals =
  try List.find (fun dt -> get_dt_name dt = name) locals
    with Not_found -> raise(Failure ("Variable " ^ name ^ " is undeclared."))

in let rec check_array_index locals = function
  Id(id) -> (match (get_dt_from_name id locals) with
      Long(id) -> true
      | _ -> raise (Failure ("Variable " ^ id ^ " is not valid index for array."))
    )
  | Literal(l) -> (match l with
      LongLiteral(s) -> true
      | _ -> raise (Failure ("Used invalid data type for array index."))
    )
  | MapGet(id, key) -> true
  | ArrayGet(id, idx) -> true
  | StmtExpr(e) -> (match e with
      ArrayPut(id, idx, e) -> check_array_index locals e
      | _ -> raise (Failure ("Used invalid expression for array index."))
    )
  | Size(id) -> true
  | _ -> raise (Failure ("Used invalid expression for array index."))

in let rec does_func_exist id = function
  [] -> false
  | hd :: tl ->
    if get_dt_name(hd.fname) = id then
      true
    else
```

```
    does_func_exist id tl

in let rec get_func_dt id functions =
  if does_func_exist id functions then
    (match functions with
      [] -> raise(Failure ("Failed to find func" ^ id))
      | hd :: tl ->
        if get_dt_name(hd.fname) = id then
          hd.fname
        else
          get_func_dt id tl
  ) else
    raise(Failure("Function " ^ id ^ " does not exist."))

in let is_null = function
  Null -> true
  | _ -> false

(* Checks for invalid assignments of data types *)
in let check_assign locals e dt no_raise =
  let rec match_string_dt = function
  String(s) -> true
  | _ -> false

  in let rec match_array_dt = function
    Array(s) -> true
    | _ -> if no_raise then false
      else raise (Failure ("Assigned array to invalid data type"))

  in let rec match_map_dt = function
    Map(s) -> true
    | _ -> if no_raise then false
      else raise (Failure ("Assigned array to invalid data type"))

  in let rec match_long_dt = function
    Long(s) -> true
    | _ -> if no_raise then false
      else raise (Failure ("Assigned long to invalid data type"))

  in let rec match_double_dt = function
    Double(s) -> true
    | _ -> if no_raise then false
      else raise (Failure ("Assigned double to invalid data type"))
```

```
in let rec match_boolean_dt = function
  Boolean(s) -> true
  | _ -> if no_raise then false
    else raise (Failure ("Assigned boolean to invalid data type"))

in let rec match_data_type match_func check_func dt = function
  FunctionCall(id,e) -> match_func (get_func_dt id functions)
  | MapPut(id, key, e) -> check_func e dt
  | ArrayPut(id, idx, e) -> check_func e dt
  | Assign(id, e) -> check_func e dt
  | JamSpread(f, sp) -> (match (f, sp) with
    (FunctionCall(id, e), Spread(FunctionCall(id2, e2)) ) ->
        match_func (get_func_dt id functions) && match_func (get_func_dt id2 functions)
    | (NoExpr, Spread(FunctionCall(fid, fe))) ->
      (match dt with
        Array(id) -> true
        | _ -> if no_raise then false else raise(Failure("Assigned jam to invalid type."))
      )
    | _ -> raise (Failure ("Improper Jam/Spread.")))
  | Spread(f)        -> (match f with
       FunctionCall(id, e) -> match_func (get_func_dt id functions)
     | _ -> raise (Failure ("Improper Spread.")))
  | NoExpr -> if no_raise then false else raise(Failure("Assigned " ^ get_dt_name dt ^ " "))

in let rec check_assign_helper e dt = match dt with
  (*  CHECK ASSIGN FOR STRING ************************************************)
  String(es) -> (match e with
    (* if it's an id get the data type from locals list *)
    Id(id) -> match_string_dt (get_dt_from_name id locals)
    | Literal(l) ->
      ( (* Check the assignment of a string to an id *)
      match l with
        StringLiteral(sl) -> true
        | _ ->  if no_raise then false else raise (Failure ("Assigned string to non-string literal."))
      )
    | Concat(e1, e2) -> true
    | MapGet(id, key) -> true
    | ArrayGet(id, idx) -> true
    | Null -> true
    | StmtExpr(e) -> match_data_type match_string_dt check_assign_helper dt e
    | _ ->  if no_raise then false else raise (Failure ("Assigned string to invalid expression."))
    )
  (* CHECK ASSIGN FOR ARRAY *************************************************)
  | Array(id) -> (match e with
```

