Cardigan

Joshua Lopez * jl3497 -- Project Leader
Muzi Gao * mg3194 | Miriam Melnick * mrm2198

1. Introduction

1.1 What is Cardigan

Cardigan is a language which can be used to define, develop, test, and play card games. Many existing languages already contain several of the data types and control structures necessary to implement simple card games, but because the languages were not specifically developed for this purpose, some setup is usually required. Cardigan seeks to provide developers with tools specifically suited for creating and deploying these games with minimal additional coding. This is accomplished by providing a small set of powerful yet flexible object types, as well as an easy to manage function declaration system. Card games can be easily implemented by simply defining sets of objects (cards, players) and functions (turns, card actions, rules). Cardigan should not be confused with a framework, which only allows users to fill in sets of customized data, but provides little control over how the game is played. A game defined in Cardigan may define any type of rules and gameplay, and may modify these elements during the game to any extent. A turn structure may contain as many actions as desired. A game may even be implemented with no turn structure at all. Cardigan also allows developers to create computerized opponents powered by algorithms which they can define in the language.

1.2 Problems Which Cardigan Can Solve

Developing card games is an extremely time consuming process usually requiring multiple iterations of text and artwork, numerous modifications to existing cards, tedious organization of numerous permutations of cards, trimming decks to size, and a fair amount cutting and pasting (with actual scissors and paste). Considering most games contain several copies of the same card, the entire process must be repeated several times per deck. Production of the product requires additional investments of time and money, as well as other overheads associated with a physical inventory.

Cardigan seeks to minimize all of these problems simultaneously by moving the entire development and production process into a single programming language. Each completed program represents a game, and any modifications to the game's rules, cards, players, and other factors require changing only a few lines of code.

The nature of programs also facilitates sharing between developers in a way that is not possible with physical cards. A Cardigan developer could access previously written code which implements a particularly popular aspect of many games, and incorporate it into their project. Any developer who choses to make their code publically available could open their games to further modifications and ‘house rules’.

2. Tutorial

Programs in Cardigan, at the highest level, consist of assignments and function definitions.

2.1 Scope

2.1.1 The Global Scope

A program begins in the global scope. Any assignments made in this scope are immutable and visible to the entire program unless they are hidden by a function (see below). Functions may only be defined in the global scope and are visible to the entire program.

2.1.2 Function Scope

Cardigan only allows assignment and function definition in the global scope, any other functionality must be defined within the body of a function. Inside a function, assignments are mutable and several other statements are permitted. In addition, variables declared within a function body are mutable.

2.2 Declaration Statements

2.2.1 Function Declaration

Declaring a function requires four parts.

1. An identifier for the function. Any valid identifier which has not been previously used.
2. An argument list. The list must be enclosed with parentheses and comma separated. The argument list is required for all functions, but it may be empty.
3. A single equals sign
4. The body of the function. Function bodies must be enclosed in curly braces. A single newline character is optionally allowed after the opening curly brace. The body of a function may be made up of any number of statements.

Functions may only be declared in the global scope, but once they have been declared, their values may be included in struct definitions.

Any valid identifiers are allowed in the argument list but if they are identical to identifiers in global scope, the global variables will be hidden in the body of the function. See below for more information on variable hiding.

Statements in the body of a function are evaluated sequentially unless the statements transfer control flow to another function or a different part of the current function.

Functions may include return statements. The synax of a return statement is the “return” keyword followed by some expression. If the return statement is encountered the value of the expression is evaluated, execution of the function then terminates, and the return value is returned by the function. There is no limit to the number of return statements a function can have, but all of their expressions must evaluate to the same

type. If no return statement is encountered, the function will return boolean false.

All Cardigan programs must include a single function called PLAY, which takes no arguments. After evaluating all global variable assignments, execution will begin at the top of the PLAY function and the program will terminate when the PLAY function finishes executing.

2.2.2 Global values

The declaration statement in Cardigan consists of an identifier, an equal sign, and a value to assign to the identifier. A simple case is

```
x = 12
```

After this statement executes, x will have the value of 12. Notice there are no type declarations. Cardigan is strongly typed, but types are determined automatically by the compiler so no keyword is necessary.

If this assignment occurred in the global scope, the value would be immutable. Changes are not permitted to immutable values so the following code would raise an error during compilation.

```
x = 10
x = 120
```

2.2.3 Local values

If the assignment occurs within the body of a function, the value is a local value. Local values are mutable, but any subsequent assignments must have the same type.

```
a() = {
        x = 10          // Declaration within a function. x now has the type int
        x = 20          // OK: value has the same type
        x = “uh oh”     // ERROR -- We cannot assign a string value to an int variable
}
```

Local variables are only visible in the scope in which they are declared, but they may be passed to functions as arguments if they need to be accessible from another scope.

If a local variable has the same identifier as a global variable, the global variable will be hidden by the local variable and any use of the identifier will refer to the local variable. If the global value is needed in the same scope, it should be assigned to a new identifier first. Note that this new identifier will be mutable, but any changes to it will not affect the

global variable.

```
x = 10
a() = {
        b = x           // b now has the same value as x
        x = 20          // we have declared a new variable called x. The global x is no
                        // longer visible from this scope, but we can still use it's value via b.
        b = 30          // We have changed b, but the global x has not changed.
}
```

2.3 Types

Cardigan has seven built in types:

2.3.1 int

ints represent integers. only digits are allowed in ints

```
x = 3
```

2.3.2 float

floats represnt floating point values. Floats must contain a decimal point

```
x = 3.2
```

2.3.3 bool

boolean true or false
the keywords for these values are the words “true” and “false”

```
y = false
```

2.3.4 string

character strings

```
foo = “bar”
```

3.3.5 collections

a structure which may be used as an array, a queue, a stack, or a linked list. Collections are contained within square brackets and their entries must be comma separated. The elements of a collection may be of any type. There are two ways to declare a collection. Either by defining all of its members

```
oneToTen = [1, 2, 3, 4, 5, 6, 7, 8, 9. 10]
```

or by defining a range using a colon

oneToTen = [1:10]

2.3.6 struct

a struct is Cardigan's version of a hash table. Keys may be defined using identifiers. The string representation of the identifier must be used to access the property of the struct using dot lookup. Structs are defined with curly braces, each key/value pair must be separated by a colon, and each entry must be separated by a comma. The entries in a struct may be of any type. If the value of an entry in a struct is the name of a function, the struct property will become an alias to the function.

```
myStruct = {info: “yes”, info2: 5, info3: false}
OUTPUT(myStruct.”info”)
```

would output

yes

2.3.7 enum

An enum is a collection of names which may be used in later comparisons. The body of an enum must begin and end with a pipe “|” character, and the values inside must be comma separated. Any valid identifier may be used as a property of the enum and they are accessed using dot lookup with the proprety identifier. Enums are immutable and their values may only be used in comparisons.

```
x = |sun, moon|

thing = x.moon

x.sun == x.moon  // this would be false

thing == x.moon   // this would be true
```

2.4 Control Structures

To actually create a game in Cardigan we need to define how to play it. To do this we use control structures, some familiar and some novel

2.4.1 Blocks

Blocks are not valid Cardigan code on their own, but they make up a crucial part of all the control structures of the language. A block is a set of statements contained within curly braces.

2.4.2 if elseif else

If statements can be created using the “if” keyword, followed by a predicate, and then a block of code. The predicate may be any expression which evaluates to a boolean value and the code contained in the block will be executed if the predicate evaluates to true. Optionally, any number of elseif blocks may be added after the “if” structure. These follow the same syntax as the “if” statement, keyword-predicate-block, except that the elseif keyword must be used instead. An if structure may include a single else structure which come last. The else structure take no predicate, just the “else” keyword and a block. Control begins at the if block, and evaluates each predicate, executing the block associated with the first which evaluates to true and ignoring all subsequent elseif/else blocks. If no predicate evaluates to true, the else block (if included) is executed.

```
if isTrue {
        doSomething()
}
elseif otherThingIsTrue {
        doSomethignElse()
}
else {
        makeSandwich()
}
```

2.4.3 for

For loops follow the structure of “for” keyword local-identifier colon collection block. The code in the block is evaluated once for each value in the collection, with that value assigned to the local-identifier.

```
x = [“one”, “two”, “three”, “four”]
for y:x{
        OUTPUT(y)
}
would print:

one
two
three
```

four

2.4.4. while

While loops are structured identically to the if/elseif blocks, but use the while keyword and may only include one block. The code in the block will continue to execute until the predicate evaluates to false.

```
stoppingCondition = true
while stoppingCondition{
        OUTPUT(“Still going”)
        ...
}
```

If no code exists within the body of the block which modifies the value of the predicate, you may have an infinite loop.

2.4.5 rules

Rules structures are a way to define several blocks in a single structure and let Cardigan figure out which one should be used. Rules support local assignments which will remain in scope for the body of the rules structure. Within the rules block are several predicate & block pairs (with no code separating). Any predicate which evaluates to true within the rules structure will be executed. The syntax for the rules structure is the “rules” keyword, a parenthesis enclosed set of local assignments, and a set of predicate block pairs

```
t = [1, 3, 5]
rules (x = t.length()){
        x > 10 {
                OUTPUT(“Woah that’s a lot”)
        }
        x < 2{
                OUTPUT(“Only one left”)
        }
        x == 5{
                OUTPUT(“five!”)
        }
}
```

2.5 Built in Functions

There are four built in functions to cardigan, all written in all caps.

2.5.1 INPUT
The input function pauses execution of the program and waits for user input. The argument to the INPUT function tells Cardigan what type of input to expect from the user. The following keyword are allowed; int, float, bool, string.

2.5.2 OUTPUT
Output prints to the terminal. Output accepts one argument which must be a string. The specified string will be printed

2.5.3 CAST
Cast converts from one type to another. See section 3.7.3 for a listing of which types of casts are allowed

2.5.4 PLAY
PLAY is the entrypoint for the program after the global variables are defined. Play is not predefined by the language, but must be included in the .crd script. This is analogous to the main function in C/C++ and Java. Play may not be define with any arguments

3. Manual

3.1. Introduction
This document describes the Cardigan language. There are seven sections to this manual:
3.1. This introduction
3.2. Tokens -- a description of each class of tokens
3.3. Types -- a brief description of each primitive and derived data type, including syntax
3.4. Scoping -- information regarding scoping of variables
3.5. Statements -- the structure of a line, or lines, of code in a Cardigan program
3.6. Expressions -- parts of statements which need to be evaluated during execution
3.7. Writing programs -- Correct program structure and a list of built in functions.

3.2. Tokens & separators
There are 8 classes of tokens: identifiers, keywords, boolean constants, integer constants, float constants, string literals, operators, and separators. Tab and space characters are considered whitespace, and any combination of these are treated as a single whitespace character. Two

consecutive tokens in a single statement must be separated by whitespace unless one of the two is either an operator or a separator, or they are the last and first tokens of two consecutive statements. Comments are never tokenized and are ignored by the compiler.

3.2.1 Comments & Separators

Comments and token separators are not tokenized, but are instead, ignored by the compiler. Comments are single-line and and begin with double slashes. Any characters between // and the end of line would be ignored.
For example, a valid comment would be like:

```
// Here is a comment
```

Token separators consist of any combination of one or more tab and space characters. These are not treated as tokens, but are used to separate tokens from each other. The compiler can make distinction between operator tokens and other types of tokens so operators and their operands need not be separated by spaces. Grouping separators are distinguishable from every type of token, including other grouping separators, so they never need to be separated from other tokens. Although it is not required, a program can include any number of token separators before or after an operator.

3.2.2 Separators

There are three types of separators: token separators, line separators, and grouping separators.

3.2.2.1 Newline separator

The newline character is used as a line separator. All of the tokens on a single line must comprise a single statement to be syntactically valid. Multi line statements may be created by ending each line with a backslash character. These will be interpreted as a single line, and therefore, a single statement.

3.2.2.2 Grouping separator

Grouping separators are used to create groups of tokens. The grouping separators are: left and right parenthesis, left and right square bracket, left and right curly brace, the pipe character, commas, and colons. Their uses are discussed in their relevant sections.

3.2.3 Identifiers

An identifier is a sequence of letters, digits, and underscores(_). The first character must be a letter or underscore. Identifiers are case sensitive, which means “foo” and “Foo” are different.

3.2.4 Keywords

Keywords are simple terms that are used throughout the program for purposes specified by its definition. Unlike identifiers or constants, keywords cannot be replaced by random names or values.
The keywords defined here are generally used in many languages, but a few (rules, Play) are given different names, terms related to card games.

int, float, string, bool	(TYPE DECLARATIONS)
if, elseif, else, while, for, ?	(CONTROL STRUCTURES)
true, false	(BOOLEAN CONSTANTS)
return	(FUNCTION EVALUATION)
CAST, PLAY, INPUT, OUTPUT	(BUILT-IN FUNCTIONS)

3.3. Types

Cardigan is a strongly statically typed language. Identifier types are determined at declaration. Cardigan has four following primitive types: bool, int, float, and string, and three derived types: structs, collections, and enums.

3.1 Primitive types

3.3.1.1 Boolean Constants

A boolean constant represents either boolean true or boolean false, represented by the `true` and `false` keywords respectively.

3.3.1.2 Integer Constants

An integer constant is a sequence of one or more digits without a decimal point. The range of integers is 0 to 2_{31}-1. Octal or hexadecimal representations of integers are not supported.
For example, 42 is a valid integer constant, but 4.2 is not.

3.3.1.3 Float Constants

A float constant consists an integer part, and a decimal part. The integer part is a sequence of one or more digits. The decimal part is a decimal point followed by zero, one or more digits. The interpretation of floating-point literals that fall outside the range of representable floating-point values is undefined.
For example, 42.7, 42. are both valid float constants, but 42 would be recognized as an integer constant.

3.3.1.4 String literals

A string is a sequence of printable characters. A string must be surrounded by double-quote characters (`"`). Special characters must be escaped using the backslash character. The following special characters are permitted:

`\n` newline
`\t` tab
`\"` double quote
`\\` backslash

3.3.2 Derived Types

Derived types include structs, collections, enums, and functions.

Structs are objects that divide other objects into fields. Structs can be used to simulate the role of hash tables or classes.
Collections are objects similar to lists, and can even be modeled as arrays, stacks, queues, and linked lists. They are usually used to create lists of game elements.

Enums are used to declare a pseudo-type, with definite values that are immutable, and are used mainly for comparisons.

3.3.2.1 Enums

An enum is an immutable structure which allows programmers to create sets of values which may be used in comparisons. The values of an enum are meant to be used as their own data type exclusive to their particular enum. Enum values may be used in equality and inequality comparisons, but they may not be the operands of any arithmetic operations or greater than/ less than comparisons.

An enum is declared using the following syntax: identifier = | identifiers |
The first identifier will be used as the name of the enum. The identifiers between the vertical bars must be a comma separated list of one or more identifiers. These become the values of the enum. Accessing the value of the enum is done through dot access. The syntax for enum dot access is:
identifier.identifier
where the first identifier is the name of the enum and the second is the name of a value of the enum.

All enums are treated as the same type so comparisons across different enums is allowed, but will always be false.

```
a = |w, x|
b = |y, z|

a.w == b.y       // This is allowed and will always be false.
```

3.3.2.2 Collections

A collection is a list of values. Each term in a collection may be any type of expressions that can be defined in the program. Collections may also contain other collections.

