do fifty-two:
A Card Game Language

Final Report

Sinclair Target
Jayson Ng
Josephine Tirtanata
Yichi Liu
Yunfei Wang

Table of Contents

1. Introduction

do *fifty-two* is an imperative, procedural, statically typed language. In that sense, it is like C and its many progeny in the fundamentals of how it works; a *do* program consists of a series of statements organized into procedures that act upon variables. But the syntax of the language attempts to be less cryptic and more intuitive to the average non-programmer, while simultaneously incorporating several elements that make programming card games a breeze.

1.1 Motivation

Card game languages have been a popular choice among PLT students in the past. In 2008, one group created a language called “PCGSL,” for Playing Card Game Simulation Language. In 2009, another group created a language called “AIC,” for All In the Cards. And in 2012, a third group created a language called “Cardigan,” which may or may not have been well implemented relative to the others but certainly has the best name of the three.

Each of these teams tried to simplify the process of programming a card game by creating a domain-specific language that abstracts away the details common to all card games. But the problem, as this team sees it, is that these languages have in most cases hewed too closely to traditional C-style syntax to be useful. Only people who have already done some programming in a language like C or Java could learn the language in a reasonable amount of time, yet those same people are also capable of programming a card game quite easily by themselves, because programming a card game isn’t that difficult. If you’re using an object-oriented language, all you really need is a “Card” class with an integer for rank and maybe a string for suit. Put some instances of that class in an array and you’re ready to go. The effort that a language like “PCGSL” or “Cardigan” might save is small, and that saved effort is likely going to be offset by the effort required to learn the language in the first place.

1.2 Proposal

We propose a card game language targeted not at proficient programmers but at people who have never programmed before. For these people, programming a card game in something like C or Java would be prohibitively difficult, so it makes sense to give them an alternative language tailored to their inexperience and to the problem of creating a card game. The time they invest in learning this language is not time they might otherwise have spent coding, but rather an opportunity for them to grow more comfortable with the idiosyncrasies of computer programming. Therefore our language will be first and foremost a pedagogical tool. It will not try to compete as an “easy” option for a programmer who could code a card game in dozens of different ways. It will instead be a way to introduce students to programming as gently as possible in a familiar domain.

Our goals for the language then are as follows: *do fifty-two*, as we've chosen to call it, must be intuitive and high-level, so that students can grasp it without feelings of frustration or despair. But it must also be sufficiently related to more traditional programming languages to serve as a good stepping-stone to them.

2. Language Tutorial

2.1 Data Types

Primitive Types	**Meaning**
Number	Numbers are defined as a 64 bit signed integer with a range of –2,147,483,648 to 2,147,483,647. Which is similar to the integer class found in other languages such as Java. Due to the nature of card games, a Number is not allowed to be initialized as non-negative
String	String is a sequence of characters surrounded by double quotes. A string is made of an array of characters.
Boolean	Booleans are defined by true and false keywords. Booleans are considered their own type, thus when an expression is defined with both a boolean operator and a non-boolean variable it will create an error.

Composite Types	**Meaning**
Card	A data type representing a card. **Fields:** Number rank Number suit String desc
Set	A data type representing an ordered collection of cards, whether that collection is supposed to be a deck, hand, or something else. **Fields:** Number size

	String desc Card top Card bottom
Player	A data type representing a player, or possibly a dealer. **Fields:** Set hand String Desc

2.2. Operators

Operator	**Meaning**
+ - * /	Standard arithmetic operators. Integer arithmetic only. Division returns floor module. Standard precedence.
= != < > >= <=	Standard relational operators, except that the equivalence operator is = not ==. Standard precedence.
| & !	Logical operators and the unary NOT operator. No bitwise operators.
:	Assignment operator
t> b> <t <b	Prepend and append operators. Take a **Set** (see below)and a **Card** and either adds the **Card** to the front or the back of the **Set**. If **Card** is null, does nothing.
.	Dot operator, Accesses field within an variable
+	String concatenation operator

2.3 Variable Operations

Declare a new variable	**new *typeName variableName*: *value*** i.e. new *Number myInteger: 0*
Add new fields to existing variable	***typeName* has *fieldType* called *fieldName*** i.e. *Player* has *Number* called *score* now Player has a field "*score"*
Redefine default environmental variables	**configure *variableName*: *value*** i.e. configure numberOfPlayers: 2

2.4 Function Operations

Declare a function	***procedureName* with *Type parameterName:*** i.e. *Sum* with *Number n1* and *Number n2:* *<function body>*
Call a function	**do *procedureName* with *argumentName*** i.e. do *Sum* with *5* and *6*
Quit a loop/conditional statement	**do quit**

3. Language Manual

3.1 Data Types

Primitive Types	**Meaning**

Number	Numbers are defined as a 64 bit signed integer with a range of 0 to 2,147,483,647. Which is similar to the integer class found in other languages such as Java. Due to the nature of card games, a Number is not allowed to be initialized as non-negative
String	String is a sequence of characters surrounded by double quotes. A string is made of an array of characters.
Boolean	Booleans are defined by true and false keywords. Booleans are considered their own type, thus when an expression is defined with both a boolean operator and a non-boolean variable it will create an error.

Composite Types	**Meaning**
Card	A data type representing a card. **Fields:** Number rank Number suit String desc
Set	A data type representing an ordered collection of cards, whether that collection is supposed to be a deck, hand, or something else. **Fields:** Number size String desc Card top Card bottom
Player	A data type representing a player, or possibly a dealer. **Fields:** Set hand String Desc

Data in *do-fifty-two* is expressed in a finite and well defined set of data types. There are three primitive types in *do*, with several composite data types to facilitate the creation of card games. There are no floating point numbers because the rules of card games don't ever -- as far as we know -- call for fractional parts. Each of the composite

data types has a field called *desc*, which contains a string description of the data it contains. When an instance of a composite data type is passed as an argument to the output function (see below), the *desc* field is what is printed.

New data types cannot be created, but the existing data types can be extended. New fields can be added to an existing type like so:

typename has *typeName* called *variableName*

A type extension is global and should not appear inside a function. Meaning that, all instances of the first data type have a filled of the second data type with the given name. For example,

```
Player has Number called score
```

would make the field *Player_score* available throughout a program.

3.2 Lexical Conventions

3.2.1 Identifier

An Identifier is a sequence of letters, digits or underscores. The first character must be alphabetic. An underscore ("_") is not considered to be alphabetic. Upper and lower case letters are considered to be different.

3.2.2 Keywords

The following identifiers are reserved for use as keywords and may not be used.

- if
- else if
- else
- while
- for
- break
- continue
- until
- true
- false
- do
- quit
- with
- output
- new
- configure
- setup
- int
- boolean
- String

3.2.3 Literals

Literals are values written in a conventional form with a fixed value.
A **Number Literal** consists of an optional minus sign, followed by one or more digits within the range of an Integer.
A **Boolean Literal** represents boolean values for true or false.
A **String Literal** represents a series of characters.

3.2.4 New Line

Like in Python, a new line terminates a statement. A program must end with a new line.

3.2.5 Whitespace

Whitespace consists of any combination of *blanks* or *tab* characters. Whitespace is used like in Python to delimit blocks if it appears at the beginning of a line.

3.2.6 Punctuations

Punctuators, are characters that have their own syntactic and semantic significance; they are not operators or identifiers

Punctuator	Use	Example
{}	Statement list delimiter	if (Boolean) { statements }
()	Conditional parameter delimiter. (Expression precedence)	if (Boolean)

3.2.7 Comments

The characters "//" introduce a single line comment. For example:

```
// Hello World !
```

3.2.8 Operators

do includes most standard operators found in any programming languages, but changes a few so that their meaning is more obvious, and adds two specific to card games. An operator is a token that creates an operation on at least one operand, which in turn yields a result. the assignment operator is not an equals sign, because assignment and tests for equality are distinct concepts in programming, yet that fact is often lost on beginner programmers who—and quite rightly—get confused by the equals sign. An added benefit here is that the actual equality operator can be a single equals sign as opposed to two.

Operator	Meaning
+ - * /	Standard arithmetic operators. Integer arithmetic only. Division returns floor module. Standard precedence.
= != < > >= <=	Standard relational operators, except that the equivalence operator is = not ==. Standard precedence.
| & !	Logical operators and the unary NOT operator. No bitwise operators.
:	Assignment operator.
.	Dot operator. Accesses a field within an object.
t> b> <t <b	Prepend and append operators. Take a **Set** (see below)and a **Card** and either adds the **Card** to the front or the back of the **Set**. If **Card** is null, does nothing.
+	String Concatenation Operator

3.3 Control Flow Statements

do incorporates most of the common control-flow statements with the exception of *switch*. In addition, it incorporates an intuitive shorthand for expressing simple loops, which can be thought of as “multiplying” a series of statements by a number. The shorthand is redundant, because it could be replaced with a *for* loop. But in our experience many algorithms can be expressed using only the shorthand, so it might be introduced as a simpler form of looping to students not ready for the menacing syntax of a *for* loop, or for the bizarreness that is zero-based numbering.

3.3.1 The *if* Statement:

You can use the ***if* statement** to conditionally execute part of your program, based on the truth value of a given expression. Here is the general form of the ***if* statement**:

```
if (test1)
    then-statement1
else if (test2)
    then-statement2
else
    else-statement
```

If *test1* evaluates to true, then *then-statement1* is executed. if *test1* evaluates to false, but *test2* evaluates to true, then *then-statement2* is executed and neither *then-statement1* nor *else-statement* is. If *test1 and test2* evaluate to false, then *else-statement* is executed and *then-statement1 or then-statement2* is not. The *else if* clause and *else* clause are optional.
Example:

```
if (count = 5)
      count: count + 1
else if (count = 6)
      count: 12
else
      count: count - 1
```

If *count = 5* evaluates to true, then the statement *count: count + 1* is executed. If *count = 5* evaluates to false and count = 6 evaluates to true, then the statement *count: count + 1* is not executed, but the statement *count: count: 12* is executed. If neither *count = 5* nor *count = 6* is true, then the statement *count: count - 1* is executed.

3.3.2 The *while* Statement:

The ***while* statement** is a loop statement with an exit test at the beginning of the loop. Here is the general form of the ***while* statement**:

```
while (test)
    statement
```

The ***while* statement** first evaluates *test*. If *test* evaluates to true, *statement* is executed, and then *test* is evaluated again. *statement* continues to execute repeatedly as long as *test* is true after each execution of *statement*.
This example increments the integer from zero through nine:

```
new Number counter: 0
while (counter < 10)
    counter: counter + 1
```

3.3.3 The for Statement:

The ***for*** **statement** is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. It is very convenient for making counter-controlled loops. Here is the general form of the **for statement**:

```
for (initialize; test; step)
   statement
```

The ***for*** **statement** first evaluates the expression *initialize*. Then it evaluates the expression *test*. If *test* is false, then the loop ends and program control resumes after *statement*. Otherwise, if *test* is true, then *statement* is executed. Finally, *step* is evaluated, and the next iteration of the loop begins with evaluating *test* again.

Most often, *initialize* assigns values to one or more variables, which are generally used as counters, *test* compares those variables to a predefined expression, and *step* modifies those variables' values. Here is another example that prints the integers from zero through nine:

```
for (new Number x: 0; x < 10; x: x + 1)
   do output with "x = " + x
```

First, it evaluates *initialize*, which assigns *x* the value 0. Then, as long as *x* is less than 10, the value of *x* is printed (in the body of the loop). Then *x* is incremented in the *step* clause and the *test* re-evaluated.

All three of the expressions in a *for* statement are optional, and any combination of the three is valid. Since the first expression is evaluated only once, it is perhaps the most commonly omitted expression. You could also write the above example as:

```
new Number x: 1
for (; x <= 10; x: x + 1)
   do output with "x = " + x
```

In this example, *x* receives its value prior to the beginning of the ***for*** **statement.**

If you leave out the *test* expression, then the *for* statement is an infinite loop (unless you put a ***break*** **statement** somewhere in statement). This is like using 1 as *test*; it is never false.

This *for* statement starts printing numbers at 1 and then continues infinitely, always printing *x* incremented by 1:

```
for (new Number x: 1; ; x++)
   do output with "x = " + x
```

If you leave out the *step* expression, then no progress is made toward completing the loop—at least not as is normally expected with a *for* statement.