```
  Id(id) -> match_array_dt (get_dt_from_name id locals)
  | ArrayLiteral(a) -> true
  | MapValues(id) -> true
  | MapKeys(id) -> true
  | MapGet(id, key) -> true
  | ArrayGet(id, idx) -> true
  | StmtExpr(e) -> match_data_type match_array_dt check_assign_helper dt e
  | _ -> if no_raise then false else raise (Failure ("Assigned array to invalid expression."))
  )
(*  CHECK ASSIGN FOR MAP ********************************************)
| Map(id) -> (match e with
  Id(id) -> match_map_dt (get_dt_from_name id locals)
  | MapLiteral(ml) -> true
  | MapGet(id, key) -> true
  | ArrayGet(id, idx) -> true
  | StmtExpr(e) -> match_data_type match_map_dt check_assign_helper dt e
  | _ -> if no_raise then false else raise (Failure "Asigned map to invalid expr.")
  )
(* CHECK ASSIGN FOR LONG ********************************************)
| Long(id) -> (match e with
  Id(id) -> match_long_dt (get_dt_from_name id locals)
  | Literal(l) -> (match l with
      LongLiteral(ll) -> true
      | _ -> if no_raise then false else raise (Failure "Assigned long to non-long literal")
    )
  | Binop (e1, o, e2) -> check_assign_helper e1 dt
  | Size(id) -> true
  | MapGet(id, key) -> true
  | ArrayGet(id, idx) -> true
  | StmtExpr(e) -> match_data_type match_long_dt check_assign_helper dt e
  | _ -> if no_raise then false else raise (Failure "Assigned long to invalid expression.")
  )
(* CHECK ASSIGN FOR DOUBLE ******************************************)
| Double(id) -> (match e with
  Id(id) -> match_double_dt (get_dt_from_name id locals)
  | Literal(l) -> (match l with
      DubLiteral(dl) -> true
      | LongLiteral(ll) -> true
      | _ ->  if no_raise then false else raise (Failure "Assigned double to invalid literal.")
    )
  | MapGet(id, key) -> true
  | ArrayGet(id, idx) -> true
  | Binop (e1, o, e2) -> check_assign_helper e1 dt
  | StmtExpr(e) -> match_data_type match_double_dt check_assign_helper dt e
```

```
        | _ -> if no_raise then false else raise (Failure "Assigned double to invalid expression.")
        )
      (* CHECK ASSIGN FOR BOOLEAN ****************************************)
      | Boolean(id) -> (match e with
        Id(id) -> match_boolean_dt (get_dt_from_name id locals)
        | Literal(l) -> (match l with
            BooleanLiteral(b) -> true
            | _ -> if no_raise then false else raise (Failure "Assigned boolean to non-boolean
literal.")
          )
        | MapGet(id, key) -> true
        | ArrayGet(id, idx) -> true
        | Binop (e1, o, e2) -> (match o with
          Seq -> true
          | Peq -> true
          | Greater -> true
          | Geq -> true
          | Less -> true
          | Leq -> true
          | And -> true
          | Or -> true
          | Neq -> true
          | _ -> false
          )
        | StmtExpr(e) -> match_data_type match_boolean_dt check_assign_helper dt e
        | _ -> if no_raise then false else raise (Failure "Assigned double to invalid expression.")
        )
      | _ -> if no_raise then false else raise (Failure "Invalid assignment")
    in check_assign_helper e dt

  in let is_map locals id =
    let rec is_map_helper = function
      Map(id) -> true
      | _ -> false
    in List.exists (fun dt -> get_dt_name dt = id && is_map_helper dt) locals

  in let is_array locals id =
    let rec is_array_helper = function
      Array(id) -> true
      | _ -> false
    in List.exists (fun dt -> get_dt_name dt = id && is_array_helper dt) locals

                    in let rec match_args_dt id e functions locals=
      if does_func_exist id functions then
```