Collections in Cardigan, are created using the syntax:
identifier = [values]
where values is a comma separated list of zero or more expressions. These expressions will be stored in the collection.

Elements of the collection are accessed with the syntax:

identifier[index]
where identifier refers to the collection, and index is an integer expression. This syntax allows you to access any element of the collection.

Collections also support several other access methods which allow them to be used as a linked list, stack, queue. Each of these may be called using the dot operator.

append(x) - Adds x to the end of the collection.
push(x) -- Adds x to the head of the collection.
pop() - Removes the first element of the collection and returns it.
peek() -- Returns but does not remove the first element of the collection
popLast() -- Removes and returns the last element of the collection.
shuffle() -- Randomizes the elements in the collection

For example the following code:

```
a = []
a.append(1)
a.append (2)
```

would result in the variable a having a value that is a collection which looks like

[1, 2]

3.3.3.3 Struct

Structs are Cardigan's version of a hash table. Structs can contain any number of properties, values of any Cardigan type including functions. Structs are declared with curly braces and may contain zero or more comma separated assignments. Within a struct an assignment is written as:

identifier: propertyName

Identifier must be a valid identifier as described above. propertyName must be a string which matches the name of one of the struct's properties.. The value of the expression will be stored in the struct under the specified identifier, and may be accessed later using dot notation. If the expression is an identifier to a function, the property of the struct will become an alias to the function and may be called like a function.

3.4. Scoping

3.4.1 Global Scope

In Cardigan, a complete program should be written in one file, and will be compiled at the same time.

Within the program may be several code blocks (see section 5.4 for more information on blocks), each with its own internal scope. At the highest level, outside of any blocks or function definitions, is the global scope. If a variable is declared in the global scope, it is visible to the entire program, but may not be modified. If a program attempts to modify the value of a previously defined global value while still in the global scope, the compiler will raise an error.

3.4.2 Local Scope

Each block of code (surrounded by curly braces) had its own local scope. Variables defined within a block will be available in the block's local scope, as well as in the body of any internal blocks. When the variable is available is referred to as "in scope", the internal scope of any internal block is called a "nested scope". When a variable is longer in scope, and any attempts to access it will result in a compiler error.

If the value of a variable is modified while it is in scope it will retain the new value until it is either modified again, or passes out of scope. This is true even if the variable is modified in a nested scope.

3.4.3 Masking & Copying

Although global variables may not be modified, it is still possible to re-use their names for variables declared in a local scope. When a variable is declared in a local scope, with the same name as a global variable, the local variable will mask the global variable, making it unavailable for as long as the local variable is in scope.

If a local variable is defined to have the value of a global variable, any subsequent modifications to the local variable will not affect the global variable.

3.4.4 Function Scope

Functions may only be declared at the global scope. Once a function is defined it may be called anywhere in the program.

3.4.5 Recursive Functions

Cardigan supports recursive function calls without any additional code. If a function is called within its own body, or from the body of a function which precedes it in the program, Cardigan will keep track of the call and as long as the function is defined later in the program, the compiler will not raise an error. If a call is made to a function which is never defined, the compiler will raise an error.

3.5. Statements

A statement is a complete line of code. There are several valid forms for a statement. A statement can be any valid expression (see below for details), a variable assignment, a control structure (one of: if, while, for, or ?), or a function definition.

3.5.1 Assignments

Assignment statements are used to assign a value to a variable. The syntax for an assignment statement is:

identifier = expression

If the identifier references a variable that has not been assigned previously, the expression may be any expression of any type. If the variable has already been declared, the expression must evaluate to the same type that the variable had before the assignment statement. Note that function definitions have a similar syntax, but are not the same.

3.5.2 Increments

Increment statements are unary operations that do not return a value. The syntax of increment statements is:

identifier operator

Any identifier with an integer or float value, including an element of a collection or a property of a struct may be used. The operators are `++` and `--`. The increment operator `++` adds the value of 1 (for integer operands) or 1.0 (for float operands) to the value of the specified identifier. The decrement operator `--` subtracts the value of 1 (for integer operands) and 1.0 (for float operands) from the value of the specified identifier.

3.5.3 Update Expressions

Update statements are binary operations which have the form: identifier operator operand. The identifier may be a variable, an entry in a collection, or a property of a struct provided the type of the identifier's value is either an int, a float, or a string. The operand may be any expression that evaluates to a value of type int or float if the identifier's value is of type int or float identifier, or of type string if the identifier's value is of type string. There are four update operators: `+=`, `-=`, `*=`, and `/=`. In the case of int and/or float identifiers and operands the `+=` operator adds the value of the identifier and the operand and replaces the value of the identifier with the result of the addition. The `-=`, `*=`, and `/=` operators work similarly, but perform subtraction, multiplication, or division respectively between the identifier and operand, replacing the identifier's value with the result of the operation.

If the identifier and operand are of type string, only the `+=` operator may be used. The operand string will be concatenated to the end of the string currently associated with the identifier, and the result will replace the value of the identifier.

3.5.4 Code Blocks

A code block is a set of curly braces surrounding a set of newline-delimited statements. The opening and closing curly

3.5.5 Control Structures

Cardian supports four types of control structures: `if`/`elseif`/`else`, `while`, `for`, rules

3.5.5.1 if

In an `if` block, the `if` keyword must be followed by a predicate, which must evaluate to either `true` or `false`, and then a block of code. If the predicate evaluates to `true`, the block of code is executed and control leaves the `if` structure. This can then be followed by 0 or more `elseif` statements, each of which has its own predicate and code block, and a single optional `else` block which has only a block and no predicate. In an `if` structure without an `else` block, at most one code block is executed. In an `if` structure with an `else` block, exactly one code block is executed.

3.5.5.2 while

A `while` statement is made up of the `while` keyword, predicate, and code block. The predicate is evaluated and if its value is `true`, the code block is executed and the predicate is re-evaluated. The code block will be run once for every time the predicate evaluates to `true`. When the predicate evaluates to `false`, control passes out of the `while` structure.

3.5.5.3 for

The `for` statement is used to iterate over items in a collection. The syntax of a `for` loop is: the `for` keyword, the name to use for the local variable in the code block, a colon, the name of the collection, and the code block to execute. The code is run once with each item in the collection. The item is passed to the code block using the specified variable name, and the code is run. The `for` loop's code block will iterate once for each item in the collection, assigning each object's value to the variable in scope sequentially. The objects will be treated as the same type as those in the collection.

3.5.5.4 rules

The rules statement is used to check several (potentially related) conditionals. It begins with the rules keyword, followed by an optional set of parentheses that contain a comma-delimited list of local variable assignments, and then a brace-delimited block. The block can contain 0 or more conditional statements, each consisting of a predicate followed by a code block. When control reaches the structure, the specified variables are bound and each conditional is evaluated in sequence. Any predicate which evaluates to true will have its associated code block executed. When control reaches the end of the brace-delimited block, the local bindings defined at its beginning go out of scope and control passes out of the rules structure.

3.5.6 Function Declaration

Functions may be declared at the top level by entering an identifier to be used as the name of the function, a parenthesis enclosed comma separated list of zero or more identifiers to be used as arguments, an equals sign, and a curly brace enclosed code block to be used as the body of

the function.

Any valid identifier may be used as the name of the function as long as it is not already bound to another type. Once the function is declared, the identifier is bound to the function type and no value of any other type may be assigned to it.

The argument list is mandatory, but it may be empty. If arguments are included, any valid identifier may be used for the name of the arguments. These will only exist within the scope of the function body, and will not be accessible once the function ends. The values of the arguments may be of any type, but once the function is called, each argument's identifier is bound to the type it was called with until the function ends. Any assignments to these identifiers in the body of the function must be of the same type each was given when the function is called.

The body of the function must be a valid code block as described above. In addition the the statements allowed inside any other code block, the body of a function may also contain the return statement. This statement consists of the word "return" followed by an expression. When control reaches this return statement, the expression will be evaluated, but no further statements in the function body will be executed. Instead, the value of the expression will be returned by the function. If control reaches the end of the function body without encountering a return statement, the value of boolean false will be returned instead.

3.6. Expressions

An expression could be either the right side of an assignment, or the entire statement. Valid types include direct expressions, unary expressions, binary arithmetic expressions, logical expressions, binary string operation, update and increment expressions, collection reference, enum reference, struct reference and function call. The explanation of each type will be described in following subsections.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

3.6.1 Operator Precedence

When expressions are evaluated, operators will be applied in the following order (listed from highest to lowest precedence).

`()`	grouping
`f(args)`	function calls
`x[index]`	indexing
`x.attribute`	referencing
`-`	unary arithmetic negation
`* / %`	multiplication, division, modulo
`+ -`	addition, subtraction
`== != < > <= >=`	comparison

`!`	logical negation
`and`	logical and
`or`	logical or

3.6.2 Direct Expressions

The following are all considered to be direct expressions:
identifiers: an identifier refers to a variable or a function. See the section of identifiers.
constants: a constant with primitive types (integer, float, or boolean), or derived types (enum, struct, or collection) is a direct expression.
strings: See the section on strings.

3.6.3 Unary Expressions

There are two unary expressions in Cardigan.
One is minus "`-`", which is used with a single numeric-valued operand such as an integer constant and a float constant. This operator forces its numeric-valued operand to be negative, and the result is of numeric type with value that equals to the negative of the numeric operand.
The other one is not "`!`", which is used with a single boolean constant or a boolean variable. This operator forces its single operand to be negative, and the result is of boolean type with value that equals to the negative of the operand.
Unary expressions are grouped right-to-left.

3.6.4 Binary Arithmetic Expressions

There are five different binary arithmetic operations: `+`, `-`, `*`, `/`, `%`. The modulo operator, `%`, is only defined when both arguments are integers. The result is the first modulo the second.
`+`, `-`, `*`, `/` are defined for any combination of integers and floats. The results are the standard arithmetic sum, difference, product, and quotient of the two numbers in question. `+` and `-` are commutative but the others are not. If both arguments are integers, then the operator will return an integer. If either or both is a float, then the operator will return a float.

3.6.5 Logical Expressions

There are four types of logical expressions: unary boolean expressions, binary boolean expressions, direct boolean expressions, and comparison expressions. Each type evaluates to a boolean value.

3.6.5.1 Unary boolean expressions

Unary boolean expressions involve the negation operator (`!`) followed by another boolean expressions. The result of this type of expression is the opposite boolean value of the expression directly following the negation operator. The ! operator has higher precedence than the other boolean operators (`and`, `or`), but lower precedence than the equality and comparison operators.

3.6.5.2 Binary boolean expressions

Binary boolean expressions have the following form:

operand operator operand

where the operands can be any boolean expression and the operator is either the `and` operator or the `or` operator. Operations involving the `and` operator return `true` if both operands have the value `true` and `false` otherwise. The `or` operator returns `true` if one or both of its operands have the value `true` and `false` if both have the value `false`. The `and` operator has higher precedence than the `or` operator, and the equality comparison has higher precedence than both the `and` and `or` operators.

3.6.5.3 Comparison expressions

Comparison expressions have the form:

operand operator operand.

The operands may be of any primitive type, an identifier which has the value of a primitive type, or an expression which evaluates to a primitive type. For the purposes of comparisons, enum values are treated as ints. All types allow use of the equality and inequality operators, `==` and `!=` respectively. Integer and float values also support the use of the following operators, `<=`, `>=`, `>`, `<`. In all cases the operands must have the same type, except for comparisons between integers and floats, which are allowed to be compared to each other. The behavior of the operators with each type is described below.

Booleans -- When the operands are booleans, the `==` operator functions like an XNOR comparison, returning true if both values are the same, and false if they are different. The "!=" operator functions as an XOR comparison, returning true if the operands have different values, and false if their values are the same.

Ints -- The equality operator "==" returns true if the operands have the same value and false if they don't. The inequality operator "!=" returns true if the operands do not have the same value, and false if they do. The strictly less operator "<" returns true if the first operand is less than the second and false if they are equal, or if the first is greater. The strictly greater operator ">" returns true if the first operand is greater, and false if they're equal or the first is less. The less or equal operator "<=" functions the same as the strictly less operator except that it evaluates to true if the values are equal. The greater or equal operator ">=" functions the same as the strictly greater operator except that it evaluates to true if the values are equal.

Floats -- The rules for float comparisons are the same as integer comparisons. Comparisons between a float and an integer are also allowed; the int is treated as a float with the same value as the integer and a 0 fraction value.

Strings -- The “==” operator performs a character-wise comparison of the strings and returns true if they are identical or false otherwise. The “!=” performs the same comparison and returns true if they are not identical, and false if they are.

3.6.5.4 String concatenation

String concatenation is performed using the + operator. The syntax is:

string + string

where each string may be a string literal or a string variable. The result of the expression will be a single string which is the concatenation of the first and second string operands.

3.6.5.5 Collection References

Collections are indexed using the [ ] separators and may be used to access any element or value in the collection using the integer mentioned inside the [ ] separatora. The syntax is:

my_col[N]

where my_col is the collection and N is an integer which gives you the Nth value of the collection.

3.6.5.6 Struct and Enum References

The element of an enum can be accessed by an identifier or an enum object, followed by .(dot), and another identifier. Here, the first identifier refers to a user-defined enum name, and the second one refers to the name of the element.

Similarly, the property of a struct can be accessed by an identifier or a struct object, followed by .(dot) and another identifier. Here, the first identifier refers to a user-defined struct name, and the second one refers to the name of the property.

3.6.5.7 Function Calls

To call a function and execute the statements inside the body of the function, the following syntax must be used: identifier(expressions), where the identifier is defined as a function in the program. The expressions is a comma separated list of zero or more expressions whose values will be assigned to the parameters of the function. The number of expressions in the function call must match the number of parameters in the function definition. When the function is called, control will jump to the first line of the function and continue until a return statement is evaluated, or the end of the function body is encountered. Control will then return to just after the point where the function call was made. If a return statement was encountered, the function call will be evaluated to have the value of the expression in the return statement. If no return statement was encountered before the end of the function body, the function call will be evaluated to have the value of boolean `false`.

3.7. Writing programs

3.7.1 Structure of a program

At the top level, a cardigan program is a series of assignment statements, either variable assignments, or function definitions. Variables declared at the top level will be treated as global variables. These may not make use of function calls, but may only consist of constants or expressions involving previously defined global variables. All function declarations will be stored for later use. A special function called `PLAY` must be included in every program as the entry point of execution.

3.7.2 File extensions

Cardigan source files have the extension .crd

3.7.3 Built in Functions

There are four built in functions in Cardigan: `INPUT`, `OUTPUT`, `CAST`, and `PLAY`.

`INPUT` is used to get user input. It should be called as a function with no arguments. When this function is called, execution of the program pauses until a user enters a value on the terminal. The function is then evaluated to have the value of whatever the user entered. The result of `INPUT` is always a string.

`OUTPUT` is used to display text on the screen. It should be called as a function with a single string argument. The argument is the string which will be printed on the screen. `OUTPUT` does not append a newline to the output string. If a newline is desired, it must be inserted in the string before `OUTPUT` is called.