This example prints the number 1 over and over, infinitely:

```
for (new Number x: 1; x <= 10;)
   do output with "x = " + x
```

3.3.4 The *{ } * N* Statement:

The ***{ } * N* Statement** execute the statement(s) in the *{ }* for *N* times. The statements in *{ }* must be executable, and *N* must be a positive integer that is greater or equal to 1. Example:

```
{
   new Number x: 1
   do output with "x = " + x
   } * 10
```

This statement executes the statements *new Number x: 1* and *do output with "x = " + x* for 10 times.

3.3.5 The *break* Statement:

User can use the ***break* statement** to terminate a *while, for, { } * N* or *{ } until B* statement. Here is an example:

```
for (new Number x: 1; x <= 10; x: x + 1)
 {
    if (x = 8)
      break
    else
      do output with x
  }
```

That example prints numbers from 1 to 7. When *x* is incremented to 8, x = 8 is true, so the ***break* statement** is executed, terminating the for loop prematurely.

If you put a ***break* statement** inside of a loop statement which itself is inside of a loop statement, the *break* only terminates the innermost loop statement.

3.3.6 The *continue* Statement:

User can use the ***continue* statement** in loops to terminate an iteration of the loop and begin the next iteration. Here is an example:

```
for (new Number x: 0; x < 100; x: x + 1)
{
  if (x / 5 > 10)
    continue
  else
    do output with " x = " + x
}
```

If you put a ***continue* statement** inside a loop which itself is inside a loop, then it affects only the innermost loop.

3.4 Procedures

In *do,* functions are called “procedures.” They are not called “functions” for reasons that will shortly become clear. They might have been called “methods,” except they are not associated with objects. They might also have been called “subroutines,” but the term “subroutine”, as I’m sure you’ll agree, is by this point rather passé.

A procedure must be defined in order for a call to the procedure to make sense, but the definition can come anywhere in a source file outside of a block. This means that a procedure does not have to be defined before it is called. Procedures are not first-class objects and they cannot be nested. They do not have to be—and in fact cannot be—declared.

A procedure consists of a header and a body. The header specifies the name of the procedure along with the types and names of its parameters. A header has the following syntax:

```
procedureName with Type parameter1 and Type parameter2 and … :
```

procedureName is the identifier associated with the procedure. `with` is a keyword that separates the identifier from the list of parameters, which consists of an arbitrary number of variable declarations separated by the keyword `and`. The colon terminates the header and begins the block that contains the body of the procedure. If a procedure takes no arguments, `with` can be omitted.

A procedure can be called anywhere within a source file. A procedure call looks like this:

```
do procedureName with argument1 and argument2
```

`do` is a keyword that signals a procedure call. The identifier *procedureName* tells the compiler which procedure to call. Everything after the keyword `with` is part of the list of arguments. The arguments are separated by the keyword `and` just as the parameters are in a procedure header. If the arguments passed to a procedure do not match the parameter types in the procedure header, the compiler will throw an error.

Arguments are evaluated left to right. Again, if a procedure takes no arguments, `with` can be omitted.

You may have noticed that the syntax for a procedure header does not include its return type. That is because procedures cannot return anything. This is a quirk of *do*; there are no functions that map input to output. Procedures can only change the values of their parameters (which are passed by reference) and the values of global variables. They are not expressions. Conceptually, a procedure can only *do* something—it can only act on data to produce a change in state.

do is thus a hyper-imperative language. A *do* program consists mostly of statements with only a few expressions here and there. For large programs, this would become a nightmare. But while the programs remain simple, we believe that using procedures instead of functions is a good thing—it better conforms to the layperson's understanding of programming as "telling the computer to do things."

3.5 Expressions

3.5.1 Arithmetic Expressions

An arithmetic expression consists operands and operators. The operands can be integer constants or variable of the Number type. Only integer arithmetic are allowed. Example of expressions:

```
2 + 2
3 - 1
4 * 5
10/2
```

Expressions can be surrounded with parenthesis to indicate order of evaluation. For example:

```
2 * (4 + 5)
```

The expression `4 + 5` will be evaluated first, resulting in `9`. Then, `2 * 9` will be evaluated, resulting in `18`.

There are 4 types of arithmetic operators.

- `+`

Adds the two operands together.

- `-`

Subtract the right operand from the left operand.

- `*`

Multiply the two operands together.

- `/`

Divide the operand on the left by the operand on the right.

3.5.2 Assignment Expressions

The assignment operator `:` stores the value of its right operand in the variable specified by the left operand. Example:

```
Number n : 5 * 2
```

3.6 Operators

3.6.1 Relational operators

You can use relational operators to determine how two operands relate to one another. Do Fifty-Two supports 6 types of comparisons.

Relational Operator	Description	Example

`=`	Checks for equality	`if (a=3)`
		`  do evaluate`
		`else`
		`  quit`

`!=`	Checks for inequality	`if (a!=3)`
		`  do evaluate`
		`else`
		`  quit`

`<`	Checks whether left operand is smaller than the right operand	`if (a<3)`
		`  do evaluate`
		`else`
		`  quit`

`>`	Checks whether left operand is greater than right operand	`if (a>3)`
		`  do evaluate`
		`else`
		`  quit`

`<=`	Checks whether left operand is equal or smaller than right operand	`if (a<=3)`
		`  do evaluate`
		`else`
		`  quit`

`>=`	Checks whether left operand is equal or greater than right operand	`if (a>=3)`
		`  do evaluate`
		`else`
		`  quit`

3.6.2 Logical operators

Logical operators can be used to negate or combine relational expressions.

The logical conjunction operator `&` tests if two expressions are both true. If the first expression is false, the second expression is not evaluated and the entire expression becomes false.

```
if((a>3)&(a<5))
     do output with "a is 4"
```

The logical conjunction operator `|` test if at least one of the two expressions are true.

```
if((a=3)|(a=5))
     do output with "a is either 3 or 5"
```

The prepend operator `!` tests if the logical expression equates to false.

```
if(!(a=3))
     do output with "a is not 3"
```

3.6.3 Field Accessor Operator

The field accessor operator _ is attached after an object name and fetches the field whose name is specified on the right side of the underscore. It returns the value of the field. The type depends on the type that the field is set to initially. Example:

```
     c.deck_top     //access the field deck.top of a Card
variable called c
```

The field accessor operator fetches the field named `top` in the object `deck`. Since the `deck` consists of `Card` types, this expression returns a `Card` type.

3.6.4 Prepend and Append Operators

The prepend and append operators `t>, b>` and `<t, <b` respectively takes a `Card` and places it to a `Set`. In both cases, if a `Set` is taken as the source, the operators will simply take the top most `Card` from the `Set`.

The prepend operator `t>` or b> takes the bottom `Card` or the top most `Card` from the `Set` in the left operand and places it in the back of the `Set` in the right operand.

The append operator `<t or <b` takes the `Card` at the bootom or the top most `Card` from the `Set` in the right operand and places it in the front of the `Set` in the left operand.

3.6.5 String Concatenation

The operator + when surrounded with strings will immediately concatenate the left and right string operands. Example:

```
do output with winner + "has lost"
```

3.7 Program Structure and Scope

3.7.1 Program Structure

A do-fifty-two program should be entirely self-contained, i.e. all the program code should be contained within a single file. No library or code import is supported.

A program should start with environmental variable assignments (by using `configure)` and type extensions if any, before any procedures are called or declared.

3.7.2 Scope

Variable declarations made at the top-level of a program (i.e., not within a procedure) are visible to the entire program, including from within procedures. Variable declarations made within procedures are visible only within those procedures. Variable declarations are not visible to the code that comes before them.

Procedure declarations can only happen at the top-level of a program. However, procedures can be called before their declarations.

Also, environmental variables and type extensions are global.

3.8 Sample Program

War! is a simple card game often played between children. The rules can be found here. What follows is an implementation of War! in do that showcases most of the features of the language.

```
// war in do
configure numberOfPlayers: 2
configure highestCard: 11

new Number warCount: 0

Player has Set called table  // player_table available

setup:
     // deal cards
     { player1_hand <t deck_top } * (deck_size / 2) //loop
     { player2_hand <t deck_top } * (deck_size / 2) //loop

round:
     do turn with player: player1
     do turn with player: player2
     do output with "Player 1 played: " + player1_table_top
     do output with "Player 2 played: " + player2_table_top
     do evaluate

turn with Player player:
     do output with player + "'s turn."
     new var
     if (player_hand_size = 0)
          if (player = player1)
               do output player + " has lost. Player2
wins!"
               do quit
          else
               do output player + " has lost. Player1
wins!"
               do quit

     do output with "Play card?"
     do input with new String in
     if (in = "y")
          player_hand_top t> player_table
     else
          do output with player + " has decided not to play
     anymore.” // no backslash needed here
          do quit
```

```
evaluate:
     if (player1_table_top > player2_table_top)
          do output with "Player 1's card is higher."
          { player1_hand <t player1_table_top } \
          * player1_table_size
          { player1_hand <t player2_table_top } \
          * player2_table_size
     else if (player1_table_top < player2_table_top)
          do output with "Player 2's card is higher."
          { player2_hand <t player1_table_top } \
* player1_table_size
          { player2_hand <t player2_table_top } \
* player2_table_size
     else
          do output with "It's a tie. That means WAR!"
          warCount: warCount + 1

     // if a set runs out of cards t> and <t won't do
anything
          { player1_hand_top t> player1_table } * 4
          { player2_hand_top t> player2_table } * 4
          do output with "Player 1 and 2 put down 4 cards."
          do evaluate
```

4. Project Plan

4.1 Identify process used for planning, specification, development and testing

4.1.1 Planning

Ocaml being a brand new language for all the team members, consumed most of the time and effort of the team at the early stage of the project. The team started Lexer successfully before they meet some obstacles when developing Parser, AST and Semantic Check. The team set the minimum goal of the project and build more feature upon it. Regarding the deliverables, the team managed to submit all the deliverables on time as a write-up report closely follows the progress of program development. During the weekly meeting, five group members are split into 2 or 3 groups to tackle different problems, and the solutions will be put together to make actual progress of the project.

4.1.2 Specification

As time limited, our main goal is to be able to compile the “War!” program that is included in the LRM, and the “War!” program reflects most of the features of Do Fifty-Two, we concentrated on writing and fixing errors of Parser, AST, SAST and Semantic Checking, along with other helper files including the Java simulation program, indentation, cache, stdlib, and Makefile.

4.1.3 Development

Most part of the development was done during the weekly meetings, along with many individual works during the second half of the semester. Compiler implementation is done in Ocaml, Target language is Java in a pythonic style.

4.1.4 Testing

Testing cases are mainly done by Jayson Ng with some others’ contributions, Jayson received program updates from all other group members, wrote the corresponding testing cases and did all the testings from small,simple cases to more complicated cases. All the testing logs are recorded and stored in the project folder.

4.1.5 Programming style guide used by the team

The team do not adopt a specific named programming style guide, but we create and follow the rules below to ensure the progress and cooperation:

1. Always communicate to the team before creating a new file, making major update and after meeting with TA, assuring the progress made is acknowledged
2. Keep the code clean and readable; introduction is given at the beginning of each file; comments are not required but strongly encouraged
3. When a code issue is unsure, always leave a comment before the team makes a decision regarding this issue.
4. Always submit a commit message if any change is committed to gitHub
5. All the tasks are distributed to each team member through Trello to keep track of each individual’s work
6. Filenames are readable and reflects the functionalities of the files
7. All files are classified and store in proper subdirectories (i.e. testing cases are stored in “test” folder, documentations are stored in “documentation” folder)

4.2 General Estimated Project Timeline

Time	**Event**
Sep 8 - Sep 15	Project Team Formed

Sep 16 - Sep 24	Topic Selected, Tools Ready, Proposal Submitted
Sep 25 - Oct 1	Meeting with TA, Proposal Feedback Returned
Oct 2 - Oct 9	Lexer in Progress, Learning Ocaml
Oct 10 - Oct 27	Keep Studying Ocaml, Meet with TA for Lexer and Parser development, LRM Submitted
Oct 27 - Nov 17	More Features Added to Parser & AST, More Test Cases Added
Nov 17 - Nov 30	Added Sample Program to test cases, All Leftover Errors Fixed, Printed outputs by adding Pretty Printing
Dec 1 - Dec 12	Worked on Semantic Checking and SAST, Compiler Finished, Java Program Generated, Test Cases Gone Over
Dec 12 - Dec 15	Final Report Written, Presentation Tasks Distributed, fix all the leftover errors/bugs
Dec 16	Presentation

4.3 Team Roles and Responsibilities

Role	Responsibility	Member
Project Manager	Timely Completion of Deliverables, Coding work	Yunfei
Language	Language Design, Coding work	Sinclair
System Architect	Compiler Architecture, ENV, Coding work	Josephine
Verification & Validation	Test Plan, Test Suites, Coding work	Jason
System Integrator	Defines System Platform,	Yichi

	Makefile and makes sure components interoperate, Coding work	

All team members meet every Monday & Wednesday to discuss project plans, project progress and write code together, all the files are done by multiple team members.