```
      (match functions with
         [] -> raise(Failure ("Failed to find func with params" ^ id ))
         | hd :: tl ->
            let rec match_formals fli eli = (match (fli, eli) with
               | ([], []) -> true
               | (f::fl, e::el) -> if check_assign locals e f true then
                  match_formals fl el
                  else false
              | (_, _) -> false
         )in
            if (get_dt_name(hd.fname) = id && (match_formals hd.formals e)) then
               true
            else
               match_args_dt id e tl locals
  ) else
    raise(Failure("Function " ^ id ^ " does not exist with those parameters."))

(* Basic recursive function for evaluating expressions *)
in let rec string_of_stmt_expr locals = function
  Assign(s, e) ->
    (* Before we assign, ensure the assignment is valid *)
    let dt = List.find (fun dt -> get_dt_name dt = s) locals in
    s ^ "=" ^ "(" ^ (string_of_data_type false dt) ^ ")" ^ string_of_assignment locals dt e

  | ArrayPut(id, idx, e) ->
    if check_array_index locals idx then
              "plt.pbj.util.ArrayUtil.set(" ^ id ^ ","
              ^ string_of_expr locals idx ^ ", "
              ^ string_of_expr locals e ^ ")"
    else
      raise (Failure "Should have failed before here.")
  | MapPut(id, key, v) -> if is_map locals id then
      id ^ ".put(" ^ string_of_expr locals key ^ ", " ^ string_of_expr locals v ^ ")"
    else
      raise (Failure (id ^ " is not a valid map type."))
  | FunctionCall(s,e) -> if match_args_dt s e functions locals then
                                  s ^ "("  ^  String.concat "," (List.map (string_of_expr locals) e) ^ ")"
                                  else raise (Failure ("failed earlier"))
  | NoExpr -> ""
  | JamSpread(f, sp)   ->
                                  (* jam: jadd(@) spread: add(@myList); *)
    (* create a map from slave to job where job is*)
    (*pass the actuals*)
                                  (let print_acts list locals = (
```

```
          (if List.length list > 0 then "(Object)" else "")
        ^ (String.concat ", (Object)" (List.map (string_of_expr locals) list))
        )
                                  in let p_r_helper x f locals = (match x with
        | At(_)    -> "PBJOp.jam( \"" ^ (fst(f)) ^ "\", new Object[]{"
                                                ^ (print_acts (snd(f)) locals) ^ "})"
        | l        -> string_of_expr locals l)
                                  in let print_acts_returned actlist f locals = (
        (if List.length actlist > 0 then "(Object)" else "")
        ^ (String.concat ", (Object)" (List.map (fun x -> (p_r_helper x f locals)) actlist))
        )
    in (match (f, sp) with
                                  (*Spread(FunctionCall(fid, fe))*)
      (FunctionCall(jid, jargs), Spread(FunctionCall(fid, fe))) ->
                                          (*Object[] *)
                                                jid ^ "( " ^ (print_acts_returned jargs (fid, fe) locals)
^ ")"
      | (NoExpr, Spread(FunctionCall(fid, args))) ->
        "PBJOp.jam(\"" ^ fid ^ "\", new Object[]{ " ^ (print_acts args locals) ^ "})"
                                          | (_,_) -> raise (Failure("improper Jam Spread 2.")))
  | Spread(f)         ->
                                  (let print_acts list locals = (
          (if List.length list > 0 then "(Object)" else "")
        ^ (String.concat ", (Object)" (List.map (string_of_expr locals) list)))
    in (match f with
      FunctionCall(id, args) ->
              "PBJOp.spread(\"" ^ id ^ "\", new Object[]{ " ^ (print_acts args locals) ^ "})"
      | _ -> raise (Failure("Spread on non-function."))))