`CAST` is used to cast a variable of one type to another. It should be called as a function with two arguments. The first is the value to be casted, the second is a primitive type keyword (int, float, string, bool) which represents the desired output type. If the value cannot be cast, the compiler throws an exception.

input type	**type keyword**	**output**
int	int	int
int	float	float
int	string	string
int	bool	`false` if the int is 0 `true` otherwise
float	int	truncate float before decimal
float	float	float
float	string	string

float	bool	`false` if the float is 0.0 `true` otherwise
string	int	an integer representation of the string if the string contains only numerals
string	float	a float containing the value represented by the string. The string must contain a valid float definition as described in the float primitive section above. An error is thrown otherwise
string	string	string
string	bool	`false` if the string is empty `true` otherwise
bool	int	1
bool	float	1.0
bool	string	`true` if the bool is true, or `false` if the bool is false
bool	bool	bool

`PLAY` is not defined by the language, but is used as the entry point to the program. Any global variables declared outside of function definitions will be assigned first, and then control will begin at the start of the `PLAY` function. When control reaches the end of the `PLAY` function the program terminates.

4. Project Plan

4.1 Process

The Cardigan group held 3 meetings a month during the semester to work out details of the language. Each of the specifications we created was a list comprising what each section of the compiler must support. For the scanner this was a list of tokens, for the parser this was a list of statements and expressions, for the AST this was a list of trees, and for the translator this was a list of java statements it would output.

Once we had the specification, we divided the responsibilities among group members. Miriam created the scanner, semantic analyzer, and test automation, Josh built the parser, including AST definitions, and implemented source code input & compiled code output, and created the final write up & slide set, Muzi handled code generation and Java-side support. Everyone worked on the translator together.

Once we were able to output compiled code, we all took turns creating test cases for each section of the compiler as it was implemented.

4.2 Style Guide

Naming
Function names should always use underscore separated names: function_names
Variable names should be camel cased: variableName

Function definitions
The first line of a function declaration should end with either the word function, the -> (for functions declared with “fun”) or the word “with” if the match statement is being used. The function should be preceded by a comment with a description of what the function does and an explanation of its arguments.

Indentation
To keep code readable, and associations between different sections clear, indent the body of the following structures: match blocks (except for the pipe symbol, see the section on matches below), if then/else blocks, function definitions, type definitions, and any parenthesized expression which is more than one line long.

Matches
Match statements should not include more than one pattern per line. All cases for the same match should have the same indentation. Patterns should be matched as deep as possible to avoid nested matching and extra empty matches

GOOD

```
  A(B(x)) -> eval_b x
| A(C(x)) -> eval_c x
| A(_) -> raise (Failure “…”)
```

NOT SO GOOD

```
A(x) -> match x wth
    B(y) -> eval_b y
|   C(y) -> eval_c y
|    _ -> raise (Failure “...”)
```

Long Lines
Lines which exceed 80 characters should be broken up across multiple lines. Try to break the lines up at some structural boundary (ie. concatenation operators or nested function calls). Each section of the line should be indented far beyond the current indentation level so it will not be confused with a separate statement.

Imperative features
If imperative features need to simulated, with “ignore” or “let _ = .. in”, the subsequent line should be at the same level of indentation.

Above all else, the highest priority should be readability of code. If following the rules in the style guide make code more difficult to read, then break them.

4.3

9/17	Decide on language
9/24	Determine basic syntax
	Agree on keywords, operators & built in functions

9/18-9/26	Write Proposal
10/1	Formally Define syntax
10/26	Build Scanner & 1st version of Parser
10/27-10/29	Write LRM
10/20-12/12	Update AST & Parser
12/3	Write source file input & compiled code output
12/12	Implement assignment statements
	Adapt test automation
	Update AST
	Update Parser
12/12-12/17	Implement translation cases
	Add test cases
	Create semantic analyzer
12/15	Write example programs
12/17	Write final report
12/19	Give presentation

4.4 Roles and Responsibilities

Muzi Gao -- At the first stage of this project, I worked with team for our first version of Parser and AST, as well as the documents. And after that, I was working on java classes that are needed to support our derived types (collection, struct, and enum). I worked with Miriam on “cardigan.ml” to combine sementic analysis with code generation. I took the responsibility to write most of codes that help output java codes, and as well as those relevant test cases.

Joshua Lopez -- As project leader I organized our meetings and assessed the status of the project so we could create the next set of tasks. I organized and wrote the outline for the project proposal and language reference manual as well as writing several of the sections. I organized and built the parser and ast definitions for Cardigan, including determining the languages CFG. I wrote the functionality in the translator to support assignment statements, function declarations, parenthesized expressions, and variable updates (+=, -=). Miriam created most of the syntactic analysis code, but I added a section which could handle keeping track of local variables and function return types. I also wrote the final report (except for the other member’s roles & responsibilities and lessons learned sections), the sample code, and prepared the demonstration slideshow.

Miriam Melnick -- I wrote most of the scanner, with organizational help from Josh. I worked with the team to develop our initial parser and AST. After that, I took charge of the semantic analysis code and the test suite. Muzi wrote a lot of her own test cases

too. I modified the microc testall script and also made it available through our Makefile. I helped edit the LRM, the presentation, and this final report. Most of the credit for this report goes to Josh. Muzi did a phenomenal job with all of the Java output - she and I worked closely in the cardigan.ml file to integrate our semantic analysis and code generation. I implemented code generation for simple unary and binary operations and other expressions.

4.5 Software Development Environment

The team used Google Code for version control and Google Docs & Google Drive to create the proposal, LRM, final report, and slideshow presentation. We also used Google Hangouts and G-Chat for remote meetings. For coding Miriam used Sublime Text, Josh used Aptana Studio, and Muzi used gedit for OCaml and Eclipse for Java. We also used the traditional blackboard and whiteboard to plan out the structure of the language syntax and the structure if the AST definitions. For testing, Miriam adapted Professor Edwards' bash test automation script.

4.6 Project Log

Wed Dec 19 22:57:42 2012 -0500: Miriam Melnick -- Implementing update
Wed Dec 19 22:29:57 2012 -0500: Miriam Melnick -- Fixing goFish.crd
Wed Dec 19 21:25:27 2012 -0500: Miriam Melnick -- Setting script to exit on first failure
Wed Dec 19 21:14:06 2012 -0500: Miriam Melnick -- Fixing cardigan script to compile cardigan to
Wed Dec 19 21:07:22 2012 -0500: Miriam Melnick -- working java code
Wed Dec 19 21:02:18 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 21:02:15 2012 -0500: Miriam Melnick -- CamelCasing
Wed Dec 19 20:38:06 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 20:37:56 2012 -0500: Skunkwaffle -- Go Fish
Wed Dec 19 20:33:39 2012 -0500: Muzi Gao -- add more comments
Wed Dec 19 20:00:30 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 20:00:25 2012 -0500: Miriam Melnick -- Java output files now have capitalized file names
Wed Dec 19 19:12:33 2012 -0500: Muzi Gao -- merge conflicts
Wed Dec 19 19:11:19 2012 -0500: Muzi Gao -- add comments, not finished
Wed Dec 19 18:56:29 2012 -0500: Miriam Melnick -- fixing the last non-parse error test. Starting on the parse error ones

Wed Dec 19 18:51:48 2012 -0500: Miriam Melnick -- Fixing tabs in rules statements
Wed Dec 19 18:40:21 2012 -0500: Miriam Melnick -- Fixing some more gold files
Wed Dec 19 18:25:18 2012 -0500: Miriam Melnick -- Tabbing
Wed Dec 19 17:57:12 2012 -0500: Miriam Melnick -- Fixed reversal problem for... 4th time?
Wed Dec 19 17:29:52 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 17:29:49 2012 -0500: Miriam Melnick -- Substituting main for play in the output
Wed Dec 19 17:27:25 2012 -0500: Muzi Gao -- print rule statement
Wed Dec 19 16:55:21 2012 -0500: Muzi Gao -- merge conflicts
Wed Dec 19 16:54:27 2012 -0500: Muzi Gao -- fixed collection function call
Wed Dec 19 16:49:31 2012 -0500: Miriam Melnick -- Fixing rules test
Wed Dec 19 16:34:53 2012 -0500: Miriam Melnick -- Fixing reversal issue
Wed Dec 19 16:27:10 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 16:27:06 2012 -0500: Muzi Gao -- add collection function access test files
Wed Dec 19 16:26:16 2012 -0500: Miriam Melnick -- Updating gold files
Wed Dec 19 16:10:00 2012 -0500: Miriam Melnick -- Fixing testall
Wed Dec 19 15:52:44 2012 -0500: Miriam Melnick -- Fixing testall script
Wed Dec 19 15:44:14 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 15:44:09 2012 -0500: Miriam Melnick -- Saving java files that aren't auto-generated
Wed Dec 19 15:40:17 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 15:40:07 2012 -0500: Skunkwaffle -- Fixed newline issue... again
Wed Dec 19 15:27:50 2012 -0500: Miriam Melnick -- Adding Muzi's java files
Wed Dec 19 15:24:31 2012 -0500: Miriam Melnick -- Fixing name of cardigan output
Wed Dec 19 14:16:02 2012 -0500: Miriam Melnick -- public static void main
Wed Dec 19 13:49:06 2012 -0500: Miriam Melnick -- Muzi's java infrastructure
Wed Dec 19 13:37:20 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 13:37:14 2012 -0500: Skunkwaffle -- fixed newline issue
Wed Dec 19 13:07:18 2012 -0500: Miriam Melnick -- Working on structs
Wed Dec 19 12:44:16 2012 -0500: Miriam Melnick -- Renamed function call test since it didn't have a function call
Wed Dec 19 12:40:51 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 12:40:48 2012 -0500: Miriam Melnick -- Fixing output for struct

Wed Dec 19 12:32:05 2012 -0500: Muzi Gao -- merge conflicts
Wed Dec 19 12:19:37 2012 -0500: Muzi Gao -- remove decrement2.crd
Wed Dec 19 12:12:53 2012 -0500: Miriam Melnick -- Merge
Wed Dec 19 12:05:04 2012 -0500: Miriam Melnick -- Adding struct elements to symbol table
Wed Dec 19 12:01:15 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 19 12:01:07 2012 -0500: Muzi Gao -- print property of structs, function call still needed
Wed Dec 19 11:40:21 2012 -0500: Miriam Melnick -- fixing test cases
Wed Dec 19 11:39:12 2012 -0500: Miriam Melnick -- Fixing a few tests
Wed Dec 19 02:34:40 2012 -0500: Muzi Gao -- merge conflict and add indeterminate type for collection-access
Wed Dec 19 01:51:05 2012 -0500: Muzi Gao -- fixing increment/decrement
Tue Dec 18 23:14:30 2012 -0500: Miriam Melnick -- Added currentFunction and untypedArgs
Tue Dec 18 22:15:04 2012 -0500: Miriam Melnick -- Functions can now return ids
Tue Dec 18 21:58:04 2012 -0500: Miriam Melnick -- Fixing some too-long lines
Tue Dec 18 21:51:12 2012 -0500: Muzi Gao -- merge conflict
Tue Dec 18 21:48:20 2012 -0500: Muzi Gao -- add access-enum
Tue Dec 18 21:45:15 2012 -0500: Miriam Melnick -- Fixed increment
Tue Dec 18 21:33:26 2012 -0500: Miriam Melnick -- Fixing order of block evaluation for the third time
Tue Dec 18 20:59:19 2012 -0500: Miriam Melnick -- Got argument types to update
Tue Dec 18 20:11:14 2012 -0500: Miriam Melnick -- fixing a typo in test script
Tue Dec 18 20:09:28 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Tue Dec 18 20:09:26 2012 -0500: Miriam Melnick -- Incrementing nestingLevel in control structures for scoping purposes
Tue Dec 18 20:07:19 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Tue Dec 18 20:07:11 2012 -0500: Muzi Gao -- merge conflicts
Tue Dec 18 19:54:58 2012 -0500: Miriam Melnick -- Got return types working for functions if you return a literal. working on ids
Tue Dec 18 19:54:19 2012 -0500: Muzi Gao -- add collectionbounds
Tue Dec 18 19:28:56 2012 -0500: Miriam Melnick -- Starting to implement untypedArgs
Tue Dec 18 18:42:14 2012 -0500: Miriam Melnick -- Still working on function types - having some trouble with argument type assignment
Tue Dec 18 18:22:10 2012 -0500: Miriam Melnick -- Starting prepping for tabs in output

Tue Dec 18 18:10:56 2012 -0500: Miriam Melnick -- Added function return types and partial argument handling
Tue Dec 18 15:51:29 2012 -0500: Muzi Gao -- dealing with collection access
Tue Dec 18 15:50:38 2012 -0500: Miriam Melnick -- Now calling save_assignment_to_hash
Tue Dec 18 15:45:22 2012 -0500: Miriam Melnick -- Added a few more function comments
Tue Dec 18 15:34:01 2012 -0500: Miriam Melnick -- Adding comments to first few functions
Tue Dec 18 15:08:13 2012 -0500: Muzi Gao -- merge conflicts
Tue Dec 18 14:28:01 2012 -0500: Muzi Gao -- print for loop
Tue Dec 18 13:59:56 2012 -0500: Skunkwaffle -- Updated code to follow style guide
Tue Dec 18 12:32:14 2012 -0500: Miriam Melnick -- Fixed all warnings
Tue Dec 18 12:18:26 2012 -0500: Miriam Melnick -- All tests passing ecept those requiring function return types
Tue Dec 18 12:08:46 2012 -0500: Miriam Melnick -- Fixing binops - no longer evaluating in ocaml
Tue Dec 18 04:39:32 2012 -0500: Skunkwaffle -- handled all warnings
Mon Dec 17 16:59:47 2012 -0500: Miriam Melnick -- fixed invert/negate type errors
Mon Dec 17 16:22:53 2012 -0500: Miriam Melnick -- Fixing test script to catch runtime errors
Mon Dec 17 16:18:19 2012 -0500: Miriam Melnick -- Updated test script - shows all failed tests at bottom
Mon Dec 17 16:03:49 2012 -0500: Miriam Melnick -- Almost all tests passing
Mon Dec 17 15:56:53 2012 -0500: Miriam Melnick -- merge
Mon Dec 17 15:54:34 2012 -0500: Miriam Melnick -- ALL THE THINGS pass (well, almost)
Mon Dec 17 15:19:05 2012 -0500: Miriam Melnick -- Fixing semicolons
Mon Dec 17 14:57:49 2012 -0500: Miriam Melnick -- Results printing in right order and being added to table in right order. Many tests failing - fixing that now.
Mon Dec 17 14:48:12 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Mon Dec 17 14:48:01 2012 -0500: Muzi Gao -- add ifstatement
Mon Dec 17 14:47:42 2012 -0500: Miriam Melnick -- finally doing things in the right order - will recommit soon with full solution.
Mon Dec 17 14:43:45 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Mon Dec 17 14:43:36 2012 -0500: Skunkwaffle -- Corrected newline handling after code block
Mon Dec 17 13:25:59 2012 -0500: Muzi Gao -- add while-statement test