4.4 Software development environment, tools and languages

- Platform and IDE: Mac OS X, Windows 7, Linux (Ubuntu)l, Vim, Emacs, Javac Compiler
- Language: Ocaml, Java
- Version Control: GitHub
- Collaboration Tools: Trello, Slack

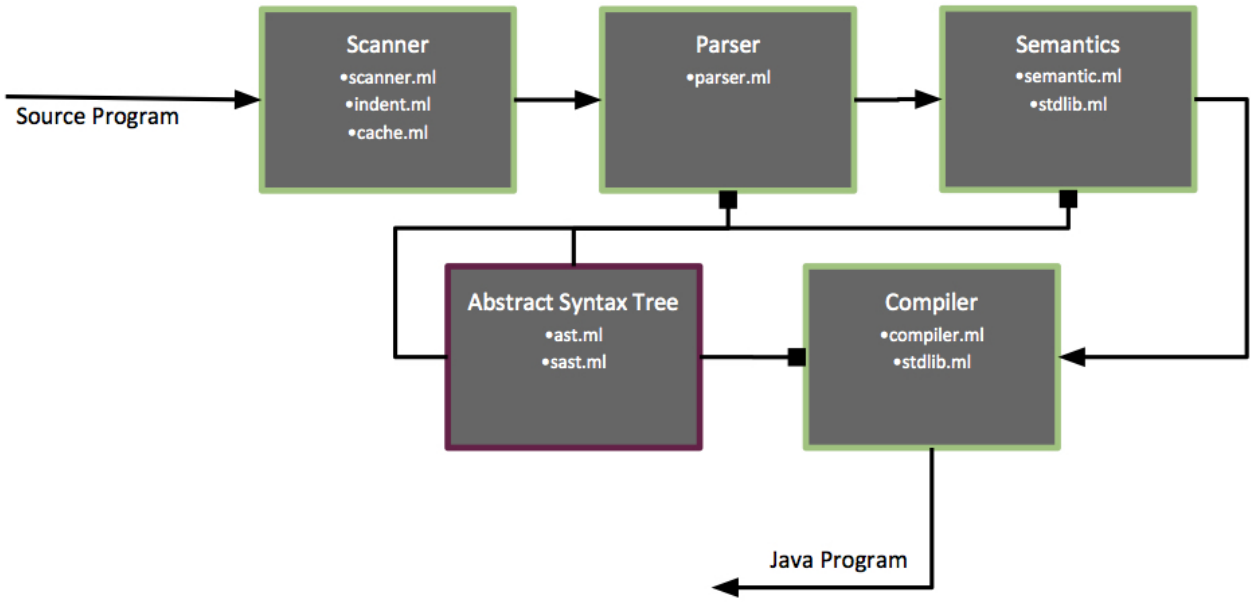
4.5 Project log

Our complete commit history can be found here, on GitHub:
https://github.com/YichiLL/4115PLT

A statistical breakdown of commits can also be found at the following link, which we are providing with the caveat that it may not be completely accurate. One of us didn't link his GitHub account to the repo until late in the semester, which meant that commits he made weren't attributed to him. Others contributed in ways that aren't reflected in the commit numbers, and some of us liked to commit after even the tiniest of changes.

https://github.com/YichiLL/4115PLT/graphs/contributors

5. Architectural Design



5.1 Interface between Components

The architecture of the Do Fifty-Two compiler consists of several functional stages that takes a *do Fifty-Two* source program and produces a Java executable card game. The relationship of these components is shown in the diagram above.

The scanning stage of the compiler turns the stream of characters from the source program into recognizable tokens for the parser. Scanner is supported by cache.ml that caches the number of DEDENT tokens as well as functions from indent.ml that help implement the python-style indentation-based blocks. By the time scanner is done, redundant whitespace and comments are removed, meaningful whitespace indentation are converted to DEDENT and INDENT tokens, and all characters are converted to tokens.

The parser uses ast.ml to assemble the sequence of tokens from the scanner into an abstract syntax tree (AST). It is responsible for analyzing the sequence of tokens and producing a structure that maps out the complete grammar of the language.

In Semantics, the structure of the abstract syntax tree is semantically checked. This includes attaching type information to the syntax tree and recognizing mismatch type declarations, input of illegal arguments in default functions, and incomplete setup of a correct do program. Semantic.ml is supported by stdlib.ml, a list of default functions

and variables that are supported by our language. Semantics allow the compiler to convert the do program to the equivalent java program and ensuring that the compiler has a comprehensible and acceptable grammar to work with.

With a semantically checked AST, the compiler is able to match configure declarations and functions into the correct java functions. With the list of functions and field declarations in stdlib.ml, the compiler can differentiate between user defined functions from default functions. This maps out to different java codes. When the compiler's work is completed, Game.java and MyPlayer.java are produced.

The architecture of the java class is presented in the following table:

<table>
<tr><td rowspan="2">Card
A card contains a face number, a value, a suit and a suit value.</td><td>Set
A set is an ordered group of cards. Set functions are fixed: shuffle, append, prepend, select, and peek.</td><td>Player
A player has a hand.</td><td>Game
The game class lists out the functions that are defined by the user. The <code>setup()</code> and <code>round()</code> methods must be defined.</td><td rowspan="2">Main
The main function creates an instance of a game and invoke <code>setup()</code>. After setup completes, it calls <code>round()</code> in an infinite loop.</td></tr>
<tr><td>Deck
A deck extends a Set and creates a deck of French Playing cards (Standard 52 cards with 4 suits)</td><td>MyPlayer
My player extends player and has additional instance variables that are defined by the source program</td><td>Utility
The utility class aids the game class by supplying functions for logical operations and I/O</td></tr>
</table>

Only the classes that are in Italics are modifiable in relation to the abstract syntax tree that is parsed from the source code.

5.2 Code Contribution

Scanner	scanner.ml	Josephine Tirtanata, Jayson Ng, Yunfei Wang, Sinclair Target, YIchi Liu
	indent.ml	Sinclair Target

	cache.ml	Sinclair Target
Parser	parser.ml	Josephine Tirtanata, Jayson Ng, Yunfei Wang, Sinclair Target, YIchi Liu
Abstract Syntax Tree	ast.ml	Josephine Tirtanata, Jayson Ng, Yunfei Wang, Sinclair Target (pretty printing), YIchi Liu
	sast.ml	Sinclair Target
	printer.ml	Sinclair Target
Semantic	semantic.ml	Sinclair Target
	stdlib.ml	Sinclair Target, Yunfei Wang
Compiler	compiler.ml	Sinclair Target, Josephine Tirtanata
Runtime Files	Player.java	Josephine Tirtanata
	Utility.java	Josephine Tirtanata
	Card.java	Josephine Tirtanata
	Deck.java	Josephine Tirtanata
	Set.java	Josephine Tirtanata
Test	Success Test Cases (*.do, *.gold)	Jayson Ng
	Failure Test Cases (*.do, *.gold)	
	Automated Test Scripts (testLogic.sh, testParse.sh)	

6. Test Plan

Test Plan:
The basic test plan involved three phase testing. The first two phases involved the entire team to test as we included more features to the compiler, while the last phase

involved manual testing. Phase one focused on testing the parser / semantics of do-fifty-two. Phase two tested the logic of the generated files. Finally, the last phase tested a fully programmed game in do-fifty-two.

Representative Programs:
Refer to Appendix - Test Code for representative programs written in do that were used for testing purposes.

Testing:
Test suits were developed in response to the complexities of the language. Each component was heavily tested to reveal the underlying inconsistencies and technicalities that were not easily noticeable. Once these issues were identified and dealt with several more rounds of testing were issued to ensure that all known bugs have been resolved and identified potential bugs.

Testing followed several stages of the development process. The first stage involved testing the AST and parser to ensure that the files created are syntactically correct and are accepted by the compiler. Afterwards logic was heavily tested to determine whether it translated properly from our language (do-fifty-two) to java. Finally we tested a fully flushed out “game” written in do-fifty-two is properly translated into java. Shell scripts were used to run all logic and parsing test cases.

Test Suites:

- Parser
  Automated tests created for parsing, scanning and creation of AST. Tests were chosen based on all features included in the AST and SAST.
- Logic
  Automated tests created to determine whether logic is properly translated. Tests were chosen based on all features related to logic.
- Game
  Created to determine whether a created game in do-fifty-two is as intended. Since this test case requires human interaction it is difficult to create an automated test script for it; as such manual testing was performed.

Tests Cases:

- ADD.do
- AND_COMPLEX_FALSE.do
- AND_FALSE.do
- AND_OR_COMPLEX.do
- AND_TRUE.do
- APPEND.do
- BREAK_FOR.do
- BREAK_WHILE.do
- CARD_COMPARISON.do
- COMMENTS.do
- COMPLEX_EXPRESSION.do

- COMPLEX_EXPRESSION_NO_PAREN.do
- CONFIGURE.do
- CONTINUE_FOR.do
- CONTINUE_FOR_NESTED.do
- CONTINUE_WHILE.do
- CONTINUE_WHILE_NESTED.do
- DIV.do
- DIV_INT.do
- DOT_OP.do
- EMPTY_LINE.do
- EQUAL.do
- FALSE.do
- FIELD_DECL.do
- FOR_LOOP_ASC.do
- FOR_LOOP_DESC.do
- GT_FALSE.do
- GTE_FALSE.do
- HELLO_WORLD.do
- IF_BASIC.do
- IF_NESTED.do
- IF_WITH_PARENTH.do
- LT.do
- LTE.do
- MINUS.do
- MULTI_VAR_DEC.do
- MULTIPLE_INDENT.do
- NOT_EQUAL.do
- OR.do
- OR_TRUE.do
- OR_FALSE.do
- PLAYER_COMPARISON.do
- PREPEND.do
- TIMES.do
- TIMES_LOOP.do
- VAR_DEC.do
- WHILE.do
- WHILE_RUN_ONCE.do
- FAIL_COMPLEX_EQUAL.do
- FAIL_MULTI_VAR_DEC_MISMATCH_TYPE.do
- FAIL_NO_ROUND.do
- FAIL_NO_SETUP.do
- FAIL_REDECL.do
- FAIL_TYPE_CHECK.do
- FAIL_UNDECL_ID.do

7. Lessons Learned

7.1 Team Lessons Learned

Yunfei Wang: A compiler is far more than a language parser or token library. Given the fact that this is the first time of us writing a compiler, choice of topic and target language is important because you always want to keep a straightforward understanding of what you do before you do more advanced and complicated compilers, since putting scanner, parser, semantic checks, ASTs together with the actual compiler could have more issue than you suppose. Working with group members is the key of this project, there is always someone who can help you to understand, and someone who you can help to understand; eventually the team will put puzzle pieces together little by little. Starting early and keeping in touch with TA are also very important, since this is a time consuming project, and there's a chance that you can get stuck and need directions. Using multiple tools for communications and version control is very important to keep everybody updated. The other one of the lessons that I learned is that although not required, having same development environment will be a great idea, given the fact that I myself had compilation problems because I was using Clic Machine (Linux) and my team members were using Mac OS X as files are compiled on Mac while not on Linux. Using integrated platform will prevent you from the situation that everybody's files compile while yours do not.

Jayson Ng: Given that it was my first time writing a compiler I found that it was difficult to properly comprehend the scope of the project without several small trial runs. Writing the AST, Parser, Scanner proved to be challenging but also rewarding. Although I did not contribute much to the final coding of the core components of the compiler I participated at the start of the project to gain a basic understanding of the overall structure and process. Throughout the entire project, I kept up with the coding and thought process of the components, in order to develop test cases for the finished project. From my experience testing is a crucial part of the development process since it helps identify bugs at an early stage. Scheduling regular meetings to work on this project was a great strategic move on our part simply because it allowed us to have regular check points and progress.