  and string_of_expr locals = function
    StmtExpr(e) -> string_of_stmt_expr locals e
    | Literal(l) -> string_of_literal l
    | Binop (e1, o, e2) ->
        let dt_long = Long("Temp1") in
        let dt_doub = Double("Temp2") in
        let dt_str = String("Temp3") in
        let dt_bool = Boolean("Temp4") in
        let dt_array = Array("Temp5") in
        let dt_map = Map("Temp6") in
        let check_binop_type locals =
          (* Expressions must be booleans for logical ops *)
          if (o = And || o = Or) then
            check_assign locals e1 dt_bool true && check_assign locals e2 dt_bool true
          (* All datatypes are java objects so expression can be any type for .equals() *)
```

```
            else if o = Seq then true
            (* Expression must be the same type for == *)
            else if o = Peq then
              (check_assign locals e1 dt_long true && check_assign locals e2 dt_long true) ||
              (check_assign locals e1 dt_doub true && check_assign locals e2 dt_doub true) ||
              (check_assign locals e1 dt_str true && check_assign locals e2 dt_str true) ||
              (check_assign locals e1 dt_bool true && check_assign locals e2 dt_bool true) ||
              (check_assign locals e1 dt_array true && check_assign locals e2 dt_array true) ||
              (check_assign locals e1 dt_map true && check_assign locals e2 dt_map true) ||
          (is_null e1) || (is_null e2)
            (* Expressions must be long or double for arith and comp ops *)
            else
              (check_assign locals e1 dt_long true || check_assign locals e1 dt_doub true)
                  && (check_assign locals e2 dt_long true || check_assign locals e2 dt_doub true)
           in let op_string =
            (match o with
              Add -> "+"
            | Sub -> "-"
            | Mult -> "*"
            | Div -> "/"
            | Mod -> "%"
            | Peq -> "=="
            | Greater -> ">"
            | Geq -> ">="
            | Less -> "<"
            | Leq -> "<="
            | And -> "&&"
            | Or -> "||"
        | _ -> "") in
            if check_binop_type locals then (match o with
        Seq -> "(" ^ string_of_expr locals e1 ^ ".equals(" ^ string_of_expr locals e2 ^ "))"
        | Neq -> "(!" ^ string_of_expr locals e1 ^ ".equals(" ^ string_of_expr locals e2 ^ "))"
        | Mod -> "(Long.valueOf(" ^ string_of_expr locals e1 ^ op_string ^ string_of_expr locals e2
^ "))"
        | _ -> "(" ^ string_of_expr locals e1 ^ op_string ^ string_of_expr locals e2 ^ ")"
        )
            else
              if (o = And || o = Or) then raise (Failure ("Invalid Type for operation " ^ op_string ^ ":
Both expressions must be type Boolean"))
              else if o = Peq then raise (Failure ("Invalid Type for operation " ^ op_string ^ ": Both
expressions must be the same type"))
              else raise (Failure ("Invalid Type for operation " ^ op_string ^ ": Both expressions must
be type Long or Double"))
      | MapLiteral(ml) -> into_map ("new Object[]{" ^
```