Mon Dec 17 13:05:31 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Mon Dec 17 13:05:26 2012 -0500: Miriam Melnick -- Added geq and tests
Mon Dec 17 13:01:56 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Mon Dec 17 13:01:43 2012 -0500: Skunkwaffle -- Added support for multiple newlines
Mon Dec 17 12:56:12 2012 -0500: Miriam Melnick -- Renamed some tests
Mon Dec 17 12:49:27 2012 -0500: Miriam Melnick -- Fixed a couple of Nithin's tests
Mon Dec 17 12:43:16 2012 -0500: Miriam Melnick -- Changed cardigan.ml to use exclusively addVar and addFunc funtions
Mon Dec 17 12:32:05 2012 -0500: Miriam Melnick -- Using getVarType
Mon Dec 17 12:22:26 2012 -0500: Miriam Melnick -- All functions from semantic.ml now copied into cardigan.ml
Mon Dec 17 12:20:50 2012 -0500: Miriam Melnick -- Frec(args, returntype)
Mon Dec 17 12:16:16 2012 -0500: Miriam Melnick -- Adding more functions from semantic.ml to cardigan.ml
Mon Dec 17 12:09:08 2012 -0500: Miriam Melnick -- Removed enum form every Vrec. Now handling enums as all other derived types
Mon Dec 17 12:01:25 2012 -0500: Miriam Melnick -- Starting to integrate semantic.ml and cardigan.ml
Mon Dec 17 11:29:42 2012 -0500: Miriam Melnick -- Added gt
Mon Dec 17 11:26:04 2012 -0500: Miriam Melnick -- Implemented lt, neq
Mon Dec 17 06:01:10 2012 -0500: Skunkwaffle -- renamed test case
Sun Dec 16 23:02:21 2012 -0500: Miriam Melnick -- Implemented /,%,and,or. NOT currently works with the word 'not', not with the exclamation mark symbol
Sun Dec 16 22:36:08 2012 -0500: Miriam Melnick -- Just realized update (ex. +=) has not been implemented yet. Marked it as such.
Sun Dec 16 22:29:35 2012 -0500: Miriam Melnick -- Implemented +,-,*
Sun Dec 16 22:01:38 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Sun Dec 16 22:01:30 2012 -0500: Skunkwaffle -- WAR
Sun Dec 16 20:01:31 2012 -0500: Miriam Melnick -- Implementing negate, invert, increment, decrement
Sun Dec 16 03:09:32 2012 -0500: Muzi Gao -- merge conflicts
Sun Dec 16 03:02:51 2012 -0500: Muzi Gao -- fixed access collection, add input & output
Sun Dec 16 02:27:48 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Sun Dec 16 02:27:44 2012 -0500: Skunkwaffle -- Created external semantic analysis script

Sat Dec 15 23:19:25 2012 -0500: Miriam Melnick -- Fixed several warnings about partial function applications
Sat Dec 15 16:37:09 2012 -0500: Miriam Melnick -- Renaming tests to have valid names
Sat Dec 15 16:17:44 2012 -0500: Skunkwaffle -- fixed errors in cardigan.ml
Sat Dec 15 02:57:33 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Sat Dec 15 02:57:26 2012 -0500: Muzi Gao -- deal with cast(x,y)
Fri Dec 14 21:55:59 2012 -0500: Nithin Chandrasekharan -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 20:50:01 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 20:49:52 2012 -0500: Skunkwaffle -- fixed backslash handling in strings
Fri Dec 14 20:18:33 2012 -0500: Nithin Chandrasekharan -- current edits for now
Fri Dec 14 20:05:12 2012 -0500: Nithin Chandrasekharan -- think i added new files
Fri Dec 14 19:56:34 2012 -0500: Nithin Chandrasekharan -- Edited type_of_expr in cardigan added some types
Fri Dec 14 19:24:18 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 19:24:14 2012 -0500: Skunkwaffle -- Input takes an argument now
Fri Dec 14 19:22:03 2012 -0500: Nithin Chandrasekharan -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 17:56:03 2012 -0500: Miriam Melnick -- Test script can now also check failures
Fri Dec 14 17:36:30 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 17:36:26 2012 -0500: Miriam Melnick -- Finished cast function
Fri Dec 14 16:09:48 2012 -0500: Skunkwaffle -- Final versions of scanner, parser, and ast
Fri Dec 14 15:52:44 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 15:52:39 2012 -0500: Skunkwaffle -- Added function calls
Fri Dec 14 15:50:58 2012 -0500: Miriam Melnick -- Testing string to float casting - it works
Fri Dec 14 15:46:12 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 15:46:08 2012 -0500: Miriam Melnick -- Cast string to int now working
Fri Dec 14 14:28:40 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 14:28:30 2012 -0500: Muzi Gao -- nested collection

Fri Dec 14 14:14:43 2012 -0500: Skunkwaffle -- added for looooooooooops
Fri Dec 14 14:04:57 2012 -0500: Skunkwaffle -- Added rules
Fri Dec 14 13:49:27 2012 -0500: Skunkwaffle -- Added while
Fri Dec 14 13:35:33 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Fri Dec 14 13:35:26 2012 -0500: Skunkwaffle -- Added if/else if/else
Fri Dec 14 11:53:36 2012 -0500: Muzi Gao -- quick fix of enum
Fri Dec 14 11:48:15 2012 -0500: Miriam Melnick -- Putting Muzi's code back (sorry I deleted it)
Fri Dec 14 11:39:35 2012 -0500: Skunkwaffle -- Fixed shift/reduce conflicts
Fri Dec 14 11:33:31 2012 -0500: Miriam Melnick -- All primitive types can cast to themselves
Fri Dec 14 11:25:03 2012 -0500: Miriam Melnick -- Casting works from int to int
Fri Dec 14 11:11:06 2012 -0500: Miriam Melnick -- Adding a note to eval_block - when you call it, add the curly braces to the result of the function
Fri Dec 14 11:09:25 2012 -0500: Miriam Melnick -- Blocks no longer have tons of extra curly braces. All tests passing except enum (which I will fix soon).
Thu Dec 13 22:06:29 2012 -0500: Nithin Chandrasekharan -- Added basic tests
Thu Dec 13 22:01:38 2012 -0500: Nithin Chandrasekharan -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 17:46:58 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 17:46:44 2012 -0500: Muzi Gao -- interpret enum object
Thu Dec 13 17:14:56 2012 -0500: Skunkwaffle -- added dot lookup and collection indexing
Thu Dec 13 16:52:56 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 16:52:51 2012 -0500: Skunkwaffle -- added return statemtn
Thu Dec 13 16:46:14 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 16:46:06 2012 -0500: Muzi Gao -- merge conflicts
Thu Dec 13 16:43:54 2012 -0500: Skunkwaffle -- added increment and decrement
Thu Dec 13 16:34:05 2012 -0500: Skunkwaffle -- Added dummy code for binary op evaluation in interpereter
Thu Dec 13 16:29:37 2012 -0500: Muzi Gao -- interpret new collection object
Thu Dec 13 16:28:12 2012 -0500: Skunkwaffle -- Added binary operations
Thu Dec 13 16:07:24 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 16:07:19 2012 -0500: Skunkwaffle -- added unary negation, algebreic and boolean

Thu Dec 13 16:05:37 2012 -0500: Miriam Melnick -- Fixing typo causing syntax error
Thu Dec 13 16:04:07 2012 -0500: Miriam Melnick -- Merge with Josh's code
Thu Dec 13 16:03:00 2012 -0500: Miriam Melnick -- Passing a test with a function body. Outputting semicolons after stmts, braces around blocks.
Thu Dec 13 15:59:26 2012 -0500: Skunkwaffle -- Added OUTPUT function
Thu Dec 13 15:53:47 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 15:53:40 2012 -0500: Skunkwaffle -- added INPUT function
Thu Dec 13 15:49:26 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 15:49:20 2012 -0500: Miriam Melnick -- All tests pass. Statements are being evaluated and function bodies should be working
Thu Dec 13 15:35:57 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 15:35:52 2012 -0500: Skunkwaffle -- added CAST function
Thu Dec 13 14:39:18 2012 -0500: Nithin Chandrasekharan -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 14:27:56 2012 -0500: Miriam Melnick -- Changing java class names to have underscores instead of hyphens
Thu Dec 13 14:12:07 2012 -0500: Skunkwaffle -- conflict fix on cardigan.ml
Thu Dec 13 14:10:21 2012 -0500: Skunkwaffle -- added updates to parser with block and statement evaluation in interpreter
Thu Dec 13 14:03:20 2012 -0500: Muzi Gao -- add enum type in test-assign-enum.gold
Thu Dec 13 14:01:48 2012 -0500: Muzi Gao -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 14:00:18 2012 -0500: Muzi Gao -- adding javatype before variable assignment
Thu Dec 13 13:38:23 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 13:38:19 2012 -0500: Miriam Melnick -- Enums now working with symbol table.
Thu Dec 13 13:07:39 2012 -0500: Skunkwaffle -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 13:07:29 2012 -0500: Skunkwaffle -- Struct and parenthesized expressiong support in parser & ast
Thu Dec 13 12:54:00 2012 -0500: Muzi Gao -- add failure message for invalid type
Thu Dec 13 12:51:33 2012 -0500: Miriam Melnick -- Adding function type_of_variable to compiler
Thu Dec 13 12:34:25 2012 -0500: Skunkwaffle -- Added dummy enum handling to interpreter

Thu Dec 13 12:18:20 2012 -0500: Miriam Melnick -- Trying to merge
Thu Dec 13 12:17:08 2012 -0500: Miriam Melnick -- Adding symbol table to some eval functions
Thu Dec 13 12:10:55 2012 -0500: Skunkwaffle -- removing redundant code from the interpreter and ast
Thu Dec 13 11:35:28 2012 -0500: Nithin Chandrasekharan -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 11:31:45 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Thu Dec 13 11:31:42 2012 -0500: Miriam Melnick -- Removing that test; we decided it is an acceptable failure
Thu Dec 13 11:31:07 2012 -0500: Skunkwaffle -- Added Enum support in parser & ast
Thu Dec 13 11:18:12 2012 -0500: Miriam Melnick -- Comments working with one caveat: comments ended with newline do not yet include EOL. test-comment-backslash-trick demonstrates this failure. we should fix it.
Thu Dec 13 11:04:41 2012 -0500: Miriam Melnick -- Adding support for multi-line commands separated by backslash
Thu Dec 13 00:27:19 2012 -0500: Miriam Melnick -- Collection declaration now adds records to symbol table
Wed Dec 12 23:33:47 2012 -0500: Miriam Melnick -- Merging Josh's Collection changes
Wed Dec 12 23:06:22 2012 -0500: Skunkwaffle -- collection tests
Wed Dec 12 23:05:16 2012 -0500: Skunkwaffle -- Added Collection declaration. Interior values are not being added to the symbol table
Wed Dec 12 22:35:58 2012 -0500: Skunkwaffle -- fixing conflict
Wed Dec 12 22:31:02 2012 -0500: Skunkwaffle -- Collection works in ast but not in symbol table
Wed Dec 12 22:26:30 2012 -0500: Miriam Melnick -- All tests pass -- yay. AST is being compiled in correct order.
Wed Dec 12 21:05:18 2012 -0500: Miriam Melnick -- Merge with Josh's code
Wed Dec 12 21:02:38 2012 -0500: Miriam Melnick -- Muzi and I got types working -- yay
Wed Dec 12 20:51:57 2012 -0500: Skunkwaffle -- Tests for mulit arg functions
Wed Dec 12 20:49:07 2012 -0500: Skunkwaffle -- funciton definitions with arguments
Wed Dec 12 20:00:35 2012 -0500: Skunkwaffle -- Added function assignment with no arguments and empty block
Wed Dec 12 19:39:49 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 12 19:39:37 2012 -0500: Miriam Melnick -- Added print hash functionality for symbol table
Wed Dec 12 19:16:00 2012 -0500: Skunkwaffle -- Fixex prior conflict
Wed Dec 12 19:13:21 2012 -0500: Skunkwaffle -- Fake functions are allowed

Wed Dec 12 19:09:09 2012 -0500: Miriam Melnick -- Merging changes to cardigan.ml
Wed Dec 12 18:40:43 2012 -0500: Skunkwaffle -- Added assigning identifiers the value of other identifiers
Wed Dec 12 18:24:20 2012 -0500: Skunkwaffle -- Added assignment for all primative data types
Wed Dec 12 16:51:39 2012 -0500: Miriam Melnick -- Fixing Makefile and adding .java, .out, .diff to .gitignore
Wed Dec 12 16:48:39 2012 -0500: Miriam Melnick -- Merge branch 'java' of https://code.google.com/p/cardigan-plt into java
Wed Dec 12 16:47:35 2012 -0500: Miriam Melnick -- Renaming testall.sh to testall
Wed Dec 12 16:47:27 2012 -0500: Skunkwaffle -- Assignment tests
Wed Dec 12 16:36:50 2012 -0500: Skunkwaffle -- interpreter that ALMOST outputs valid jave code
Wed Dec 12 15:55:01 2012 -0500: Miriam Melnick -- Signalling which file passed testall.sh
Wed Dec 12 15:50:35 2012 -0500: Miriam Melnick -- Test script now compares .java to .gold
Wed Dec 12 15:38:05 2012 -0500: Miriam Melnick -- Updating test script to keep java files
Wed Dec 12 14:52:34 2012 -0500: Skunkwaffle -- OMG it works
Wed Dec 12 14:39:16 2012 -0500: Nithin Chandrasekharan -- ast2.mli
Wed Dec 12 12:40:10 2012 -0500: Miriam Melnick -- New parser based on varAssignments
Wed Dec 12 12:06:54 2012 -0500: Miriam Melnick -- Welcome to java branch
Wed Dec 12 11:55:43 2012 -0500: Miriam Melnick -- Putting in UnaryNegInt as a temporary fix for UnaryNeg.
Wed Dec 12 04:11:14 2012 -0500: Josh Lopez -- fixed string handling in the scanner and added test case for escape chars
Sat Dec 8 22:37:57 2012 -0500: Nithin Chandrasekharan -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Sat Dec 8 22:33:13 2012 -0500: Nithin Chandrasekharan -- Have updated the types and also added string_of_exp
Wed Dec 5 17:52:36 2012 -0500: Miriam Melnick -- To test alternate ast, (1) change first line of parser to 'open Ast2' and (2) run make alt
Wed Dec 5 17:34:16 2012 -0500: Nithin Chandrasekharan -- Alt AST
Wed Dec 5 16:56:20 2012 -0500: Miriam Melnick -- Added parenexpr. That test and str are currently failing
Wed Dec 5 16:42:39 2012 -0500: Miriam Melnick -- Committing the tests I just added
Wed Dec 5 16:42:11 2012 -0500: Miriam Melnick -- Added tests for floats and bools