Sinclair Target: We had trouble splitting up the work. There were five of us collaborating on this project, but especially in the beginning, we couldn't find five different things for us to do. We wasted time watching others code or dealing with source control problems when multiple people edited the same file at once. Ideally, we would have assigned a module to each person and made that person responsible for it. That would have allowed us to work in parallel rather in series, as we mostly did. To put it another way, there weren't enough "black boxes." Most of us worked on the same thing at the same time, and most of us had to understand how the whole compiler worked, not just our part of it.

Josephine Tirtanata: Building a compiler was really hard to start with because I didn't have a good understanding of what a compiler is. Furthermore oCaml is a very difficult language, one that is not similar to any of the languages I know. These issues made it difficult to start on the project and to divide the project into dividable sections. I also

think that an important aspect of this project is to make sure you have a good idea of the language you want to make. It would be beneficial if you have a clear understanding of the language you want to make, so that you're able to interpret the source code into a meaningful target program. After completing this project, I believe I am so much better in functional programming and a lot more comfortable with the idea of working in a group.

7.2 Advice for Future Team

1. Start as early as you can, always keep adding code
2. Version control is important, make sure you use Git/SVN
3. Communicate with TAs and the professor
4. Know your language well
5. Be comfortable with oCaml
6. Using same environment and IDE is preferable
7. Team work, help each other on the code and everything
8. Meet at least twice a week
9. Write good final report

8. Appendix

Test Code

war.do

```
configure numberOfPlayers: 2
configure highestCard: ace
configure ascendingOrder: true

Player has Set called table

new Number warCount: 0

setup:
        new Number deckSize : deck.size
        { player1.hand <t deck } * (deckSize / 2)
        { player2.hand <t deck } * (deckSize / 2)
```

```
round:
	do turn with player1
	do turn with player2
	do output with "Player 1 played: " + player1.table.top.desc
	do output with "Player 2 played: " + player2.table.top.desc
	do evaluate

turn with Player player:
	do output with player.desc + "'s turn."
	if player.hand.size = 0:
		if player = player1:
			do output with player.desc + " has lost. Player2 wins!"
			do quit
		else:
			do output with player.desc + " has lost. Player1 wins!"
			do quit

	do output with "Play card?"

	new String in : ""
	do input with in
	if in = "y":
		player.hand t> player.table
	else:
		do output with player.desc + " has decided not to play anymore."
		do quit

evaluate:
	new Boolean done : false

	while !done:
		if player1.table.top > player2.table.top:
			do output with "Player 1's card is higher."
			{ player1.hand <t player1.table } * player1.table.size
			{ player1.hand <t player2.table } * player2.table.size
			done : true
		else:
			if player1.table.top < player2.table.top:
				do output with "Player 2's card is higher."
				{ player2.hand <t player1.table } * player1.table.size
				{ player2.hand <t player2.table } * player2.table.size
				done : true
```

```
        else:
            do output with "It's a tie. That means WAR!"
            warCount: warCount + 1

            { player1.hand t> player1.table } * 4
            { player2.hand t> player2.table } * 4
            do output with "Player 1 and 2 put down 4 cards."
```

war.do - Game.java

```
import java.util.Scanner;
import java.util.ArrayList;

public class Game {
ArrayList<MyPlayer> players;
Set deck;
int numberOfPlayers = 4;
int highestCard = 12;
boolean ascendingOrder = true;
int warCount = 0;
public Game() {
numberOfPlayers = 2;
highestCard = Card.ACE;
ascendingOrder = true;deck = new Deck(highestCard, ascendingOrder);
deck.shuffle();
players = new ArrayList<MyPlayer>();
for(int i = 0; i < numberOfPlayers; i++) {
players.add(new MyPlayer("Player " + (i+1)));
}
}

public void setup()
{
int deckSize = deck.size();
for (int i = 0; i < (deckSize / 2); i++)
{
Set.append(deck, Set.TOP, players.get(0).hand);

}

for (int i = 0; i < (deckSize / 2); i++)
{
```

```
Set.append(deck, Set.TOP, players.get(1).hand);

}
}

public void round()
{
turn(players.get(0));
turn(players.get(1));
System.out.println(("Player 1 played: " + players.get(0).table.top().toString()));
System.out.println(("Player 2 played: " + players.get(1).table.top().toString()));
evaluate();}

private void turn( MyPlayer player)
{
System.out.println((player.toString() + "'s turn."));
if ((player.hand.size() == 0))
{
if ((player == players.get(0)))
{
System.out.println((player.toString() + " has lost. Player2 wins!"));
System.exit(0);}
else
{
System.out.println((player.toString() + " has lost. Player1 wins!"));
System.exit(0);}
}
else
{
}

System.out.println("Play card?");
String in = "";
in = Utility.inputString();
if ((Utility.compareString(in, "y")))
{
Set.prepend(player.hand, Set.TOP, player.table);
}
else
{
System.out.println((player.toString() + " has decided not to play anymore."));
System.exit(0);}
}
```

```
private void evaluate()
{
boolean done = false;
while (!done)
{
if ((Utility.cardGreaterThan(players.get(0).table.top(), players.get(1).table.top())))
{
System.out.println("Player 1's card is higher.");
for (int i = 0; i < players.get(0).table.size(); i++)
{
Set.append(players.get(0).table, Set.TOP, players.get(0).hand);

}

for (int i = 0; i < players.get(1).table.size(); i++)
{
Set.append(players.get(1).table, Set.TOP, players.get(0).hand);

}

done = true;}
else
{
if ((Utility.cardLessThan(players.get(0).table.top(), players.get(1).table.top())))
{
System.out.println("Player 2's card is higher.");
for (int i = 0; i < players.get(0).table.size(); i++)
{
Set.append(players.get(0).table, Set.TOP, players.get(1).hand);

}

for (int i = 0; i < players.get(1).table.size(); i++)
{
Set.append(players.get(1).table, Set.TOP, players.get(1).hand);

}

done = true;}
else
{
System.out.println("It's a tie. That means WAR!");
```

```
warCount = (warCount + 1);
for (int i = 0; i < 4; i++)
{
Set.prepend(players.get(0).hand, Set.TOP, players.get(0).table);

}

for (int i = 0; i < 4; i++)
{
Set.prepend(players.get(1).hand, Set.TOP, players.get(1).table);

}

System.out.println("Player 1 and 2 put down 4 cards.");}
}
}
}
}
```

war.do - MyPlayer.java

```
public class MyPlayer extends Player {
Set table;
public MyPlayer(String playerName) {
super(playerName);
table = new Set();}
}
```

HELLO_WORLD.do

```
setup:
        do output with "Hello, World!"

round:
        do quit
```

HELLO_WORLD.do - Game.java

```
import java.util.Scanner;
import java.util.ArrayList;

public class Game {
ArrayList<MyPlayer> players;
Set deck;
int numberOfPlayers = 4;
```

```
int highestCard = 12;
boolean ascendingOrder = true;

public Game() {
deck = new Deck(highestCard, ascendingOrder);
deck.shuffle();
players = new ArrayList<MyPlayer>();
for(int i = 0; i < numberOfPlayers; i++) {
players.add(new MyPlayer("Player " + (i+1)));
}
}

public void setup()
{
System.out.println("Hello, World!");}

public void round()
{
System.exit(0);}
}
```

HELLO_WORLD.do - MyPlayer.java

```
public class MyPlayer extends Player {
public MyPlayer(String playerName) {
super(playerName);
}
}
```

testLogic.sh

```
#!/bin/sh

DO_FIFTY_TWO="./compile"
JAVA="javac"
RUNTIME="runtime/"
MAIN="main"

# Set time limit for all operations
ulimit -t 30

globallog=testLogic.log
globalfaillog=testLogicFAIL.log
```

```
rm -f $globallog
rm -f $globalfaillog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testLogic.sh [options] [.do files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

Compiler(){
    Run "make"
}

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

SignalErrorFail() {
    if [ $error eq 0 ] ; then
    echo "$1 failed to fail"; else
    echo "OK - $* failed"
    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"
        echo "FAILED $1 differs from $2" 1>&2
    }
```

```
}

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

# Run <args>
# Report the command, run it, and report any errors
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalErrorFail "$1 failed on $*"
    return 0
    }
}

CompileRunTime(){
    Run "javac -g MyPlayer.java" &&
    Run "javac -g Game.java" &&
    Run "javac -g Utility.java" &&
    Run "javac -g main.java" &&
    Run "javac -g Card.java" &&
    Run "javac -g Deck.java" &&
    Run "javac -g Player.java" &&
    Run "javac -g Set.java"
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\//
                        s/.do//'`
    reffile=`echo $1 | sed 's/.do$//'`
    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 "$DO_FIFTY_TWO" $1 &&
    Run "mv Game.java MyPlayer.java $RUNTIME" &&
    Run "cd runtime/" &&
    CompileRunTime &&
    Run "java -cp . $MAIN >" ../tests/${basename}.out &&
    Run "make clean" &&
    Run "cd .." &&
    Compare tests/${basename}.out tests/${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 tests/*.out
    fi
    echo "OK - $basename succeeds"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\//
                        s/.do//'`
    reffile=`echo $1 | sed 's/.do$//'`
    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="$generatedfiles tests/test_failure/${newjavafile}.java
tests/test_failure/${basename}.diff tests/test_failure/${basename}.out" &&
    RunFail "$DO_FIFTY_TWO" $1 ">" tests/test_failure/${basename}.out

    # Report the status and clean up the generated files

    if [ $error -lt 1 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f tests/test_failure/${basename}.out
    fi
    echo "OK - $basename succeeds"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}


shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/*.do"
    failfiles="tests/test_failure/*.do"
fi
for file in $files
do
    case $file in
    *)
        Check $file 2>> $globallog
        ;;
    esac
done
for file2 in $failfiles
do
    case $file2 in
    *)
        CheckFail $file2 2>>  $globalfaillog
        ;;
    esac
```

```
done

exit $globalerror
```

testParse.sh

```
#!/bin/sh

DO_FIFTY_TWO="./printer"

# Set time limit for all operations
ulimit -t 30

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

keep=0

Usage() {
   echo "Usage: testParse.sh [options] [.do 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"
         echo "FAILED $1 differs from $2" 1>&2
   }
```

```
}

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

# Run <args>
# Report the command, run it, and report any errors
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalErrorFail "$1 failed on $*"
    return 0
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\//
                        s/.do//'`
    reffile=`echo $1 | sed 's/.do$//'`
    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 "$DO_FIFTY_TWO" $1 ">" tests/${basename}.out
    #Compare 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
}

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/*.do tests/*.do"
fi

for file in $files
do
    case $file in
    *)
        Check $file 2>> $globallog
        ;;
    esac
done

exit $globalerrorexit $globalerror
```

README.txt

A compiler for Do, a card game programming language designed for
first-time programmers.

Final project for COMS 4115, Programming Languages and Translators.