```
        String.concat "," (List.map (fun (d,e) -> string_of_literal d ^ "," ^ string_of_expr locals e)
ml) ^
        "}")
    | ArrayLiteral(a) ->
        let rec array_expr locals array = match array with
              []     -> []
            | e::a   -> string_of_expr locals e :: array_expr locals a
        in "new ArrayList<Object> (Arrays.asList(" ^ (String.concat ", " (List.rev (array_expr locals
a))) ^ "))"
    | Null              -> "null"
    | ArrayGet(id, idx)    ->
      if check_array_index locals idx then
        id ^ ".get(" ^ string_of_expr locals idx ^ ".intValue())"(**)
      else
        raise (Failure "Should have failed before here.")
    | MapGet(id, key) ->
      if is_map locals id then
        id ^ ".get(" ^ string_of_expr locals key ^ ")"
      else
        raise (Failure (id ^ " is not a valid map type."))
    | MapKeys(id) ->  if is_map locals id then
        "new ArrayList<Object>(" ^ id ^ ".keySet())"
      else
        raise (Failure (id ^ " is not a valid map type."))
    | MapValues(id) -> if is_map locals id then
        "new ArrayList<Object>(" ^ id ^ ".values()"
      else
        raise (Failure (id ^ " is not a valid map type."))
    | Size(id) -> if (is_map locals id || is_array locals id) then
        "new Long(" ^ id ^ ".size())"
      else
        raise (Failure (id ^ " is not a valid map or array."))
    | Concat(e1, e2) ->
      (* Start it off with an empty string so java knows to concat any numeric values vs addition.
*)
      "(\"\" + " ^ string_of_expr locals e1 ^ " + " ^ string_of_expr locals e2 ^ ")"
                          | At(e)             ->  "new Spreadable( " ^ string_of_expr locals e ^ ")"
    | Id(s) ->
      (* Ensures that the used id is within the current scope *)
      if(List.exists (fun dt -> get_dt_name dt = s) locals) then
        s
       else
      if (check_globals s globals) then
        s
```

```
    else
      raise (Failure ("Undeclared variable " ^ s))
and string_of_assignment locals dt e =
  if check_assign locals e dt false then
     (match dt with
      Long(s) -> "Long.valueOf(\"\" + (" ^ string_of_expr locals e ^ "))"
      | Double(s) -> "Double.valueOf(\"\" + (" ^ string_of_expr locals e ^ "))"
      | String(s) -> "\"\" + (" ^ string_of_expr locals e ^ ")"
      | _ -> string_of_expr locals e
    )
  else
    raise (Failure ("Failed check assign."))

in let check_valid_for_stmt = function
 ExprAsStmt(stmt_of_expr) -> true
 | DeclareAssign(dt, e) -> true
 | Declare(dt) -> true
 | NoStmt -> true
 | _ -> false

in let rec string_of_stmt (output, locals) = function
  Block(string_of_stmts) ->
    let l = List.fold_left string_of_stmt ("", locals) string_of_stmts
    in (output ^ "{\n" ^ (fst l) ^ "}\n", locals)
  | Print(s) -> (output ^ "System.out.println(" ^ string_of_expr locals s ^ ");\n", locals)
                          | If (p, t, Block([])) ->
                                  (output
        ^ "if(" ^ string_of_expr locals p ^ ") "
        ^ fst (((fun x -> string_of_stmt ("", locals) x) t)), locals)
                          | If (p, t, f) ->
    (output
        ^ "if(" ^ string_of_expr locals p ^ ") "
        ^ (fst (string_of_stmt ("", locals) t))
        ^ "\n else " ^ (fst (string_of_stmt ("", locals) f)), locals )
  | For (s1, e, se, b) ->
    let output_pair = (string_of_stmt ("", locals) s1)
    in let init_locals = (snd output_pair)
    in if (check_valid_for_stmt s1) then
     if (check_assign init_locals e (Boolean("")) true) then
        (output
                                      ^ "for("
                                                ^ (fst output_pair)
            ^ string_of_expr init_locals e ^ "; "
            ^ string_of_stmt_expr init_locals se ^ ") "
```