Wed Dec 5 16:27:11 2012 -0500: Miriam Melnick -- moved literals into their own type in AST
Wed Dec 5 16:16:59 2012 -0500: Miriam Melnick -- Adding comments to cardigan.ml
Wed Dec 5 15:40:02 2012 -0500: Miriam Melnick -- Got id test passing
Mon Dec 3 21:22:27 2012 -0500: Miriam Melnick -- Updated testall.sh. Made cardigan print out results of eval
Mon Dec 3 19:07:03 2012 -0500: Josh Lopez -- changed interpreter to output, well, something
Mon Dec 3 18:57:22 2012 -0500: Josh Lopez -- adding message to interpreter
Mon Dec 3 18:53:29 2012 -0500: Miriam Melnick -- It compilesgit status
Mon Dec 3 18:32:11 2012 -0500: Miriam Melnick -- Still getting some build errors
Mon Dec 3 17:52:06 2012 -0500: Josh Lopez -- interpreter that tries to use the ast
Mon Dec 3 17:37:55 2012 -0500: Miriam Melnick -- Temp changes; not working yet
Wed Nov 28 17:56:06 2012 -0500: Miriam Melnick -- Made progress on moving things from parser to ast. Not done yet.
Mon Nov 26 18:04:44 2012 -0500: Miriam Melnick -- moving things to the ast
Mon Nov 26 17:36:30 2012 -0500: Miriam Melnick -- Deleting old microc tests
Mon Nov 26 16:45:06 2012 -0500: Miriam Melnick -- Adding cardigan and *.class to .gitignore
Sun Nov 25 18:01:43 2012 -0500: Josh Lopez -- Added compiler script cardigan.ml and removed its exclude rule from .gitignore
Mon Nov 19 17:44:45 2012 -0500: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Mon Nov 19 17:44:39 2012 -0500: Miriam Melnick -- adding enumStringOrId stringOrId to parser
Sat Nov 17 23:12:31 2012 -0500: Josh Lopez -- Added all symbols to scanner and updated parser
Mon Nov 12 23:10:37 2012 -0500: Miriam Melnick -- finished cleaning up parser
Mon Nov 12 22:41:21 2012 -0500: Miriam Melnick -- moving some more copies from parser into ast
Mon Nov 12 21:49:49 2012 -0500: Miriam Melnick -- starting to move copies from parser to ast
Mon Nov 12 18:08:20 2012 -0500: Miriam Melnick -- adding a comment to the scanner
Mon Nov 12 18:01:32 2012 -0500: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Mon Nov 12 18:01:17 2012 -0500: Josh Lopez -- scanner with comments
Mon Nov 12 17:59:29 2012 -0500: Josh Lopez -- Scanner with comments
Mon Nov 12 17:57:16 2012 -0500: Miriam Melnick -- now building the scanner too
Mon Nov 12 15:12:46 2012 -0500: Miriam Melnick -- adding float literals to scanner

Mon Nov 12 15:10:54 2012 -0500: Miriam Melnick -- Added string literals to scanner (for real).
Mon Nov 12 15:10:16 2012 -0500: Miriam Melnick -- Added string literals to scanner. Not yet hooked into rest of program.
Mon Nov 12 02:13:37 2012 -0500: Muzi Gao -- add dot to scanner
Mon Oct 29 17:32:23 2012 -0400: Miriam Melnick -- fixing shift/reduce error in collection declaration
Mon Oct 29 17:24:50 2012 -0400: Miriam Melnick -- fixed edge cases for collection declaration
Mon Oct 29 17:23:05 2012 -0400: Miriam Melnick -- fixing syntax for collections and allowing range syntax
Mon Oct 29 16:44:47 2012 -0400: Miriam Melnick -- allow empty collection
Mon Oct 29 16:42:54 2012 -0400: Miriam Melnick -- allow empty structs
Sat Oct 27 00:14:25 2012 -0400: Miriam Melnick -- fixing exprG
Sat Oct 27 00:12:19 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Sat Oct 27 00:12:14 2012 -0400: Miriam Melnick -- implementing unaryneg and cleaning things up
Sat Oct 27 00:01:17 2012 -0400: Miriam Melnick -- merging with Nithin's code
Fri Oct 26 23:51:01 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 23:50:57 2012 -0400: Miriam Melnick -- fixing typos INTerior in parser
Fri Oct 26 23:48:05 2012 -0400: Miriam Melnick -- got rid of all the not reduced errors
Fri Oct 26 23:44:07 2012 -0400: Miriam Melnick -- fewer rules not being reduced
Fri Oct 26 23:45:10 2012 -0400: Josh Lopez -- removed confilcts
Fri Oct 26 23:40:57 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 23:40:53 2012 -0400: Josh Lopez -- boolean is now a comparison
Fri Oct 26 23:38:27 2012 -0400: Josh Lopez -- boolean is now a comparison
Fri Oct 26 23:35:43 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 23:35:39 2012 -0400: Miriam Melnick -- adding funcCall to binary ops
Fri Oct 26 23:29:18 2012 -0400: Muzi Gao -- merge Josh's conflicts
Fri Oct 26 23:28:49 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 23:28:46 2012 -0400: Josh Lopez -- else structure
Fri Oct 26 23:26:15 2012 -0400: Muzi Gao -- add funcCall to binary comparision
Fri Oct 26 23:23:31 2012 -0400: Miriam Melnick -- adding rules structure
Fri Oct 26 22:52:35 2012 -0400: Miriam Melnick -- calling funcCall where appropriate
Fri Oct 26 22:50:19 2012 -0400: Miriam Melnick -- adding more expr forms

Fri Oct 26 22:48:19 2012 -0400: Miriam Melnick -- merge
Fri Oct 26 22:47:56 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:47:53 2012 -0400: Miriam Melnick -- fixing numOp to match ast
Fri Oct 26 22:47:03 2012 -0400: Muzi Gao -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:47:00 2012 -0400: Muzi Gao -- merge conflicts
Fri Oct 26 22:46:55 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:49:30 2012 -0400: Josh Lopez -- numop operators
Fri Oct 26 22:46:50 2012 -0400: Miriam Melnick -- adding valid exprs to the expr rule in parser
Fri Oct 26 22:44:44 2012 -0400: Muzi Gao -- add output strings to binary comparision
Fri Oct 26 22:43:51 2012 -0400: Miriam Melnick -- added numop
Fri Oct 26 22:41:22 2012 -0400: Muzi Gao -- merge Josh's work
Fri Oct 26 22:41:11 2012 -0400: Josh Lopez -- boolean fix
Fri Oct 26 22:40:14 2012 -0400: Josh Lopez -- unary negation
Fri Oct 26 22:39:40 2012 -0400: Miriam Melnick -- fixing merge
Fri Oct 26 22:39:12 2012 -0400: Muzi Gao -- binary comparision
Fri Oct 26 22:39:10 2012 -0400: Josh Lopez -- unary negation
Fri Oct 26 22:39:06 2012 -0400: Miriam Melnick -- some more binops
Fri Oct 26 22:37:03 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:36:58 2012 -0400: Josh Lopez -- plus and mod binary operators
Fri Oct 26 22:35:15 2012 -0400: Miriam Melnick -- merging changes Josh/Miriam
Fri Oct 26 22:34:31 2012 -0400: Miriam Melnick -- added MINUS
Fri Oct 26 22:32:49 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:32:37 2012 -0400: Josh Lopez -- some binary ops
Fri Oct 26 22:31:59 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:31:57 2012 -0400: Miriam Melnick -- maintaining consistency in parser
Fri Oct 26 22:28:01 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:27:59 2012 -0400: Miriam Melnick -- adding function calls
Fri Oct 26 22:23:47 2012 -0400: Josh Lopez -- dot lookup
Fri Oct 26 22:19:15 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 22:19:12 2012 -0400: Josh Lopez -- collection index
Fri Oct 26 22:14:40 2012 -0400: Miriam Melnick -- starting expr in parser

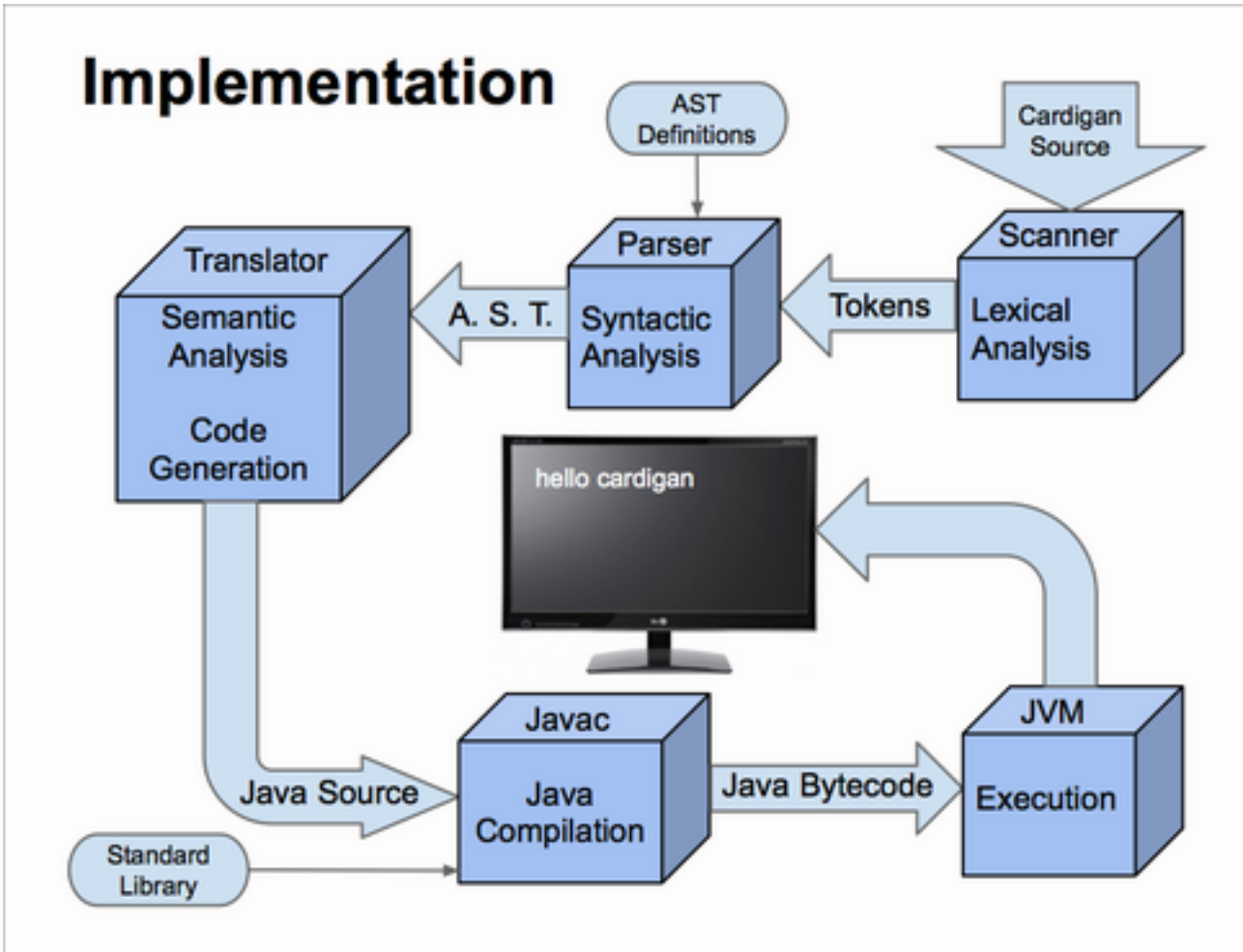
Fri Oct 26 22:08:11 2012 -0400: Miriam Melnick -- allowing functions to have no args
Fri Oct 26 21:33:03 2012 -0400: Miriam Melnick -- function body block fix
Fri Oct 26 21:30:34 2012 -0400: Miriam Melnick -- Merging Muzi's changes
Fri Oct 26 21:29:30 2012 -0400: Miriam Melnick -- merging Nithin's changes
Fri Oct 26 21:28:38 2012 -0400: Muzi Gao -- fixed derivedType
Fri Oct 26 21:27:36 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 21:27:34 2012 -0400: Miriam Melnick -- Fixed shift/reduce conflict on block
Fri Oct 26 21:07:32 2012 -0400: Miriam Melnick -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 21:09:45 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 21:09:21 2012 -0400: Josh Lopez -- struct fix
Fri Oct 26 21:07:23 2012 -0400: Miriam Melnick -- merge
Fri Oct 26 21:05:14 2012 -0400: Miriam Melnick -- Fixing I -> id
Fri Oct 26 20:58:24 2012 -0400: Muzi Gao -- derived type -- keep collection as placeholder now
Fri Oct 26 20:57:02 2012 -0400: Muzi Gao -- derived type -- keep collection as placeholder now
Fri Oct 26 20:56:56 2012 -0400: Miriam Melnick -- Resolving merge between Josh and Miriam
Fri Oct 26 20:55:05 2012 -0400: Muzi Gao -- derived type -- keep collection as placeholder now
Fri Oct 26 20:51:50 2012 -0400: Josh Lopez -- assignments
Fri Oct 26 20:51:44 2012 -0400: Miriam Melnick -- Merging Miriam's changes of the parser. Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 20:51:36 2012 -0400: Miriam Melnick -- Added stmt and block to parser
Fri Oct 26 20:49:56 2012 -0400: Josh Lopez -- assignment statements
Fri Oct 26 20:46:16 2012 -0400: Miriam Melnick -- adding *.output to gitignore
Fri Oct 26 20:45:41 2012 -0400: Miriam Melnick -- really merging
Fri Oct 26 20:45:15 2012 -0400: Miriam Melnick -- merge
Fri Oct 26 20:37:29 2012 -0400: Miriam Melnick -- trying again
Fri Oct 26 20:37:00 2012 -0400: Miriam Melnick -- Added block, stmt - with placeholders for A,E,W
Fri Oct 26 20:35:12 2012 -0400: Josh Lopez -- Added structs
Fri Oct 26 20:34:48 2012 -0400: Muzi Gao -- enum done
Fri Oct 26 20:34:01 2012 -0400: Muzi Gao -- enum done
Fri Oct 26 20:26:41 2012 -0400: Josh Lopez -- more tokens
Fri Oct 26 20:25:43 2012 -0400: Josh Lopez -- further tokens

Fri Oct 26 20:24:21 2012 -0400: Muzi Gao -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 20:22:05 2012 -0400: Muzi Gao -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 20:21:24 2012 -0400: Miriam Melnick -- adding more tokens
Fri Oct 26 20:19:03 2012 -0400: Josh Lopez -- token fix
Fri Oct 26 20:17:40 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 20:17:31 2012 -0400: Josh Lopez -- more tokens
Fri Oct 26 20:16:28 2012 -0400: Muzi Gao -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 20:16:25 2012 -0400: Muzi Gao -- backup
Fri Oct 26 20:11:44 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 20:11:35 2012 -0400: Josh Lopez -- Really added the parser info this time
Fri Oct 26 20:11:18 2012 -0400: Miriam Melnick -- got rid of 'never reduced' warnings in parser
Fri Oct 26 20:01:14 2012 -0400: Josh Lopez -- function identifiers
Fri Oct 26 20:01:01 2012 -0400: Miriam Melnick -- fixing gitignore
Fri Oct 26 19:57:26 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 19:57:15 2012 -0400: Josh Lopez -- identifiers
Fri Oct 26 19:54:46 2012 -0400: Josh Lopez -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 19:52:03 2012 -0400: Muzi Gao -- get rid of testing lines in README
Fri Oct 26 19:51:11 2012 -0400: Muzi Gao -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Fri Oct 26 19:50:52 2012 -0400: Josh Lopez -- identifier
Fri Oct 26 19:50:14 2012 -0400: Muzi Gao -- just try pushing
Fri Oct 26 19:49:01 2012 -0400: Miriam Melnick -- Added literal to parser and simplified scanning of booleans.
Fri Oct 26 19:31:05 2012 -0400: Miriam Melnick -- ast now only contains cardigan stuff
Fri Oct 26 18:06:35 2012 -0400: Miriam Melnick -- Test environment while we are building ast and parser. Disconnected from actually running programs.
Fri Oct 26 17:51:15 2012 -0400: Josh Lopez -- added operators
Fri Oct 26 17:21:55 2012 -0400: Miriam Melnick -- Scanner now passes booleans as a boolean type with a value of true/false
Fri Oct 26 15:31:36 2012 -0400: Miriam Melnick -- Changed comments from microc style to cardigan style.