Scanner.mll

```
(* scanner.mll groups characters read from input into tokens that are then
 * passed to the parser *)

{
    open Parser
    open Indent
    open Printf

    (* Persistent reference cell counter for the current indent depth. *)
    let cur_depth = ref 0

    (* This function returns INDENT or DEDENT tokens whenever we change depth.
     * DEDENT means we've reached the end of a block. INDENT means we've
     * entered one. In OCaml ";" has lower precedence than "if", hence all
     * the begins and ends.
     *
     * If we dedent more than 1 level, we need to produce a token that holds
     * the number of levels we have dedented. This gets turned into multiple
     * dedent tokens later. *)
    let eval_indent str =
        let depth =
            Indent.depth_count 0 (Indent.explode str)
        in
            if depth < !cur_depth then begin
                let diff = !cur_depth - depth in
                cur_depth := depth;

                if diff > 1 then
                    DEDENT_MULT(diff)
                else
                    DEDENT
            end
            else if depth == !cur_depth then
                NEWLINE
```

```
            else begin
                cur_depth := depth;
                INDENT
            end
}

(* Complicated Regexes *)
let rgx_indent = ('\n'[' ''\t']*("//"[^'\n']*)*)+
let rgx_id = ['a'-'z']['A'-'Z''a'-'z''0'-'9''_']*

rule token = parse
(* White Space and Comments *)
| [' ''\t']                     { token lexbuf }
| rgx_indent as str             { eval_indent str }
| eof                           { EOF }

(* Operators *)
| '+'                           { ADD }
| '-'                           { MINUS }
| '*'                           { TIMES }
| '/'                           { DIVIDE }
| '<'                           { LT }
| '>'                           { GT }
| "<="                          { LTOE }
| ">="                          { GTOE }
| '='                           { EQUAL }
| "!="                          { NOTEQUAL}
| "|"                           { DISJ }    (* i.e. disjunct *)
| "&"                           { CONJ }    (* i.e. conjunct *)
| "!"                           { NOT }
| "t>"                          { PREPEND_TOP }
| "b>"                          { PREPEND_BOTTOM }
| "<t"                          { APPEND_TOP }
| "<b"                          { APPEND_BOTTOM }

(* Variables *)
| "new"                         { NEW }
| "configure"                   { CONFIGURE }
| "has"                         { HAS }
| "called"                      { CALLED }

(* Functions *)
| "do"                          { DO }
| "with"                        { WITH }
| "and"                         { AND }

(* Control Flow *)
```

```
| '('                           { OPENPAREN }
| ')'                           { CLOSEPAREN }
| '{'                           { OPENBRACE }
| '}'                           { CLOSEBRACE }
| "if"                          { IF }
| "else"                        { ELSE }
| "while"                       { WHILE }
| "for"                         { FOR }
| "break"                       { BREAK }
| "continue"                    { CONTINUE }

(* Literals *)
| ['0'-'9']+ as num             { NUMBER_LITERAL(int_of_string(num)) }
| '\"'[^'\"']* '\"' as str      { STRING_LITERAL(str) }
| "true"                        { BOOL_LITERAL(true) }
| "false"                       { BOOL_LITERAL(false) }

(* -------------- Miscellaneous ------------ *)
(* IDs can be any lowercase letter followed by a combination of numbers,
 * letters, or underscores. *)
| rgx_id as id                  { ID(id) }
| ((rgx_id)'.')+(rgx_id) as id      { DOT_ID(id) }

(* Type IDs can be an uppecase letter followed by a combination of letters. *)
| ['A'-'Z']['A'-'Z''a'-'z']* as _type              { TYPE(_type) }

(* This triggered if comment starts a program. Otherwise comments taken care of
 * in rgx_indent. *)
| "//"[^'\n']*                  { token lexbuf }

(* Punctuation *)
| ":"                           { COLON }
| ";"                           { SEMI }
```

Parser.mly

```
%{
(* parser.mly takes a sequence of tokens produced by the scanner and assembles
 * them into an abstract syntax tree. Each pattern-action rule takes some
 * pattern in the thus-far-assembled input and createse a "higher" type out
 * of it. *)

open Ast
```

```
(* Splits an id with dots in it into a list of ids *)
let split_dot_id dot_id =
    Str.split (Str.regexp "[.]") dot_id

%}

%token EOF
%token NEWLINE INDENT DEDENT
%token <int> DEDENT_MULT
%token DO WITH AND
%token NEW COLON CONFIGURE
%token IF ELSE WHILE FOR SEMI BREAK CONTINUE
%token OPENPAREN CLOSEPAREN OPENBRACE CLOSEBRACE
%token <bool> BOOL_LITERAL
%token <string> STRING_LITERAL ID DOT_ID TYPE
%token <int> NUMBER_LITERAL
%token ADD MINUS TIMES DIVIDE LT LTOE GT GTOE EQUAL NOTEQUAL
%token NOT DISJ CONJ
%token PREPEND_TOP PREPEND_BOTTOM APPEND_TOP APPEND_BOTTOM
%token HAS CALLED

/* Lowest Precedence */
%left DISJ CONJ
%left EQUAL NOTEQUAL
%nonassoc LT LTOE GT GTOE
%left ADD MINUS
%left TIMES DIVIDE
%right NOT
/* Highest Precedence */

%start program
%type <Ast.program> program

%%

program:
    | header vdecl_list func_list EOF       { { configs = List.rev (fst $1);
                                                field_decls = List.rev (snd $1);
                                                vars = List.rev $2;
                                                funcs = List.rev $3; } }
    | header func_list EOF                  { { configs = List.rev (fst $1);
                                                field_decls = List.rev (snd $1);
                                                vars = [];
                                                funcs = List.rev $2; } }

header:
    | config_list                           { ($1, []) }
```

```
    | config_list field_decl_list          { ($1, $2) }

config_list:
    | /* nothing */                        { [] }
    | config_list config                   { $2 :: $1 }
    | config_list NEWLINE                  { $1 }

config:
    | CONFIGURE ID COLON expr              { { config_id = $2; config_value = $4 } }

field_decl_list:
    | field_decl                           { [$1] }
    | field_decl_list field_decl           { $2 :: $1 }
    | field_decl_list NEWLINE              { $1 }

field_decl:
    | TYPE HAS TYPE CALLED ID              { { parent_type = $1;
                                               field_type = $3;
                                               field_id = $5; } }

vdecl_list:
    | vdecl                                { [$1] }
    | vdecl_list vdecl                     { $2 :: $1 }
    | vdecl_list NEWLINE                   { $1 }

vdecl:
    | NEW TYPE ID COLON expr               { VarDecl({ var_decl_id = $3;
                                                       var_decl_type = $2;
                                                       var_decl_value = $5 }) }

func_list:
    | func                                 { [$1] }
    | func_list func                       { $2 :: $1 }
    | func_list NEWLINE                    { $1 }

func:
    | ID COLON block                       { { decl_name = $1;
                                               formals = [];
                                               body = $3; } }
    | ID WITH formal_list COLON block      { { decl_name = $1;
                                               formals = List.rev $3;
                                               body = $5; } }

formal_list:
    | formal                               { [$1] }
    | formal_list AND formal               { $3 :: $1 }
```

```
formal:
    | TYPE ID                                { { formal_id = $2; formal_type = $1 } }

block:
    | INDENT stmt_list DEDENT               { List.rev $2 }
    | INDENT stmt_list EOF                  { List.rev $2 }

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

stmt:
    | stmt NEWLINE                          { $1 }
    | update                                { Update($1) }
    | DO ID                                 { Call({ fname = $2; args = [] }) }
    | DO ID WITH arg_list                   { Call({ fname = $2;
                                                     args = List.rev $4 }) }
    | IF expr COLON block                   { If($2, $4, []) }
    | IF expr COLON block ELSE COLON block  { If($2, $4, $7) }
    | WHILE expr COLON block                { While($2, $4) }
    | FOR update SEMI expr SEMI update
      COLON block                           { For($2, $4, $6, $8) }
    | BREAK                                 { Break }
    | CONTINUE                              { Continue }
    | OPENBRACE stmt CLOSEBRACE TIMES expr  { TimesLoop($2, $5) }
    | expr PREPEND_TOP expr                 { Prepend($1, $3, Top) }
    | expr PREPEND_BOTTOM expr              { Prepend($1, $3, Bottom) }
    | expr APPEND_TOP expr                  { Append($3, $1, Top) }
    | expr APPEND_BOTTOM expr               { Append($3, $1, Bottom) }

arg_list:
    | expr                                  { [$1] }
    | arg_list AND expr                     { $3 :: $1 }

update:
    | vdecl                                 { $1 }
    | var COLON expr                        { Assign($1, $3) }

expr:
    | NUMBER_LITERAL                        { Number($1) }
    | BOOL_LITERAL                          { Boolean($1) }
    | STRING_LITERAL                        { String($1) }
    | var                                   { Var($1) }
    | expr ADD expr                         { Binop($1, Add, $3) }
    | expr MINUS expr                       { Binop($1, Minus, $3) }
    | expr TIMES expr                       { Binop($1, Multiply, $3) }
    | expr DIVIDE expr                      { Binop($1, Divide, $3) }
```

```
    | expr LT expr                          { Binop($1, Lt, $3) }
    | expr LTOE expr                        { Binop($1, Ltoe, $3) }
    | expr GT expr                          { Binop($1, Gt, $3) }
    | expr GTOE expr                        { Binop($1, Gtoe, $3) }
    | expr EQUAL expr                       { Binop($1, Equal, $3) }
    | expr NOTEQUAL expr                    { Binop($1, NotEqual, $3) }
    | expr DISJ expr                        { Binop($1, Disj, $3) }
    | expr CONJ expr                        { Binop($1, Conj, $3) }
    | NOT expr                              { Unop(Not, $2) }
    | OPENPAREN expr CLOSEPAREN             { $2 }

var:
    | ID                                    { SimpleId($1) }
    | DOT_ID                                { DotId(split_dot_id $1) }

%%
```