```
                                                    ^ (fst (string_of_stmt ("", init_locals) b )), locals)
        else raise (Failure ("For condition must return a boolean expression."))
      else raise (Failure ("For initialization must be a valid statment."))
    | While (e, b) ->
                                    (output
        ^ "while(" ^ string_of_expr locals e ^ ") "
        ^ (fst (string_of_stmt ("", locals) b)), locals )
    | DoWhile (b, e) ->
          (output
                                                ^ "do\n" ^ (fst (string_of_stmt ("", locals) b))
        ^ "while(" ^ string_of_expr locals e ^ "); "
        , locals )
    | Return(e) ->
      if (check_assign locals e fdecl.fname true || is_null e) then
      ( output ^ "return " ^ "("^ string_of_data_type false fdecl.fname ^ ")" ^ string_of_expr locals
e ^ ";\n", locals)
      else
        raise (Failure ("Invalid return expression for function: " ^ (get_dt_name fdecl.fname)))
    | ExprAsStmt(e) -> (output ^ string_of_stmt_expr locals e ^ ";\n", locals)
    | Declare(dt) ->
      let name = get_dt_name dt in
      if List.exists (fun dt -> get_dt_name dt = name) locals then
        raise (Failure ("Variable " ^ name ^ " has already been declared."))
      else
        (output ^ (string_of_data_type true dt) ^ " = " ^ default_init dt ^ ";\n", dt :: locals)
    | DeclareAssign(dt, e) ->
      let name = get_dt_name dt in
      if List.exists (fun dt -> get_dt_name dt = name) locals then
        raise (Failure ("Variable " ^ name ^ " has already been declared."))
      else
        if check_assign locals e dt false then
          (output ^ (string_of_data_type true dt) ^ " = " ^
            "(" ^ (string_of_data_type false dt) ^ ")" ^
            (string_of_assignment locals dt e) ^ ";\n", dt :: locals)
        else
          raise (Failure ("Failed check assign on declare assign."))
    | NoStmt -> (output ^ ";", locals)

  in "public static " ^
  (string_of_data_type true fdecl.fname) ^ "(" ^ String.concat ", " (List.map (string_of_data_type
true) fdecl.formals) ^ ")"
    ^ fst (string_of_stmt ("", fdecl.formals) (Block fdecl.body))
 in let func_is_void = function
  Void(s) -> true
```

```
  | _ -> false

 in let rec check_all_are_true = function
  [] -> true
  | hd :: tl ->
    if hd then
      check_all_are_true tl
    else
      false

 in let check_return_statements func =
  let rec check_return_helper last_stmt is_outer = function
    [] -> (match last_stmt with
        Return(e) -> true
        | _ -> if ((not is_outer) || func_is_void func.fname) then true
          else false
      )
    | hd :: tl -> (match hd with
        Block(stmts) ->
          if check_return_helper hd false stmts then
            check_return_helper hd is_outer tl
          else
            false
        | _ -> check_return_helper hd is_outer tl
      )
 in check_return_helper (Block []) true (func.body)

 (* The next line is the heart of it ans is where this all really starts *)
 in if check_all_are_true (List.map check_return_statements functions) then
  "package plt.pbj;\n" ^ imports ^ "public class PBJRunner {\n" ^
  String.concat "" (List.map (translate_globals) globals) ^
  String.concat "" (List.map (translate_helper) functions) ^
  "}\n"
 else
  raise(Failure("Functions had invalid returns."))
```

Makefile

```
pbjc : ast.cmo parser.cmi parser.cmo scanner.cmo compile.cmo pbjc.cmo
   ocamlc -o pbjc ast.cmo parser.cmo scanner.cmo compile.cmo pbjc.cmo

pbjc.cmo : compile.ml parser.ml scanner.ml pbjc.ml
   ocamlc -c pbjc.ml
```

```
parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

scanner.ml : scanner.mll
	ocamllex scanner.mll

%.cmi : %.mli
	ocamlc -c $<

%.cmo : %.ml
	ocamlc -c $<

.PHONY : clean
clean :
	rm -f pbjc scanner.ml *.cmi *.cmo parser.ml parser.mli parser.output scanner.ml
```

package.sh

```
#!/bin/sh

./compiler/pbjc -c $1 > backend/src/plt/pbj/PBJRunner.java

cd backend # into backend
./build.sh # compile backend.

cp -r bin/* ../distro

cd .. # back to root

cd distro
jar cmvf MANIFEST.MF PBJ.jar com plt

mv PBJ.jar ../
```