Fri Oct 26 15:12:25 2012 -0400: Miriam Melnick -- Changed name to cardigan in Makefile and changed all tests to .crd files
Fri Oct 26 14:42:37 2012 -0400: Miriam Melnick -- Fixed bug with booleans. Parser now builds with no warnings and tests pass.
Mon Oct 22 18:58:12 2012 -0400: Nithin -- collections update
Mon Oct 22 18:42:22 2012 -0400: Nithin -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Mon Oct 22 18:37:13 2012 -0400: Nithin -- Merge branch 'master' of https://code.google.com/p/cardigan-plt
Mon Oct 22 18:32:11 2012 -0400: Nithin -- loops
Mon Oct 22 18:26:28 2012 -0400: Nithin -- loops
Mon Oct 22 18:04:08 2012 -0400: Nithin -- Okay this should finally work
Fri Oct 26 23:04:48 2012 -0400: Miriam Melnick -- made parser a little more readable
Fri Oct 26 23:00:59 2012 -0400: Miriam Melnick -- simplifying fake expression rules
Fri Oct 26 22:59:56 2012 -0400: Miriam Melnick -- added comparison rule to parser
Fri Oct 26 22:58:18 2012 -0400: Miriam Melnick -- expr=D. sweet.
Mon Oct 22 18:03:03 2012 -0400: Nithin -- Okay this should finally work
Mon Oct 22 16:11:00 2012 -0400: Nithin -- hh
Sat Oct 20 18:42:12 2012 -0400: Miriam Melnick -- Added boolean true/false to scanner and parser. We do not yet have the parser accepting anything but integers, so I gave false the code 5000 and true the code 5001. Not great for readability, but sufficient (I think) to check the ambiguity of our grammar.
Mon Oct 8 18:47:36 2012 -0400: Miriam Melnick -- initial commit. Using microc base with the addition of the mod operator.

5. Architectural Design

5.1 Components



5.2 Interfaces

Scanner:

input -- a lexbuf created using Lexing.from_string on the entire source program
output -- a Scanner.token stream

Parser:

input -- a Scanner.token stream

output -- an Ast.program object. The object should contain a list of Ast.Assignment objects

Translator:

input -- An Ast.program object

output -- A single string containing the compiled program, which is automatically saved into a .java file.

Within the Translator are several eval functions which handle translation of each type of component in an Ast. These functions all implement the following interface

tree -- the abstract syntax subtree to be evaluated

currentFunction -- the function we are currently defining

funcArgs -- the list of arguments to the function we are currently defining

symbols -- the symbol table for the scope in which the ast will be translated

nestingLevel -- the number of layers of nesting currently being applied

Final compilation script:

Calls make clean, make cardigan, executes cardigan on the given file, then runs javac on the output and runs the resulting .class.

5.3 Contributions

Scanner -- Miriam Melnick
Parser -- Joshua Lopez
AST -- Joshua Lopez
Semantic Analysis -- Miriam Melnick, Joshua Lopez
Translator -- Miriam Melnick, Muzi Gao, Joshua Lopez
Test cases -- Miriam Melnick, Muzi Gao, Joshua Lopez
Output code generation -- Muzi Gao, Miriam Melnick, Joshua Lopez
Java support classes -- Muzi Gao

6. Test Plan

6.1 Representative program -- GoFish.crd

```
PLAY()={

deck =
["A", "2", "3", "4", "5", "6", "7", "8", "9", "J", "Q", "K"]

    hand = []
    for i:[1:7]{
      card = deck.pop()
      hand.append(card)
    }

    guessed = false
    while not guessed{
      guess = INPUT(string)
      idx = 0
      while not guessed and idx < hand.length(){
           if hand[idx] == guess{
                guessed = true
           }
           idx++
      }

      if not guessed{
           OUTPUT("Go Fish!")
      }
    }

    OUTPUT("You win!")
}
```

6.2 Compiled program GoFish.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Scanner;
```

```
public class GoFish{
static cInput input = new cInput();
//Begin Cardigan Code

public static void main(String[] args){
     cCollection deck=new cCollection();
     deck.append("A");
     deck.append("2");
     deck.append("3");
     deck.append("4");
     deck.append("5");
     deck.append("6");
     deck.append("7");
     deck.append("8");
     deck.append("9");
     deck.append("J");
     deck.append("Q");
     deck.append("K");
     cCollection hand=new cCollection();

     cCollection loop=new cCollection(1,7);
     for(int i=0;i<loop.size();i++){
          int j=loop.find((int)i);
          cCollection card=deck.pop();;
          hand.append(card);
     }
     boolean guessed=false;
     while (!guessed) {
          String guess=input.cInputString();
          int idx=0;
          while ((!guessed)&&(idx<7)) {
               if(hand[idx]==guess) {
                    boolean guessed=true;
               }
               idx++;}
          if((!guessed)) {
               System.out.print("Go Fish!");
          }}
     System.out.print("You win!");
}
```

```
//End Cardigan Code
}
```

6.3 Test suites

6.3.1 Types of tests

Our automated test suite allows two types of programs: those which are supposed to succeed and those which are supposed to fail. Succeeding programs must begin with the prefix “test-” and failing programs must begin with “fail-”. For succeeding tests there must also be a .gold file with the same name as the Cardigan source file which contains the desired output. The Cardigan source is passed through the compiler and the resulting code is generated and compared to the .gold file. If they match the test passes. For failed tests, a compile time error must be generated. If this occurs the test is considered to have “failed successfully” and the test passes.

6.3.2 testall.sh

```
#!/bin/sh

CARDIGAN="./cardigan_"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

failed=""
generatedfiles=""


Usage() {
    echo "Usage: testall [options] [.crd files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}
```

```
SignalError() {
    if [ $error -eq 0 ] ; then
     echo "FAILED"
     error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any,
written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
     SignalError "$1 differs" &&
    failed="$failed \n $1"
     echo "FAILED $1 differs from $2" 1>&2
    }
}


# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    failed="$failed \n $2"
     SignalError "$1 failed on $*"
     return 1
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.crd//'`
    reffile=`echo $1 | sed 's/.crd$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
    javafile=`echo $basename |sed -e 's/^//g' -e 's/-/_/g'`
```

```
    ajavafile=`echo $javafile | perl -pe 's/\S+/\u$&/g'`

    newjavafile=`echo $ajavafile | perl -pe 's/([^ ])_([a-z])/
\\1\\u\\2/g'`

    echo 1>&2
    echo "###### Testing $basename" 1>&2


    generatedfiles="$generatedfiles tests/${newjavafile}.java
tests/${basename}.diff tests/${basename}.out" &&
    Run "$CARDIGAN" $1 ">" tests/${basename}.out &&
    Compare tests/${newjavafile}.java tests/${basename}.gold
tests/${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
     if [ $keep -eq 0 ] ; then
         rm -f $generatedfiles
     fi
     echo "OK - $basename succeeds"
     echo "###### SUCCESS" 1>&2
    else
     echo "###### FAILED" 1>&2
     globalerror=$error
    fi
}

SignalErrorFail() {
    if [ $error eq 0 ] ; then
    echo "$1 failed to fail"; else
    echo "OK - $* failed"
    fi
    echo "  $1"
}

# Run <args>
# Report the command, run it, and report any errors
RunFail() {
    echo $* 1>&2
```

```
    eval $* && {
    SignalErrorFail "$1 failed on $*"
    return 0
    }
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.crd//'`
    reffile=`echo $1 | sed 's/.crd$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
    javafile=`echo $basename |sed -e 's/^//g' -e 's/-/_/g'`
    newjavafile=`echo $javafile | perl -pe 's/\S+/\u$&/g'`

    echo 1>&2
    echo "###### Testing $basename" 1>&2


    generatedfiles="$generatedfiles tests/${basename}.diff
tests/${basename}.out" &&
    RunFail "$CARDIGAN" $1 ">" tests/${basename}.out

    # Report the status and clean up the generated files

    if [ $error -lt 1 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "###### FAILED TO FAIL" 1>&2
    globalerror=$error
    else
    echo "OK - $basename fails"
    echo "###### SUCCESSFULLY FAILED" 1>&2
    fi
}

while getopts kdpsh c; do
    case $c in
     k) # Keep intermediate files
         keep=1
```

```
          ;;
      h) # Help
          Usage
          ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.crd tests/fail-*.crd"
fi

for file in $files
do
    case $file in
      *test-*)
          Check $file 2>> $globallog
          ;;
    *fail-*)
        CheckFail $file 2>> $globallog
        ;;
      *)
          echo "unknown file type $file"
          globalerror=1
          ;;
    esac
done
tput setaf 1
echo "Failed tests: " $failed
tput sgr0

exit $globalerror
```

6.3.3 Test programs

```
fail-access-prove-statictype1.crd
```

```
b=45
a="test case"
b=a

------------

fail-access-prove-statictype2.crd

a=2
a="2"

------------

fail-assign.crd

x=
------------

fail-empty-call.crd

PLAY()
------------

fail-function-assign-play.crd

PLAY(b)={
b=4
return b
}

d=3
PLAY(d)
------------

fail-nested-scope.crd

PLAY()={
     x=false
     if x {
          b = 1
```

```
     }

     c = b
}

------------

fail-outofbounds-collection.crd

a = [3,4]
b = a[2]

------------

goFish.crd

PLAY()={
     deck =
["A", "2", "3", "4", "5", "6", "7", "8", "9", "J", "Q", "K"]

     hand = []
     loop=[1:7]
     for j:loop{
          card = deck.pop()
          hand.append(card)
     }

     guessed = false
     while not guessed{
          guess = INPUT(string)
          idx = 0

          while (not guessed and (idx < 7)){
               if hand[idx] == guess{
                    guessed = true
               }
               idx++
          }

          if not guessed{
               OUTPUT("Go Fish!")
```

```
          }
     }

     OUTPUT("You win!")
}
```

---

goldfish.crd

```
x()={
a = [3,"hello"]
b = a[0]
c = a[1]
}
```

---

guppie.crd

```
PLAY()={

deck =
["A", "2", "3", "4", "5", "6", "7", "8", "9", "J", "Q", "K"]
    a=1
    deck[a]
}
```

---

test-access-collection-function.crd

```
PLAY()={
     b=[1,2,3,4]
     b.shuffle()
     b.popLast()
}
```

---

test-access-collection.crd

```
x(){
a = [3,"hello"]
b = a[0]
c = a[1]
}
```

---

test-access-enum.crd

```
x=|a,b,c|
y=|d,e|
m=x.a
n=y.d
```

---

test-assign-boolean.crd

```
a=true
b=false
```
---

test-assign-collection.crd

```
a = [2, false, 2.3, "hello world"]
```

---

test-assign-collectionbounds.crd

```
a=[1:10]
```

---

test-assign-enum.crd

```
x=|a,b|
y=|a,c,e|
```

```
------------

test-assign-float.crd

a = 2.2
b = a

------------

test-assign-identifier.crd

b=2
a=b
------------

test-assign-int.crd

a=3

------------

test-assign-string.crd

a="testing 123"
b="escape\""
c="new line\n"
d="tab\t"
e="backslash\\"
f="all\\\n\t\""

------------

test-binop-and.crd

PLAY()={x=true and false}
------------

test-binop-divide-float-float.crd

PLAY() = {x=35.1/6.2}
------------
```

```
test-binop-divide-float-int.crd

PLAY() = {x=35.1/6}
------------

test-binop-divide-int-float.crd

PLAY() = {x=35/6.2}
------------

test-binop-divide-int-int.crd

PLAY() = {x=35/6}
------------

test-binop-geq-float-float.crd

PLAY()={b=3.0 >= 3.0}
------------

test-binop-geq-float-int.crd

PLAY()={b=3.0 >= 3}
------------

test-binop-geq-int-float.crd

PLAY()={b=3 >= 3.0}
------------

test-binop-geq-int-int.crd

PLAY()={b=3 >= 4}
------------

test-binop-gt-float-float.crd

PLAY()={b=43.2>31.3}
------------
```

```
test-binop-gt-float-int.crd

PLAY()={b=43.2>31}
------------

test-binop-gt-int-float.crd

PLAY()={b=40>321.3}
------------

test-binop-gt-int-int.crd

PLAY()={b=40>3}
------------

test-binop-leq-float-float.crd

PLAY()={b=4.4<=3.2}
------------

test-binop-leq-float-int.crd

PLAY()={b=4.1<=3}
------------

test-binop-leq-int-float.crd

PLAY()={b=4<=3.2}
------------

test-binop-leq-int-int.crd

PLAY()={b=4<=3}
------------

test-binop-lt-float-float.crd

PLAY()={b=5.4<12.1}
------------

test-binop-lt-float-int.crd
```

```
PLAY()={b=5.4<12}
------------

test-binop-lt-int-float.crd

PLAY()={b=5<12.2}
------------

test-binop-lt-int-int.crd

PLAY()={b=5<12}
------------

test-binop-minus-float-float.crd

PLAY() = {x=35.1-6.2}
------------

test-binop-minus-float-int.crd

PLAY() = {x=5.2-6}
------------

test-binop-minus-int-float.crd

PLAY() = {x=14-5.2}
------------

test-binop-minus-int-int.crd

PLAY() = {x=14-5}
------------

test-binop-mod-int-int.crd

PLAY()={x=50%4}
------------

test-binop-neq-collections.crd
```

```
a=[5,4]
b=[5,4]
c=[4,5]
d=   a!=b
e=   b!=c
f=   a!=a
------------

test-binop-neq-false.crd

PLAY()={b= true != true}
------------

test-binop-neq-true.crd

PLAY()={b= true != false}
------------

test-binop-or.crd

PLAY()={x=true or false}
------------

test-binop-plus-float-float.crd

PLAY() = {x=35.1+6.2}
------------

test-binop-plus-float-int.crd

PLAY() = {x=5.2+6}
------------

test-binop-plus-int-float.crd

PLAY() = {x=4+5.2}
------------

test-binop-plus-int-int.crd

PLAY() = {x=4+5}
```

```
------------

test-binop-plus-string-string.crd

PLAY() = {z="hello " + "world"}
------------

test-binop-times-float-float.crd

PLAY() = {x=3.1*6.2}
------------

test-binop-times-float-int.crd

PLAY() = {x=5.2*6}
------------

test-binop-times-int-float.crd

PLAY() = {x=3*5.2}
------------

test-binop-times-int-int.crd

PLAY() = {x=3*5}
------------

test-cast-bool-bool.crd

PLAY()={b=CAST(true,bool)}

------------

test-cast-bool-float-valid.crd

PLAY()={b=CAST(true,float)}

------------

test-cast-bool-int-valid.crd
```

```
PLAY()={b=CAST(true,int)}
------------

test-cast-bool-string-valid.crd

PLAY()={
     b=CAST(true,string)
     c=CAST(false,string)
}

------------

test-cast-float-bool-valid.crd

PLAY()={b=CAST(5.4,bool)}
------------

test-cast-float-float.crd

PLAY()={b=CAST(5.4,float)}
------------

test-cast-float-int-valid.crd

PLAY()={b=CAST(5.4,int)}
------------

test-cast-int-bool-valid.crd

PLAY()={
     b=CAST(5,bool)
     c=CAST(0,bool)
}

------------

test-cast-int-float-valid.crd

PLAY()={b=CAST(5,float)}
------------
```

test-cast-int-int.crd

```
PLAY()={b=CAST(5,int)}
```

---

test-cast-int-string-valid.crd

```
PLAY()={b=CAST(5,string)}
```

---

test-cast-string-bool-valid.crd

```
PLAY()={b=CAST("true",bool)}
```

---

test-cast-string-float-valid.crd

```
PLAY()={b=CAST("5.4",float)}
```

---

test-cast-string-int-valid.crd

```
PLAY()={b=CAST("10",int)}
```

---

test-cast-string-string.crd

```
PLAY()={b=CAST("rainbow",string)}
```

---

test-collection-nest.crd

```
PLAY()={
b=[1,"a",["hello",2],4.3]
c=4
d=[]}
```

---

test-comment-backslash.crd