ast.ml

```
(* ast.ml defines a set of disjoint unions or algebraic types that appear
 * in our parse tree. The parser is responsible for assembling a series
 * of tokens into our tree, and ultimately specifies the complete grammar
 * for our language. But the AST can be thought of as specifying the
higher-
 * level structure of our grammar, once all tokens have been parsed into
 * a type. ast.ml defines all of those types, i.e. every type that will
appear
 * in our tree.
 *
 * NOTE: In this file a type cannot be used before it has been declared.
 * NOTE: If you change anything about a type, please update the matching
pretty
 * print function below. *)

(*
============================================================================
*)
(*                            Abstract Syntax Tree
*)
(*
============================================================================
*)
(* Standard operations of any arity. *)
type op = Add | Minus | Multiply | Divide | Equal | NotEqual | Lt | Gt |
Ltoe
            | Gtoe | Disj | Conj | Not

(* A variable, like "a," "turn_count," or "player.hand.top"
```

```
 * Variables can appear in expressions or as part of assignments. *)
type var =
    | SimpleId of string
    | DotId of string list

type expr =
    | Number of int            (* Literal *)
    | String of string         (* Literal *)
    | Boolean of bool          (* Literal *)
    | Var of var
    | Unop of op * expr
    | Binop of expr * op * expr

(* Record for a configuration declaration, i.e. assignment to environment
 * variable. *)
type config_decl = {
    config_id : string;
    config_value : expr;
}

(* Record for a field declaration.
 * e.g. Table has Set called discard *)
type field_decl = {
    parent_type : string;
    field_type : string;
    field_id : string;
}

(* The header for a program consists of configure and field declarations.
*)
type header = config_decl list * field_decl list

(* Record for variable declaration.
 * Here we're using "_type" because "type" is reserved in OCaml *)
type var_decl = {
    var_decl_id : string;
    var_decl_type : string;
    var_decl_value : expr;
}

(* Record for a function call *)
type func_call = {
    fname : string;
    args : expr list;
}

(* An "update" is a kind of statement that you can put in the initial
 * assignment and update sections of a for-loop:
 *
 *      for (update; condition; update) ...
 *
 * An update can only be a variable declaration or an assignment. You
can't
```

```
 * have other kinds of statements--like if statements or while loops—in
a
 * for-loop header. *)
type update =
    | Assign of var * expr
    | VarDecl of var_decl

(* Whether a card is drawn form the top or bottom of a deck. *)
type draw_source = Top | Bottom

(* None of our statements are also expressions. They do not evaluate to
 * anything; they only have side-effects. *)
type stmt =
    | Update of update (* Ensures Assigns and VarDecls are statements *)
    | Call of func_call
    | If of expr * stmt list * stmt list
    | While of expr * stmt list
    | For of update * expr * update * stmt list
    | Break
    | Continue
    | TimesLoop of stmt * expr
    | Prepend of expr * expr * draw_source
    | Append of expr * expr * draw_source

(* A formal argument has a type and an ID, but no assigned value. *)
type formal = {
    formal_id : string;
    formal_type : string;
}

(* Record for a function declaration. *)
type func_decl = {
    decl_name : string;
    formals : formal list;
    body : stmt list;
}

(* A program consists of a series of variable declarations followed by a
series
 * of function declarations. *)
type program = {
    configs : config_decl list;
    field_decls: field_decl list;
    vars : update list;
    funcs: func_decl list;
}

(*
============================================================================
*)
(*                              Pretty Printing
*)
```

```
(*
================================================================================
*)
(* The printed tree has a (<type>, val) tuple for each node in the AST. *)
let string_of_op = function
    | Add -> "+"
    | Minus -> "-"
    | Multiply -> "*"
    | Divide -> "/"
    | Equal -> "="
    | NotEqual -> "!="
    | Lt -> "<"
    | Gt -> ">"
    | Ltoe -> "<="
    | Gtoe -> ">="
    | Disj -> "|"
    | Conj -> "&"
    | Not -> "!"

let string_of_var = function
    | SimpleId id -> "(<Var> " ^ id ^ ")"
    | DotId id_list -> "(<Var> id:" ^ (String.concat "." id_list) ^ ")"

let rec string_of_expr expr =
    let value =
        match expr with
        | Number num -> "(<Number> " ^ string_of_int num ^ ")"
        | String str -> "(<String> " ^ str ^ ")"
        | Boolean boolean ->
            let b =
                if boolean then
                    "true"
                else
                    "false"
            in
                "(<Boolean> " ^ b ^ ")"
        | Var v -> string_of_var v
        | Unop(op, e) -> "(<Unop> " ^ string_of_op op ^ string_of_expr e
^ ")"
        | Binop(e1, op, e2) -> "(<Binop> " ^ string_of_expr e1 ^ " " ^
                               string_of_op op ^ " " ^ string_of_expr e2
^ ")"
    in
        "(<Expr> " ^ value ^ ")"

(* e.g. (<Call> name:foo args:[expr, expr]) *)
 let string_of_call call =
    let args_s =
        String.concat ", " (List.map (fun arg -> string_of_expr arg)
call.args)
    in
        "(<Call> id:" ^ call.fname ^ " args:[" ^ args_s ^ "])"

let string_of_var_decl var_d =
```

```
    "(<VarDecl> id:" ^ var_d.var_decl_id ^ " type:"
    ^ var_d.var_decl_type ^ " value:" ^
    string_of_expr var_d.var_decl_value ^ ")"

let string_of_update update =
    let value =
        match update with
        | Assign(var, e) -> "(<Assign> var:" ^ string_of_var var ^ "
expr:"
                            ^ string_of_expr e ^ ")"
        | VarDecl(var_d) -> string_of_var_decl var_d
    in
        "(<Update> " ^ value ^ ")"

let rec string_of_stmt stmt =
    let value =
        match stmt with
        | Call call -> string_of_call call
        | Update(update) -> string_of_update update
        | If(e, tb, fb) ->  (* expr, true-block, false-block *)
               "(<If> p:" ^ string_of_expr e ^ " t-block:[\n  " ^
               string_of_block tb ^ "\n] f-block:[\n  " ^
               string_of_block fb ^ "\n])"
        | While(e, b) ->  (* expr, block *)
               "(<While> p:" ^ string_of_expr e ^ " loop:[\n  " ^
               string_of_block b ^ "\n])"
        | Break -> "(<Break>)"
        | Continue -> "(<Continue>)"
        | For(a, e, u, b) ->  (* assign, expr, update, block *)
               "(<For> assign:" ^ string_of_update a ^  " p:" ^
               string_of_expr e ^ " update:" ^
               string_of_update u ^ " loop:[\n  " ^
               string_of_block b ^ "\n])"
        | TimesLoop(stmt, expr) ->
               "(<TimesLoop> statement:" ^ string_of_stmt stmt ^ "
times:" ^
               string_of_expr expr ^ ")"
        | Prepend(e1, e2, draw_source) ->
               let op =
                   match draw_source with
                   | Top -> "t>"
                   | Bottom -> "b>"
               in
                   "(<Prepend> " ^ string_of_expr e1 ^ " " ^ op ^ " " ^
                   string_of_expr e2 ^ ")"
        | Append(e1, e2, draw_source) ->
               let op =
                   match draw_source with
                   | Top -> "<t"
                   | Bottom -> "<b"
               in
                   "(<Append> " ^ string_of_expr e1 ^ " " ^ op ^ " " ^
                   string_of_expr e2 ^ ")"
    in
```

```
        "(<Stmt> " ^ value ^ ")"
and string_of_block block =
    String.concat ",\n  " (List.map string_of_stmt block)

let string_of_function func =
    let formals_s =
        String.concat ", " (List.map (fun formal -> formal.formal_type ^
                                " " ^ formal.formal_id) func.formals)
    in
        "(<Func> fname:" ^ func.decl_name ^ " formals:[" ^ formals_s
        ^ "] body:\n  " ^ string_of_block func.body ^ "\n)"

let string_of_config (config : config_decl) =
    "(<Configure> id:" ^ config.config_id ^ " value:" ^
    string_of_expr config.config_value ^ ")"

let string_of_field_decl field_decl =
    "(<FieldDecl> parent_type:" ^ field_decl.parent_type ^ " field_type:"
^
    field_decl.field_type ^ " id:" ^ field_decl.field_id ^ ")"

(* List.fold_left here instead of String.concat so we can get a \n at the
end
 * of the list as well as between the items in the list. *)
let string_of_program program =
    let append_nl s1 s2 =
        s1 ^ s2 ^ "\n"
    in let configs_s =
        List.fold_left append_nl "" (List.map string_of_config
program.configs)
    in let field_decls_s =
        List.fold_left append_nl ""
            (List.map string_of_field_decl program.field_decls)
    in let vars_s =
        List.fold_left append_nl "" (List.map string_of_update
program.vars)
    in let funcs_s =
        List.fold_left append_nl "" (List.map string_of_function
program.funcs)
    in
        "(<Prgm>\n" ^ configs_s ^ field_decls_s ^ vars_s ^ funcs_s ^
")\n"
```

indent.ml

```
(* indent.ml contains useful methods used in scanner.mll that help implement
 * python-style indentation-based blocks. Indents can be made using actual
 * tab characters or spaces. A mix can even be used, but it would be
 * very confusing. *)
```

```
(* Converts a string to a list of char *)
let explode str =
    let rec exp len ls =
        if len < 0 then
            ls
        else
            exp (len - 1) (str.[len] :: ls)
    in
        exp (String.length str - 1) []

(* Returns the indentation level of the last line in the given string.
 * i.e., the number of tabs and spaces after the last '\n' *)
let rec depth_count count_so_far = function
    | [] -> count_so_far
    | c :: ls ->
        match c with
        | '\t'
        | ' ' -> depth_count (count_so_far + 1) ls
        | '\n' -> depth_count 0 ls
        | _ -> depth_count count_so_far ls
```

cache.ml

```
open Parser

let rec build_list num_left ls =
    if num_left == 0 then
        ls
    else
        build_list (num_left - 1) (DEDENT :: ls)

(* Reads tokens from the scanner and passes them on to the parser. If a
 * DEDENT_MULT token is read, caches a number of DEDENT tokens equal to the
 * depth change, which are then each passed to the parser before going back
 * to getting tokens from the scanner. *)
let process =
    let cache = ref [] in (* This will be like a static var in the function. *)
    fun lexbuf ->
        match !cache with
        | tok::ls -> cache := ls; tok
        | [] -> match Scanner.token lexbuf with
```

```
            | DEDENT_MULT(diff) -> cache := build_list (diff - 1) []; DEDENT
            | tok -> tok
```

stdlib.ml

```
(* stdlib.ml contains the API for our runtime system. You can think of it as
 * do's stdlib.h or stdio.h. It allows programmers to interact with the runtime
 * system we've set up with our java classes.
 *
 * This API is in the form of a whole bunch of different declaration types.
 * In C, this is accomplished by including a C header file full of declarations
 * in C. We aren't using include statements, so we just need a list of all
 * the declarations to pre-load into our environment as if they had actually
 * been declared. That way different variables and functions can be set and
 * called in a do program without confusing the semantic analyzer.
 *
 * See check_prgm in semantic.ml. That's where these lists are loaded into the
 * type-checking environment. *)

open Sast

(* A list of var_decls that correspond to variables in our environment. The
 * list has the form [(var_decl, java) ...], i.e. it's a list of tuples with
 * the equivalent java code that a reference to a var will be converted to
 * at compile time. All of the var_decl_value fields are 0 because we don't
 * need them. *)
let vars = [
    ( { var_decl_id = "player1";
        var_decl_type = PlayerType;
        var_decl_value = (Sast.Number(0), NumberType); },
      "players.get(0)");
    ( { var_decl_id = "player2";
        var_decl_type = PlayerType;
        var_decl_value = (Sast.Number(0), NumberType); },
      "players.get(1)");
    ( { var_decl_id = "player3";
        var_decl_type = PlayerType;
        var_decl_value = (Sast.Number(0), NumberType); },
      "players.get(2)");
    ( { var_decl_id = "player4";
        var_decl_type = PlayerType;
        var_decl_value = (Sast.Number(0), NumberType); },
      "players.get(3)");
    ( { var_decl_id = "jack";
        var_decl_type = NumberType;
```

```
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.JACK");
  ( { var_decl_id = "queen";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.QUEEN");
  ( { var_decl_id = "king";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.KING");
  ( { var_decl_id = "ace";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.ACE");
  ( { var_decl_id = "diamond";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.DIAMOND");
  ( { var_decl_id = "club";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.CLUB");
  ( { var_decl_id = "heart";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.HEART");
  ( { var_decl_id = "spade";
      var_decl_type = NumberType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "Card.SPADE");
  ( { var_decl_id = "deck";
      var_decl_type = SetType;
      var_decl_value = (Sast.Number(0), NumberType); },
    "deck");
]

(* A list of config_decls that correspond to configurable environment variables
 * in our runtime environment. Here config_value also doesn't matter. *)
let configs = [
  { config_id = "numberOfPlayers";
    config_value = (Sast.Number(0), NumberType);
    config_type = NumberType; };
  { config_id = "highestCard";
    config_value = (Sast.Number(0), NumberType);
    config_type = NumberType; };
  { config_id = "ascendingOrder";
    config_value = (Sast.Boolean(false), BooleanType);
```

```
      config_type = BooleanType; }
]

(* A list of fields for each of our aggregate types. IDs containing dots get
 * checked against this list to make sure they are valid. See check_fields
 * in semantic.ml *)
let fields = [
   ({ parent_type = CardType; field_type = NumberType; field_id = "rank"; },
        "val");
   ({ parent_type = CardType; field_type = NumberType; field_id = "suit"; },
        "suit");
   ({ parent_type = CardType; field_type = StringType; field_id = "desc"; },
        "toString()");
   ({ parent_type = SetType; field_type = NumberType; field_id = "size" },
        "size()");
   ({ parent_type = SetType; field_type = CardType; field_id = "top" },
        "top()");
   ({ parent_type = SetType; field_type = CardType; field_id = "bottom" },
        "bottom()");
   ({ parent_type = SetType; field_type = StringType; field_id = "desc"; },
        "toString()");
   ({ parent_type = PlayerType; field_type = SetType; field_id = "hand" },
        "hand");
   ({ parent_type = PlayerType; field_type = StringType; field_id = "desc"; },
        "toString()");
]

(* a list of tuples: ((funcs:Sast.func_decl),(java:java_call)) *)
let funcs = [
   (* ========================================================= *)
   (*                    Main Functions                         *)
   (* ========================================================= *)
   (* Print a line *)
   { decl_name = "output";
      formals = [];
      body = []; };

   (* Print a string *)
   { decl_name = "output";
     formals = [ { formal_id = "str";
                   formal_type = Sast.StringType; } ];
     body = []; };

   (* Print a number *)
   { decl_name = "output";
     formals = [ { formal_id = "number";
                   formal_type = Sast.NumberType; } ];
```

```
  body = []; };

(* Print a boolean *)
{ decl_name = "output";
  formals = [ { formal_id = "boolean";
                formal_type = Sast.BooleanType; } ];
  body = []; };

(* Print a card *)
{ decl_name = "output";
  formals = [ { formal_id = "card";
                formal_type = Sast.CardType; } ];
  body = []; };

(* Print a set *)
{ decl_name = "output";
  formals = [ { formal_id = "set";
                formal_type = Sast.SetType; } ];
  body = []; };

(* Print a player *)
{ decl_name = "output";
  formals = [ { formal_id = "player";
                formal_type = Sast.PlayerType; } ];
  body = []; };

(* Input Bool *)
{ decl_name = "input";
  formals = [ { formal_id = "bool";
                formal_type = Sast.BooleanType; } ];
  body = []; };

(* Input Int *)
{ decl_name = "input";
  formals = [ { formal_id = "num";
                formal_type = Sast.NumberType; } ];
  body = []; };

(* Input String *)
{ decl_name = "input";
  formals = [ { formal_id = "str";
                formal_type = Sast.StringType; } ];
  body = []; };

(* Quit *)
{ decl_name = "quit";
  formals = [];
```

```
     body = []; };

(*

   (* ====================== *)
   (* MyPlayer type functions *)
   (* ====================== *)
   ({ decl_name = "";
      formals =
          [{ formal_id = "card";
             formal_type = CardType; };  ];
      body = []; },
      "drawCard(card)" ) ;
   ({ decl_name = "";
      formals =
          [{ formal_id = "set";
             formal_type = SetType; }];
      body = []; },
      "drawCard(set)") ;
   ({ decl_name = "";
      formals =
          [{ formal_id = "i";
             formal_type = NumberType; }];
      body = []; },
      "playCard(i)") ;
   ({ decl_name = "";
      formals = [];
      body = []; },
      "getScore()") ;
   ({ decl_name = "";
      formals = [];
      body = []; },
      "selectCard()")

   (* ================= *)
   (* Set type functions *)
   (* ================= *)
   ({ decl_name = "";
      formals = [];
      body = []; },
      "shuffle()") ;
   ({ decl_name = "";
      formals = List.concat ({ formal_id = "card"; formal_type = CardType; } list)
[ { formal_id = "i"; formal_type = NumberType; };
      { formal_id = "j"; formal_type = NumberType; } ] ;
      body = []; },
      "swap(deck, i ,j)") ;
```

```
   ({ decl_name = "";
      formals = [];
      body = []; },
      "draw()") ;
   ({ decl_name = "";
      formals =
          [{ formal_id = "n";
             formal_type = NumberType; }];
      body = []; },
      "draw_hand(n)") ;
      *)
]
```

print.ml

```
(* printer.ml prints the AST for a program using the pretty print functions
 * defined in ast.ml
 *
 * USAGE: ./printer test_file.do *)

open Ast

let print_tree lexbuf =
    let program =
        Parser.program Cache.process lexbuf
    in
        print_string (string_of_program program)

let _ =
    let file =
        open_in Sys.argv.(1)
    in
        print_tree (Lexing.from_channel file)
```

sast.ml

```
(* sast.ml contains our semantically analyzed abstract syntax tree. Basically,
 * it is our ast but with type information attached.
 *
 * Many of the AST types have to redeclared here even if they haven't
 * changed. If they incorporate a type that has itself changed, the whole
 * type has to reclared so that the new version is incorporated rather than
 * the old version. *)
open Ast
```

```
type datatype = BooleanType | NumberType | StringType | CardType | SetType
                | PlayerType

type simple_expr =
    | Number of int
    | String of string
    | Boolean of bool
    | Var of string
    | Unop of op * expr
    | Binop of expr * op * expr
and expr = simple_expr * datatype

type config_decl = {
    config_id : string;
    config_value : expr;
    config_type : datatype;
}

type field_decl = {
    parent_type : datatype;
    field_type : datatype;
    field_id : string;
}

type var_decl = {
    var_decl_id : string;
    var_decl_type : datatype;
    var_decl_value : expr;
}

type update =
    | Assign of string * expr
    | VarDecl of var_decl

type func_call = {
    fname : string;
    args : expr list;
}

type stmt =
    | Update of update
    | Call of func_call
    | If of expr * stmt list * stmt list
    | While of expr * stmt list
    | For of update * expr * update * stmt list
    | Break
```

```
    | Continue
    | TimesLoop of stmt * expr
    | Prepend of expr * expr * draw_source
    | Append of expr * expr * draw_source

type formal = {
    formal_id : string;
    formal_type : datatype;
}

type func_decl = {
    decl_name : string;
    formals : formal list;
    body : stmt list;
}

type program = {
    configs : config_decl list;
    field_decls : field_decl list;
    vars : update list;
    funcs : func_decl list;
}

(* For use with exceptions *)
let string_of_type = function
    | BooleanType -> "Boolean"
    | NumberType -> "Number"
    | StringType -> "String"
    | CardType -> "Card"
    | SetType -> "Set"
    | PlayerType -> "Player"
```