```
a=\
// fooble
4
b=3
------------

test-comment-backslashes.crd

a=\
// fooble
4
//funsies
b=3
------------

test-comment.crd

a=4
// fooble
b=3
------------

test-decrement.crd

PLAY()={
     b=5
     c=5--
     d=b--
     e=5.3
     f=5.3--
     g=e--
}

------------

test-empty-function.crd

PLAY()={}
------------

test-function-body-several-stmts.crd
```

```
PLAY() = {
     b=5
     c=6
     d=7
}
------------
```

test-function-call-arg-constant.crd

```
PLAY(b)={
    b=4
    return b
}

c=PLAY(3)

------------
```

test-function-call-global.crd

```
PLAY()={}
a()={return 5}
c=a()
------------
```

test-function-call-play.crd

```
PLAY(b)={
b=4
return b
}

d=3
c=PLAY(d)

------------
```

test-function-call.crd

```
PLAY()={OUTPUT(5)}
```

```
a()={return 5}

b()={c=a()}
------------

test-function-empty-body.crd

PLAY()={a=5}

b()={}
------------

test-function-if.crd

PLAY(b)={
    if b>4 {
        c=4
    }
    else {
        c=6
    }

    return c
}

------------

test-function-local-body.crd

PLAY() ={
     d="testing 123 \n"
     e=true
     f=   7
}

------------

test-function-localvar.crd

PLAY()={b=3}
```

```
------------

test-function-multiple-args.crd

PLAY()={}

a(b, c)={
     b=5
    c=3
}

------------

test-function-one-arg.crd

PLAY(b) = {}

------------

test-function-reassign-arg.crd

a(b, c) ={
     b=5
     c=5.6
}

------------

test-function-return-arg.crd

PLAY()={}

a(b, c) ={
     b=5
     c=5.6
     return b
}

------------

test-function-return-float.crd
```

```
PLAY()={
     return 4.3
}
------------

test-function-return-id-int.crd

PLAY() = {
     d=7
     return d
}

------------

test-function-return-id-string.crd

PLAY()={
     d="foo"
     return d
}
------------

test-function-return-int.crd

PLAY() = {
     d=7
     return 7
}

------------

test-function-return-string.crd

PLAY()={
     return "foo"
}
------------

test-function-stmt-eval-order.crd
```

```
PLAY(b, c) ={
     b=5
     c=b + 0.6
     return c
}
```

---

test-functions-type-unclear.crd

```
PLAY(b) = {
     f = b+1
}
```

---

test-increment-float.crd

```
PLAY()={
     b=5.7
     c=5.7++
  d=b++
}
```

---

test-increment-int.crd

```
PLAY()={
     b=5
     c=5++
  d=b++
}
```

---

test-increment2.crd

```
PLAY()={}

a(b, c)={
```

```
    b=5
    b++
    c=3
}

d=7
------------

test-input.crd

PLAY()={
     x = INPUT(int)
     y = INPUT(float)
     z = INPUT(string)
     w = INPUT(bool)
}

------------

test-invert-bool.crd

PLAY()={b=not true}
------------

test-invert-id.crd

PLAY()={
    d=true
    b= not d
    return b
}
------------

test-linebreak-several.crd

a=\
\
\
\
\
3
```

```
b\
=4
------------

test-linebreak.crd

a=\
3
b=4
------------

test-local-global-scope.crd

d="1234"
PLAY() = {
     e=d
     d="new 123 \n"
     e=d
}

f=d
------------

test-multiple-assignments.crd

PLAY()={
a=3
b=5
c=7
d=9
e=11
}

f=6
------------

test-multiple-functions.crd

PLAY()={b=6}

c()={d=12}
```

```
e()={f=18}
------------

test-multiple-separators.crd

a   =       3
b = 4
c    =              "testing 123 \n"
d    =true
e=             5.6
f=7

------------

test-negate-id.crd

PLAY()={}

a(b)={
d= -3.3
b= -d
return b
}
------------

test-negation-minus.crd

PLAY()={x=5 - - 2}
------------

test-negation.crd

PLAY()={x=not true}
------------

test-output.crd

PLAY()={
     x=1
     y="hello world"
```

```
     OUTPUT(x)
     OUTPUT(y)
}
```

---

test-spaces.crd

```
a   =       3
b = 4
```

---

test-stmt-for.crd

```
PLAY()={
  y=[1:10]
     for x:y{
          sum = 1
     }

     for player:players{
       OUTPUT("test")
     }

}
```

---

test-stmt-if-else-if.crd

```
PLAY()={
     if x {
          b = 1
     }
     elseif y{
          b=2
     }
     else {
          b=3
     }
```

```
}
```

---

test-stmt-if-else-nested.crd

```
PLAY()={
     x=1
     y=1
     if x {
          b = 1
     }
     elseif y{
          if z{
               b=2
          }
          else{
               b=3
          }
     }
     else {
          b=4
     }
}
```

---

test-stmt-if-else.crd

```
PLAY()={
     x=true
     if x {
          b = 1
     }
     else{
          b=2
     }
}
```

---

test-stmt-if.crd

```
PLAY()={
     x=false
     if x {
          b = 1
     }
}
```

---

test-stmt-rule.crd

```
PLAY()={
    rules (flag1=1, flag2=false) {true {x=4} false {y=5}}
}
```

---

test-stmt-while.crd

```
PLAY()={
     valid = true
     while not valid {
          OUTPUT("hello")
          valid=false
     }
}
```

---

test-struct.crd

```
PLAY()={
  num=20
  b={name:"namestring",age:20,number:num}
}
```

---

test-struct2.crd

```
PLAY()={
  num=20
  b={name:"namestring",age:20,number:num}
  c=b.name
}

------------

test-tabs.crd

a    =              3
b    =4

------------

war.crd

rank=|two, three, four, five, six, seven, eight, nine, ten,
jack, queen, king, ace|
suit=|hearts, clubs, diamonds, spades|

HIGHEST = 0
NEXT = 1
WAR_CARDS = 3

PLAY()={
    deck = cartesian(rank, suit)   //Standard library func
    players = []                    //Empty collection
    OUTPUT("How many players?")     //Cardigan's print stmt
    numPlayers = INPUT(int)         //Specify input type
    for id:[1:numPlayers]{
          players.append({playerNumber:id, hand:[]})
     }    // Add players
     deck.shuffle()                      // built in function
     deck.deal(players)                  // standard library
     winner = {}                         // define winning
conditions
     while !winner {                     // while conditions have
not been met
     result = round(players)
```

```
     players[result.winner].hand.append(result.cards)
     if players[result.winner].hand.length == deck.length {
          winner = players[result.winner]
     }
    }
    OUPUT(CAST(winner.id, string) + " is the winner!")
}

round(players)={
    roundCards = [ ]
    for player:players {
        roundCards.append(player.hand.pop())
    }
    roundCards.sort(value)
     result = {}
    rules (highCard=roundCards[HIGHEST].value, next=NEXT){
        highCard.value > roundCards[next].value {
          result.winner = highCard.id
        }
        highCard.value == roundCards[next].value {
          participants = []
          participants.push(players[highCard.id])
          while roundCards[next].value == highCard {
               participants.push(players[roundCards[next].id])
          }
          warResults = war(participants)
          roundCards.append(warResults.cards)
          result.winner = warResults.winner
        }
    }
    result.cards = roundCards
    return result
}

war(participants) = {
     OUTPUT("War!")
     spoils = []
     for participant:participants{
          spoils.append(participant.hand.pop(WAR_CARDS))
     }
     result = round(participants)
```

```
    spoils.append(result.cards)
    return {winner: result.winner, cards: spoils}
}

------------
```

6.4 Testing method

As each type of expression/statement was added to the translator, we created a test case to ensure it was outputting code correctly. In most cases there was more than one kind of code that could be generated. For example:

if	if else if	if else	if else if else	if else if else if	if else if else if else

In these cases we created a test case for each of the possible cases an evaluation could be used to create.

6.5 Contributions

The test cases were automated using a version of the testall.sh script included with microc, with some modifications by Miriam Melnick.

Most of the test cases were written by the person who implemented the eval function for the corresponding Cardigan code.

In general, Muzi handled derived types, built in functions, and control structures, Miriam handled unary and binary operators and Josh handled primitive assignments, update statements and function declarations/calls. We all implemented these functions together at the same time so there may have been some overlap.

7. Lessons Learned

7.1 Muzi Gao

1. Having a great group makes things better and easier.
2. It's true that we possibly start too early without preparing well, but for me, it's hard to estimate the deadline unless I know the task well. Since then, it would always be better to start early and make it in an iterative way instead of knowing everything but without enough time to implement it.
3. Be creative in brainstorm, but be realistic when making plans.
5. Make priority for all tasks, and implement them one by one.
4. OCaml is hard to debug, especially when being compared with Java.

7.2 Joshua Lopez

1. It's possible to start too early. Implementing components before we really knew how to build them led to more work than just waiting another couple of weeks.
2. Newlines are a frustrating line delimiter. Regex matches and CFG patterns are way more complicated
3. Ocaml is unforgiving and unhelpful. The sooner you stop looking at its errors and start paying attention to what you write, the better off you'll be
4. Get to know your team members. The sooner you know each other's strengths and how much you can trust them, the better off everyone will be.
5. Set realistic deadlines. Building a scanner only takes a day or so, defining the abstract syntax isn't a small job.

7.3 Miriam Melnick

1. Wow, this project is so much easier with a great group.
2. Stay on task - don't let time pass without working on the project.
3. Be prepared and pick up each others' slack.
4. Be realistic - make a list of fancy features that would be nice but don't implement them until core functionality is working.
5. Commit early, commit often.
6. Test early, test even more often.
7. Communicate regularly and make sure everyone knows the status of the project and what they should be working on.

8. Notes

8.1 Missing Group Member

Originally, Cardigan was a four person group. Our fourth member, Nithin Chandrasekharan suffered from several health related issues which kept from from regularly attending group meetings. On more than one occasion when he was present, he had a seizure during the meeting.

Unfortunately his condition made it very difficult for Nithin to stay updated on the status of the project and he was not able to contribute to the development of the compiler as much as the rest of the group. Most of the code he checked in had to be rewritten by other members of the group, and he made only limited contributions to the project proposal and LRM, most of which also had to be rewritten. Nithin did create a few of the more exhaustive test cases which are included in the test suite, but most of these also had to be modified to support the exact output of each piece of functionality (ie. correct formatting, nested cases).

Realizing that he had made few significant contributions to the project, Nithin decided to withdraw from the group on December 15th, 2012, five days before the project deadline. The remaining members of Cardigan decided it would be better to complete the compiler without implementing all of the features we originally intended, rather support all features with a non-working compiler. Among these were including java source compilation into the compiler, implementing bisect for code coverage evaluation, and some of the more card-game related features (ie. winning conditions & predefined players, decks etc), most of which were just condensed versions of already existing syntax. We hope the effect of circumstances which were beyond anyone’s control on the amount and quality of work we were able to accomplish will be taken into account when evaluating our project.

8.2 Outstanding Issues

Our type inference algorithm is not correctly determining the types of collection access variables. Our plan to fix this would be to add the types of these items to the symbol table and check in the semantic analysis code if the type in the symbol table was valid.

We have not fully tested our language's recursive features. Our plan to proceed here would be to add an extra table to our eval functions and check types of recursive functions separately from non-recursive functions.

9. Appendix

9.1 scaner.mll

```
{ open Parser }

rule token = parse
     [' ' '\t' '\r']                              { token lexbuf }
     (* Whitespace *)
     | "//"                                       { comment lexbuf }
     (* Comments *)

     (* Primitive types *)
     | ['0'-'9']+ as lxm                 { INT(int_of_string lxm)
}
     | ['0'-'9']+'.'['0'-'9']* as fl     {
FLOAT(float_of_string(fl)) }
     | ['0'-'9']*'.'['0'-'9']+ as fl     {
FLOAT(float_of_string(fl)) }
     | "true"                                     { BOOLEAN(true) }
     | "false"                                    { BOOLEAN(false) }
     | ('\"'([^'\\''\"']|'\\'['\\''n''t''\"'])*'\"') as st
{STRING(st)}

     (* Grouping symbols *)
     | '('                                        { LPAREN }
     | ')'                                        { RPAREN }
     | '{'['\r''\n''\t'' ']*                      { LCURL }

     (* Allow one newline *)
     | '}'                           { RCURL }
     | '['                                        { LBRAC }
     | ']'                                        { RBRAC }
```

```
 | '|'                                    { BAR }

 (* Punctuation *)
 | ','                                    { COMMA }
 | '.'                                    { DOT }
 | ':'                                    { COLON }
 | '='                                    { ASSIGN }

(* Inc/Dec Operators *)
| "++"                                    { INCREMENT }
| "--"                                    { DECREMENT }

 (* Arithmetic Operators *)
 | '+'                                    { PLUS }
 | '-'                                    { MINUS }
 | '*'                                    { TIMES }
 | '/'                                    { DIVIDE }
 | '%'                                    { MOD }

(* Update Operators *)
| "+="                                    { PLUSEQUALS }
| "-="                                    { MINUSEQUALS }
| "*="                                    { TIMESEQUALS }
| "/="                                    { DIVIDEEQUALS }

 (* Boolean comparison operators *)
 | "not"                                  { NOT }
 | "and"                                  { AND }
 | "or"                                   { OR }

 (* Comparison operators *)
 | "!="                                   { NEQ }
 | "=="                                   { EQ }
 | '<'                                    { LT }
 | ">"                                    { GT }
 | "<="                                   { LEQ }
 | ">="                                   { GEQ }

 (* Separators *)
 | '\n'['\n'' ''\t']*                     { EOL }
 | eof                                    { EOF }
```

```
      (* Keywords *)
      | "if"                                    { IF }
      | "elseif"                                { ELSEIF }
      | "else"                                  { ELSE }
      | "while"                                 { WHILE }
      | "for"                                   { FOR }
      | "return"                                { RETURN }
      | "rules"                                 { RULES }

      (* Built in functions *)
      | "CAST"                                  { CAST }
      | "INPUT"                                 { INPUT }
      | "OUTPUT"                                { OUTPUT }

      (* Type keywords *)
      | "int"                                   { INTTYPE }
      | "float"                                 { FLOATTYPE }
      | "string"                                { STRINGTYPE }
      | "bool"                                  { BOOLTYPE }

      (* Identifier *)
      | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm
      { ID(lxm) }

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

and comment = parse
  '\n'                                              { token
lexbuf }
| _                                                 { comment
lexbuf }
```

9.2 parser.mly

```
%{open Ast %}
```

```
%token<string> ID

/* Primitive types */
%token<int> INT
%token<bool> BOOLEAN
%token<float> FLOAT
%token<string> STRING

/* Grouping symbols */
%token LPAREN RPAREN RCURL LCURL RBRAC LBRAC BAR

/* Punctuation */
%token COMMA DOT COLON ASSIGN

/* Arithmetic Operators */
%token PLUS MINUS TIMES DIVIDE MOD

/* Boolean Operators */
%token NOT AND OR

/* Comparison Operators */
%token NEQ EQ LT GT LEQ GEQ

/* Update Operators */
%token PLUSEQUALS MINUSEQUALS TIMESEQUALS DIVIDEEQUALS

/* Separators */
%token EOL EOF

/* Keywords */
%token IF ELSEIF ELSE WHILE FOR RETURN RULES

/* Built in functions */
%token CAST PLAY INPUT OUTPUT

/* Type keywords */
%token INTTYPE FLOATTYPE STRINGTYPE BOOLTYPE

/* Increment/Decrement keywords */
%token INCREMENT DECREMENT
```

```
/* Associativity rules */
%right ASSIGN
%left EQ NEQ LT GT LEQ GEQ
%left AND OR
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left NOT
%left INCREMENT DECREMENT DOT

/* Start info */
%start program
%type <Ast.program> program
%%

arguments:
    /* no args */                                                    {[]}
|    ID
{[Identifier($1)]}
|    ID COMMA arguments
{Identifier($1)::$3}

identifier:
   ID
{Identifier($1)}

functionId:
    ID LPAREN arguments RPAREN
{FunctionIdentifier($1, $3)}

block:
    LCURL statementList RCURL
{Block($2)}

ruleBlock:
    expression block
{[RuleBlock($1, $2)]}
|    expression block ruleBlock
{RuleBlock($1, $2)::$3}

expressionList:
```

```
    /* empty list */                                    {[]}
|    expression
{[$1]}
|    expression COMMA expressionList                    {$1::$3}

structDef:
    ID COLON expression
{StructDef($1, $3)}

structBody:
    /* empty struct */                                      {[]}
|    structDef
{[$1]}
|    structDef COMMA structBody                         {$1::$3}

typeKeyword:
    INTTYPE
{"int"}
|    FLOATTYPE
{"float"}
|    STRINGTYPE
{"String"}
|    BOOLTYPE
{"boolean"}

binaryOperation:
    expression PLUS expression
{Plus($1,$3)}
|    expression MINUS expression
{Minus($1,$3)}
|    expression TIMES expression
{Times($1,$3)}
|    expression DIVIDE expression
{Divide($1,$3)}
|    expression MOD expression
{Mod($1,$3)}
|    expression AND expression
{And($1,$3)}
|    expression OR expression
{Or($1,$3)}
```

```
|    expression NEQ expression
{Neq($1,$3)}
|    expression EQ expression
{Eq($1,$3)}
|    expression LT expression
{Lt($1,$3)}
|     expression GT expression
{Gt($1,$3)}
|    expression LEQ expression
{Leq($1,$3)}
|    expression GEQ expression
{Geq($1,$3)}

expression:
      INT
{Integer($1)}
|    BOOLEAN
{Boolean($1)}
|    FLOAT
{Floating($1)}
|    STRING
{StringString($1)}
|    identifier
{IdExpr($1)}
|    LBRAC expression COLON expression RBRAC
{CollectionBounds($2, $4)}
|   LBRAC expressionList RBRAC
{Collection($2)}
|    BAR arguments BAR
{Enum($2)}
|    LPAREN expression RPAREN
{ParenExpr($2)}
|    LCURL structBody RCURL
{Struct($2)}
|    CAST LPAREN expression COMMA typeKeyword RPAREN
   {Cast($3, $5)}
|    INPUT LPAREN typeKeyword RPAREN
{Input($3)}
|    MINUS expression
{Negate($2)}
```

```
|     NOT expression
{Invert($2)}
|    binaryOperation
{BinaryOp($1)}
|    expression INCREMENT
{Increment($1)}
|    expression DECREMENT
{Decrement($1)}
|    identifier LBRAC expression RBRAC
{CollectionIndex($1, $3)}
|    identifier DOT expression
{Lookup($1, $3)}
|    functionId
{FunctionCall($1)}

statementList:
    statement
{[$1]}
|    statementList EOL                                    {$1}
|    statementList EOL statement                      {$3::$1}

idUpdate:
    identifier PLUSEQUALS expression              {PlusUp($1,
$3)}
|    identifier MINUSEQUALS expression
{MinusUp($1, $3)}
|    identifier TIMESEQUALS expression
{TimesUp($1, $3)}
|    identifier DIVIDEEQUALS expression
{DivideUp($1, $3)}

elseStatements:
    ELSE block
{[IfStatement("else", Boolean(true), $2)]}
|    ELSEIF expression block
{[IfStatement("else if", $2, $3)]}
|    ELSEIF expression block elseStatements
{IfStatement("else if", $2, $3)::$4}

assignmentList:
    /*empty*/                                             {[]}
```

```
|    assignment
{[$1]}
|    assignment COMMA assignmentList                    {$1::$3}

statement:
    expression
{Expression($1)}
|    assignment
{LocalVar($1)}
|    idUpdate
{Update($1)}
|    OUTPUT LPAREN expression RPAREN
{Output($3)}
|    RETURN expression
{Return($2)}
|    IF expression block
{IfStructure(IfStatement("if", $2, $3)::[])}
|    IF expression block elseStatements
{IfStructure(IfStatement("if", $2, $3)::$4)}
|    WHILE expression block
{WhileStructure($2, $3)}
|    RULES LCURL ruleBlock RCURL
{RuleStructure([], $3)}
|    RULES LPAREN assignmentList RPAREN LCURL ruleBlock RCURL
{RuleStructure($3, $6)}
|    FOR identifier COLON expression block
{ForStructure($2, $4, $5)}

assignment:
    identifier ASSIGN expression
{Assignment($1, $3)}
|    functionId ASSIGN block
{FunctionAssign($1, $3)}

program:
    assignment
{[$1]}
|    program EOL assignment
{($3::$1)}
```

9.3 ast.mli

```
type javaType =
  JavaVoid
| JavaBool
| JavaInt
| JavaFloat
| JavaString
| JavaInvalidType
| JavaCollection
| JavaEnumItem
| JavaEnum
| JavaStruct
| Indeterminate

type identifier =
    Identifier of string

type functionId =
    FunctionIdentifier of string * identifier list

type expression =
    Integer of int
|    Boolean of bool
|    Floating of float
|    StringString of string
|    IdExpr of identifier
|   Collection of expression list
|   CollectionBounds of expression * expression
|    Enum of identifier list
|    ParenExpr of expression
|    Struct of structDef list
|    Cast of expression * string
|   Input of string
|    Negate of expression
|   Invert of expression
|    BinaryOp of binaryOperation
|    Increment of expression
|   Decrement of expression
|    CollectionIndex of identifier * expression
|    Lookup of identifier * expression
```

```
|    FunctionCall of functionId

and structDef =
    StructDef of string * expression

and binaryOperation =
    Plus of expression * expression
|    Minus of expression * expression
|    Times of expression * expression
|    Divide of expression * expression
|    Mod of expression * expression
|    And of expression * expression
|    Or of expression * expression
|    Neq of expression * expression
|    Eq of expression * expression
|    Lt of expression * expression
|    Gt of expression * expression
|    Leq of expression * expression
|    Geq of expression * expression

type statement =
    Expression of expression
|    LocalVar of assignment
|   Update of update
|    Output of expression
|    Return of expression
|    IfStructure of ifStatement list
|    WhileStructure of expression * block
|    RuleStructure of assignment list * ruleBlock list
|    ForStructure of identifier * expression * block

and block =
    Block of statement list

and ruleBlock =
    RuleBlock of expression * block

and assignment =
    Assignment of identifier * expression
|    FunctionAssign of functionId * block
|    Empty
```

```
and update =
    PlusUp of identifier * expression
|    MinusUp of identifier * expression
|    TimesUp    of identifier * expression
|    DivideUp of identifier * expression

and ifStatement =
    IfStatement of string * expression * block

type program =
    assignment list
```

9.4 cardigan.ml

9.5 Makefile

```
cardigan_ : cardigan.ml ast.mli parser.mly scanner.mll
     ocamllex scanner.mll
     ocamlyacc parser.mly
     ocamlc -c ast.mli
     ocamlc -c parser.mli
     ocamlc -c scanner.ml
     ocamlc -c parser.ml
     ocamlc -c cardigan.ml
     ocamlc -o cardigan_ parser.cmo scanner.cmo cardigan.cmo

parser.cmi : ast.cmo

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

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

ast.cmo :
ast.cmx :

.PHONY : clean
```

```
clean :
     rm -f cardigan_ parser.ml parser.mli parser.output \
     scanner.ml testall.log *.cmo *.cmi *.out *.diff tests/
*.class\
     tests/*.diff tests/*.out tests/Test*.java tests/Fail*.java

.PHONY : test
test :
     ./testall


.PHONY : testk
testk :
     ./testall -k


.PHONY : all
all : clean cardigan test

.PHONY : allk
allk: clean cardigan testk
```

9.6 Java Libraries

cCollection.java

```
import java.util.Collections;
import java.util.LinkedList;

public class cCollection {
     public LinkedList<Object> collectionContainer = new
LinkedList<Object>();

     public cCollection(){
          this.collectionContainer=new LinkedList<Object>();
     }

     public cCollection(Object[] args){
          for (int i=0; i< args.length; i++){
               this.collectionContainer.add(args[i]);
          }
```

```
    }
    public cCollection(int low, int high){
        for (int i=low; i<high; i++){
            this.collectionContainer.add(i);
        }
    }

    //append(x) - Adds x to the end of the collection.
    public void append(Object x) {
        this.collectionContainer.add(x);
    }

    //push(x) -- Adds x to the head of the collection.
    public void push(Object x) {
        this.collectionContainer.push(x);
    }

    //pop() - Removes the first element of the collection and
returns it.
    public Object pop() {
        return this.collectionContainer.poll();
    }

    //peek() -- Returns but does not remove the first element
of the collection
    public Object peek() {
        return this.collectionContainer.peek();
    }

    //popLast() -- Removes and returns the last element of the
collection.
    public Object popLast() {
        return this.collectionContainer.pollLast();
    }

    //size() -- Returns the size of collection
    public int size(){
        return this.collectionContainer.size();
    }

    //get(index) -- Returns the element in collection
```

```
	public Object find(int index){
		return this.collectionContainer.get(index);
	}

	public void shuffle(){
		Collections.shuffle(this.collectionContainer);
	}
}
```

---

cEnum.java

```
import java.util.HashMap;

public class cEnum {
	public HashMap<String, EnumType> enumContainer = new
HashMap<String, EnumType>();

	public void add(String p, String id){
		EnumType enumType = new
EnumType(this.enumContainer.size()+1, id);
		this.enumContainer.put(p, enumType);
	}

	public EnumType find(String k){
		return this.enumContainer.get(k);
	}
}
```

---

cFunction.java

```
public interface cFunction {
	Object function();
}
```

---

cInput.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class cInput {
     public int cInputInt(){
          int result = 0;
          boolean valid = false;

          InputStreamReader isReader=new
InputStreamReader(System.in);
          String input = "";
          try {
               while(!valid){

                    input=new
BufferedReader(isReader).readLine();
                    try{
                         result=Integer.parseInt(input);
                         valid=true;
                    }catch(NumberFormatException e){
                         System.out.print("Invalid input, please
try again!\n");
                    }
               }
          } catch (IOException e) {

          }

          return result;
     }

     public float cInputFloat(){
          float result = 0;
          boolean valid = false;

          InputStreamReader isReader=new
InputStreamReader(System.in);
          String input = "";
          try {
```

```
               while(!valid){
                    input=new
BufferedReader(isReader).readLine();
                    try{
                         result=Float.parseFloat(input);
                         valid=true;
                    }catch(NumberFormatException e){
                         System.out.print("Invalid input, please
try again!\n");
                    }
               }
          } catch (IOException e) {

          }
          return result;
     }

     public String cInputString(){
          String result="";

          InputStreamReader isReader=new
InputStreamReader(System.in);
          try {
               result=new BufferedReader(isReader).readLine();
          } catch (IOException e) {

          }
          return result;
     }

     public boolean cInputBool(){
          boolean result = false;
          boolean valid = false;

          InputStreamReader isReader=new
InputStreamReader(System.in);
          String input = "";
          try {
               while(!valid){
                    input=new
BufferedReader(isReader).readLine();
```

```
                    if(input.equals("true")||
input.equals("false")){
                         result = Boolean.parseBoolean(input);
                         valid = true;
                    }
                    else{
                         System.out.print("Invalid input, please
try again!\n");
                    }
               }
          } catch (IOException e) {

          }
          return result;
     }
}
```

---

cStruct.java

```
import java.util.HashMap;

public class cStruct {
     public HashMap<String, Object> propertyContainer = new
HashMap<String,Object>();
     public HashMap<String, cFunction> functionContainer = new
HashMap<String,cFunction>();

     public void addProperty(String name, Object value){
          this.propertyContainer.put(name, value);
     }

     public Object findProperty(String name){
          return this.propertyContainer.get(name);
     }

}
```

```
EnumType.java

public class EnumType {
     public int enumValue;
     public String enumIdentifier;


     public EnumType(int value, String id) {
          // TODO Auto-generated constructor stub
          this.enumValue = value;
          this.enumIdentifier = id;
     }
}
```

9.7 Cardigan script (compilation)

```
#!/bin/sh

CARDIGAN="./cardigan_"

# Set time limit for all operations
ulimit -t 30

# Set script to exit immediately upon failure
set -e
Usage() {
    echo "Usage: testall [options] [.crd files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}


filename="$1"
basename=`echo $1 | sed 's/.*\\///
                              s/.crd//'`
reffile=`echo $1 | sed 's/.crd$//'`
basedir="`echo $1 | sed 's/\/[^\/]*$//'`/"
javafile=`echo $basename |sed -e 's/^//g' -e 's/-/_/g'`
ajavafile=`echo $javafile | perl -pe 's/\S+/\u$&/g'`
```

```
newjavafile=`echo $ajavafile | perl -pe 's/([^ ])_([a-z])/
\\1\\u\\2/g'`


rawjava=$newjavafile.java

make clean
make cardigan_
$CARDIGAN $filename
cd $basedir
# echo $rawjava
echo `javac $rawjava`
echo `java $newjavafile`
```