semantic.ml

```
(* semantic.ml creates an sast from our ast. It basically resolves each item
 * in the tree to a type and raises errors if there is a type mismatch
 * or unknown ID reference. *)
open Ast
open Sast

(* ---------------------------- Context ------------------------------------ *)
(* A symbol_table maps ids -> var_decls and allows us to verify that a variable
 * has been declared and that it is the right type. *)
type symbol_table = {
```

```
    parent : symbol_table option;
    mutable vars: Sast.var_decl list;
}

(* An environment represents the current context for a particular node in our
 * tree. *)
type environment = {
    configs: Sast.config_decl list;
    mutable fields: Sast.field_decl list;
    scope : symbol_table;
    mutable unchecked_calls: Sast.func_call list;
    mutable func_decls: Sast.func_decl list;
    can_break : bool; (* If a break statement makes sense. *)
    can_continue: bool; (* If a continue statement makes sense. *)
}

(* -------------------------- Exceptions ----------------------------------- *)
exception UnknownType of string
exception UndeclaredID of string
exception TypeMismatch of string
exception WrongType of string
exception Redeclaration of string
exception BadProgram of string (* No setup or round procedures. *)

(* For syntax-ish errors that we're only catching now. *)
exception IllegalUsage of string

(* ----------------------- Helper Functions ------------------------------- *)
(* Converts a string representing a type to a type, or throws an error if the
 * type is unrecognized. *)
let type_of_string type_str =
    match type_str with
    | "Boolean" -> BooleanType
    | "Number" -> NumberType
    | "String" -> StringType
    | "Card" -> CardType
    | "Set" -> SetType
    | "Player" -> PlayerType
    | _ -> raise (UnknownType("The type \"" ^ type_str ^ "\" is not valid."))

(* Looks for a matching config_decl and returns it. *)
let find_config env id =
    List.find (fun config_decl -> id = config_decl.config_id) env.configs

(* Looks for a var in the current scope. If it isn't there, checks the next
 * scope. If we've reach global scope and we still haven't found the var, throw
 * an error. *)
```

```
let rec find_var scope id =
    try
        List.find (fun vdecl -> id = vdecl.var_decl_id) scope.vars
    with Not_found ->
        match scope.parent with
        | Some(parent) -> find_var parent id
        | _ -> raise Not_found

(* Checks to see if a var is in the current local scope. *)
let exists_var_local scope id =
    List.exists (fun vdecl -> id = vdecl.var_decl_id) scope.vars

(* Checks if a type has a field called id by looking through the fields
 * available in the current program. *)
let find_field env (_type, id) =
    List.find (fun field_decl ->
                ((_type, id) = (field_decl.parent_type, field_decl.field_id)))
                    env.fields

(* Checks to see if a field called id exists in the type _type. *)
let exists_field env (_type, id) =
    List.exists (fun field_decl ->
                 ((_type, id) = (field_decl.parent_type, field_decl.field_id)))
                    env.fields

(* Checks if each arg matches its formal. *)
let rec match_args args formals =
    match args, formals with
    | (arg :: args_rest), (formal :: formals_rest) ->
        let _, arg_type = arg
        in let form_type = formal.formal_type
        in if (arg_type = form_type) then
            match_args args_rest formals_rest
        else
            false
    | (_ :: _), []
    | [], (_ :: _) -> false
    | [], [] -> true

(* Tries to match a function call to a func_decl in the environment. Matches
 * with both id and arg types so our overloaded output function works. *)
let find_func_decl env fname args =
    List.find (fun func_decl -> (fname = func_decl.decl_name) &&
                                (match_args args func_decl.formals))
                                    env.func_decls

(* Checks if a function already exists using an ID only. So programmers using
```

```
 * our language cannot overload functions themselves. *)
let exists_func_decl env fname =
    List.exists (fun func_decl -> (fname = func_decl.decl_name)) env.func_decls

(* ============================================================================ *)
(*                            Semantic Analysis                                 *)
(* ============================================================================ *)
(* Recursively checks each ID in a DotId node after the first.
 * We need to check each ID to verify it's a valid field
 * in the type of the previous ID. *)
let rec check_fields env last_type id_ls =
    match id_ls with
    | id :: ls ->
        let id = List.hd id_ls
        in let field_decl =
            try
                find_field env (last_type, id)
            with Not_found ->
                raise (UndeclaredID("Undeclared field: \"" ^ id ^ ".\""))
        in
            check_fields env field_decl.field_type ls
    | [] -> last_type

(* Takes a node of type var in our AST and checks to make sure it refers
 * to something in the current scope. Then returns a node of type simple_expr
 * in our SAST, basically a normal var with attached type information retrieved
 * from the current scope or environment. *)
let check_var env = function
    | Ast.SimpleId(id) ->
        let vdecl =
            try
                find_var env.scope id
            with Not_found ->
                raise (UndeclaredID("Undeclared identifier: \"" ^ id ^ ".\""))
        in
            (id, vdecl.var_decl_type)
    | Ast.DotId(id_list) ->
        (* Check if the first id is a valid var, then check if subsequent ids
         * are valid fields. Final type is type of last field. *)
        let first_id = List.hd id_list
        in let vdecl =
            try
                find_var env.scope first_id
            with Not_found ->
                raise (UndeclaredID("Undeclared identifier: \"" ^ first_id ^
                        ".\""))
        in let final_id =
```

```
            String.concat "." id_list
        in
            (final_id, check_fields env vdecl.var_decl_type (List.tl id_list))

(* Takes a node of type Ast.expr and converts to Sast.expr, i.e. an expr
 * with an associated type. Also checks to make sure than all ops are used
 * with appropriate and matching types. *)
let rec check_expr env = function
    | Ast.Number(num) -> Sast.Number(num), NumberType
    | Ast.String(str) -> Sast.String(str), StringType
    | Ast.Boolean(b) -> Sast.Boolean(b), BooleanType
    | Ast.Var(var) ->
       let id, _type =
           check_var env var
       in
           Var(id), _type
    | Ast.Unop(op, expr) ->
       let checked_expr =
           check_expr env expr
       in let _, _type =
           checked_expr
       in begin match _type with
       | BooleanType -> Sast.Unop(op, checked_expr), _type
       | _ -> raise (WrongType(string_of_type _type ^ " cannot be used with" ^
                    " the \"" ^ string_of_op op ^ "\" operator.")) end
    | Ast.Binop(expr1, op, expr2) ->
       let checked_expr1 =
           check_expr env expr1
       in let _, type1 =
           checked_expr1
       in let checked_expr2 =
           check_expr env expr2
       in let _, type2 =
           checked_expr2
       in
           if (not (type1 = type2)) then
               raise (TypeMismatch(string_of_type type1 ^ " does not match "
                                   ^ string_of_type type2 ^ "."))
           else
               let raise_error _type op =
                   raise (WrongType(string_of_type _type ^ " cannot " ^
                          "be used with the \"" ^ string_of_op op ^
                          "\" operator."))
               in let _type =
                   match op with
                   | Add ->
                       begin match type1 with
```

```
                    | NumberType | StringType -> type1
                    | _ -> raise_error type1 op end
                | Minus | Multiply | Divide ->
                    begin match type1 with
                    | NumberType -> type1
                    | _ -> raise_error type1 op end
                | Equal | NotEqual ->
                    BooleanType
                | Lt | Gt | Ltoe | Gtoe ->
                    begin match type1 with
                    | NumberType | CardType -> BooleanType
                    | _ -> raise_error type1 op end
                | Disj | Conj ->
                    begin match type1 with
                    | BooleanType -> BooleanType
                    | _ -> raise_error type1 op end
                | _ -> raise (Failure("Illegal operator."))
            in
                Sast.Binop(checked_expr1, op, checked_expr2), _type

(* Takes a var_decl node and checks to see if the var has already been declared
 * in the current scope. Raise an error if it has. Then checks to make sure
 * the var decl has the type that it is supposed to have. If it does, we then
 * add it to the current scope and return an Sast.var_decl. *)
let check_var_decl env (vdecl : Ast.var_decl) =
    if exists_var_local env.scope vdecl.var_decl_id then
        raise (Redeclaration("The variable \"" ^ vdecl.var_decl_id ^ "\" has" ^
               " already been declared in its scope."))
    else
        let checked_expr =
            check_expr env vdecl.var_decl_value
        in let _, _type =
            checked_expr
        in
            if ((type_of_string vdecl.var_decl_type) =  _type) then
                let checked_vdecl =
                    { var_decl_id = vdecl.var_decl_id;
                      var_decl_type = _type;
                      var_decl_value = checked_expr; }
                in
                    (* Add to scope then return *)
                    env.scope.vars <- checked_vdecl :: env.scope.vars;
                    checked_vdecl
            else
                raise (TypeMismatch("You have assigned an expression of type" ^
                        "\"" ^ string_of_type _type ^ "\" to a variable of " ^
                        "type \"" ^ vdecl.var_decl_type ^ ".\""))
```

```
(* Header var decls get parsed as updates, but we need a function to check them
 * that always returns a var_decl so that they can be put into scope. See
 * check_pgrm below. *)
let check_update_header env update =
    match update with
    | Ast.Assign(_, _) -> raise (WrongType("You cannot assign in the header."))
    | Ast.VarDecl(vdecl) ->
        check_var_decl env vdecl

(* Checks an update by checking its subtypes. Also makes sure assignments
 * are valid. *)
let check_update env = function
    | Ast.Assign(var, expr) ->
        let var_id, var_type =
            check_var env var
        in let checked_expr =
            check_expr env expr
        in let _, expr_type =
            checked_expr
        in
            if (var_type = expr_type) then
                Sast.Assign(var_id, checked_expr)
            else
                raise (TypeMismatch("Cannot assign an expression of type \"" ^
                        string_of_type expr_type ^ "\" to a variable of " ^
                        "type \"" ^ string_of_type var_type ^ ".\""))
    | Ast.VarDecl(vdecl) ->
        Sast.VarDecl(check_var_decl env vdecl)

(* Takes a call and adds it to the environment to be checked later --
 * see check_call and check_prgm. Also checks the arguments to the call. *)
let add_call env (call : Ast.func_call) =
    let unchecked_call =
        { fname = call.fname;
          args = List.map (check_expr env) call.args }
    in
        env.unchecked_calls <- unchecked_call :: env.unchecked_calls;
        unchecked_call

(* Checks statements for semantic errors. Statements themselves don't have
 * types. Scoping largely implemented here. *)
let rec check_stmt env = function
    | Ast.Update(update) -> Sast.Update(check_update env update)
    | Ast.Call(call) -> Sast.Call(add_call env call)
    | Ast.If(expr, tblock, fblock) ->
        let checked_expr, expr_type = check_expr env expr in
```

```
    begin match expr_type with
    | NumberType | BooleanType ->
        let new_scope =
            { parent = Some(env.scope);
              vars = []; }
        in let new_env =
            { configs = env.configs;
              fields = env.fields;
              scope = new_scope;
              unchecked_calls = env.unchecked_calls;
              func_decls = env.func_decls;
              can_break = env.can_break;
              can_continue = env.can_continue; }
        in
            let checked_tblock =
                check_block new_env tblock
            in let checked_fblock =
                check_block new_env fblock
            in
                Sast.If((checked_expr, expr_type), checked_tblock,
                        checked_fblock)
    | _ -> raise (WrongType("The type \"" ^ string_of_type expr_type ^
                 "\" cannot appear in the predicate of an if statement."))
    end
| Ast.While(expr, block) ->
    let checked_expr, expr_type = check_expr env expr in
    begin match expr_type with
    | NumberType | BooleanType ->
        let new_scope =
            { parent = Some(env.scope);
              vars = []; }
        in let new_env =
            { configs = env.configs;
              fields = env.fields;
              scope = new_scope;
              unchecked_calls = env.unchecked_calls;
              func_decls = env.func_decls;
              can_break = true;
              can_continue = true; }
        in
            let checked_block =
                check_block new_env block
            in
                Sast.While((checked_expr, expr_type), checked_block)
    | _ -> raise (WrongType("The type \"" ^ string_of_type expr_type ^
                 "\" cannot appear in the predicate of an if statement."))
    end
```

```
| Ast.For(setup, expr, update, block) ->
    let checked_setup = check_update env setup in
    let checked_expr, expr_type = check_expr env expr in
    let checked_update = check_update env update in
    begin match expr_type with
    | NumberType | BooleanType ->
       begin match checked_update with
       | Sast.Assign(_,_) ->
           let new_scope =
               match checked_setup with
               | Sast.VarDecl(var_decl) ->
                   { parent = Some(env.scope);
                     vars = [ var_decl ]; }
               | _ ->
                   { parent = Some(env.scope);
                     vars = []; }
           in let new_env =
               { configs = env.configs;
                 fields = env.fields;
                 scope = new_scope;
                 unchecked_calls = env.unchecked_calls;
                 func_decls = env.func_decls;
                 can_break = true;
                 can_continue = true; }
           in
               let checked_block =
                   check_block new_env block
               in
                   Sast.For(checked_setup, (checked_expr, expr_type),
                            checked_update, checked_block)
       | _ -> raise (IllegalUsage("You cannot declare a variable in the"
                     ^ " update section of a for loop header.")) end
    | _ -> raise (WrongType("The type \"" ^ string_of_type expr_type ^
                 "\" cannot appear in the predicate of an if statement."))
    end
| Ast.Break ->
    if env.can_break then
        Sast.Break
    else
        raise (IllegalUsage("You can only use a break statement " ^
               "inside of a while or for loop."))
| Ast.Continue ->
    if env.can_continue then
        Sast.Continue
    else
        raise (IllegalUsage("You can only use a continue statement " ^
               "inside of a while or for loop."))
```

```
    | Ast.TimesLoop(stmt, expr) ->
        let checked_stmt =
            check_stmt env stmt
        in let checked_expr, expr_type =
            check_expr env expr
        in
            begin match expr_type with
            | NumberType ->
                Sast.(TimesLoop(checked_stmt, (checked_expr, expr_type)))
            | _ ->
                raise (WrongType("You can only use expressions of type " ^
                           "Number to repeat a statement."))
            end
    | Ast.Prepend(e1, e2, draw_source) ->
        let checked_expr1, expr_type1 =
            check_expr env e1
        in let checked_expr2, expr_type2 =
            check_expr env e2
        in begin match expr_type1, expr_type2 with
        | SetType, SetType ->
            Sast.Prepend((checked_expr1, expr_type1),
                         (checked_expr2, expr_type2), draw_source)
        | _ -> raise (WrongType("The prepend operator can only be used with " ^
                     "variables of type Set."))
        end
     | Ast.Append(e1, e2, draw_source) ->
        let checked_expr1, expr_type1 =
            check_expr env e1
        in let checked_expr2, expr_type2 =
            check_expr env e2
        in match expr_type1, expr_type2 with
        | SetType, SetType ->
            Sast.Append((checked_expr1, expr_type1),
                        (checked_expr2, expr_type2), draw_source)
        | _ -> raise (WrongType("The append operator can only be used with " ^
                       "variables of type Set."))
and check_block env block =
    List.map (check_stmt env) block

(* Checks that a config_decl refers to an existing configurable variable and
 * that the expression is of the right type. *)
let check_config env (config_decl : Ast.config_decl) =
    let real_config = (* i.e. the existing config *)
        try
            find_config env config_decl.config_id
        with Not_found ->
            raise (UndeclaredID("There is no configurable variable with the " ^
```

```
                "id \"" ^ config_decl.config_id ^ ".\""))
    in let checked_expr =
        check_expr env config_decl.config_value
    in let _, expr_type =
        checked_expr
    in
        if (expr_type = real_config.config_type) then
            { config_id = config_decl.config_id;
              config_value = checked_expr;
              config_type = expr_type; } (* Returning Sast.config_decl *)
        else
            raise (TypeMismatch("The configurable \"" ^ config_decl.config_id ^
                  "\" ^ " has type \"" ^
                  string_of_type real_config.config_type ^ "\" and cannot " ^
                  "be configured with an expression of type \"" ^
                  string_of_type expr_type ^ ".\""))

(* Checks that a field_decl is adding to type Player, since we decided that
 * it didn't make sense to extend Card or Set. Then makes sure that the
 * ID doesn't match a field that already exists. Finally, adds the field
 * to the environment. *)
let check_field_decl env (field_decl : Ast.field_decl) =
    match (type_of_string field_decl.parent_type) with
    | PlayerType ->
        begin match (type_of_string field_decl.field_type) with
        | NumberType | BooleanType | StringType | SetType ->
            if (not (exists_field env (PlayerType, field_decl.field_id))) then
                let checked_field =
                    { parent_type = type_of_string field_decl.parent_type;
                      field_type = type_of_string field_decl.field_type;
                      field_id = field_decl.field_id; }
                in
                    env.fields <- checked_field :: env.fields;
                    checked_field (* Returning Sast.field_decl *)
            else
                raise (Redeclaration("You cannot add the field \"" ^
                        field_decl.field_id ^ " to Player, because Player " ^
                        "already has a field by that name."))
        | _ -> raise (WrongType("You cannot add a field of type \"" ^
        field_decl.field_type ^ "\" to Player."))
        end
    | _ -> raise (WrongType("You cannot add a field to any type except Player."))

(* Converts an Ast.formal to an Sast.formal. This won't be necessary if we
 * decided to parse type strings into proper types in the first place. *)
let check_formal (formal : Ast.formal) =
    { formal_id = formal.formal_id;
```

```
      formal_type = (type_of_string formal.formal_type); }

(* Converts a formal parameter to a variable declaration. *)
let var_of_formal formal =
    { var_decl_id = formal.formal_id;
      var_decl_type = formal.formal_type;
      var_decl_value = (Sast.Number(0), NumberType); } (* This shouldn't ever be
accessed. *)

(* Checks the body of a function declaration before adding the function to the
 * environment. Also checks to make sure we aren't redeclaring a function.
 * This check is performed simpy with IDs, so overloading is not possible. *)
let check_func_decl env (func_decl : Ast.func_decl) =
    if (not (exists_func_decl env func_decl.decl_name)) then begin
        (* Adds formals to scope before checking body. *)
        let checked_formals =
            List.map check_formal func_decl.formals
        in let new_scope =
            { parent = Some(env.scope);
              vars = List.map var_of_formal checked_formals; }
        in let new_env =
            { configs = env.configs;
              fields = env.fields;
              scope = new_scope;
              unchecked_calls = env.unchecked_calls;
              func_decls = env.func_decls;
              can_break = false;
              can_continue = false; }
        in let checked_fdecl =
            { decl_name = func_decl.decl_name;
              formals = List.map check_formal func_decl.formals;
              body = List.map (check_stmt new_env) func_decl.body; }
        in
            env.func_decls <- checked_fdecl :: env.func_decls;
            checked_fdecl
        end
    else
        raise (Redeclaration("The procedure \"" ^ func_decl.decl_name
                ^ "\" already exists and cannot be redeclared."))

(* Checks to see if setup and round are declared in a program. *)
let rec has_setup_and_round has_setup has_round func_decls =
    match func_decls with
    | [] -> has_setup && has_round
    | fdecl :: rest ->
        match (fdecl.decl_name, List.length fdecl.formals) with
        | "setup", 0 -> has_setup_and_round true has_round rest
```

```
        | "round", 0 -> has_setup_and_round has_setup true rest
        | _ -> has_setup_and_round has_setup has_round rest

(* Checks to see if a call corresponds to a declared function. If not, throw
 * an error. Since a call only matches if it has been given the right args,
 * a call using the correct ID but wrong arg types will not work.
 * This function has type unit. *)
let check_call env (call : Sast.func_call) =
    let _ =
        try
            find_func_decl env call.fname call.args
        with Not_found ->
            raise (UndeclaredID("The procedure \"" ^ call.fname ^ "\" has " ^
                   "not been declared with the given parameters.."))
    in
        match call.fname with
        | "input" ->
            (* Input only ever takes one arg. *)
            let expr, _ =
                List.hd call.args
            in
                begin match expr with
                | Var(_) -> ()
                | _ -> raise (IllegalUsage("You can only use a variable " ^
                                "expression with \"input.\""))
                end
        | _ -> () (* Returns unit. *)

(* Performs semantic analysis on a program. Also makes sure that a program
 * has a setup and a round procedure. Finally, ensures that all calls are
 * matched with a func_decl.
 *
 * In other words, takes an AST and makes an SAST.
 *
 * The variables, configurations, fields, and functions provided by stdlib.ml
 * are added to the environment here. *)
let check_prgm (prgm : Ast.program) =
    let std_vars =
        List.map fst Stdlib.vars
    in let std_fields =
        List.map fst Stdlib.fields
    in let header_scope =
        { parent = None;
          vars = std_vars; }
    in let header_env =
        { configs = Stdlib.configs;
          fields = std_fields;
```

```
  scope = header_scope;
  unchecked_calls = [];
  func_decls = [];
  can_break = false;
  can_continue = false; }
in let added_fields =
    List.map (check_field_decl header_env) prgm.field_decls
in let all_fields =
    List.append std_fields added_fields
in let (added_vars : var_decl list) =
    List.map (check_update_header header_env) prgm.vars
in let (added_vars_update : update list) =
    List.map (fun var_decl -> Sast.VarDecl(var_decl)) added_vars
in let all_vars =
    List.append std_vars added_vars
in let global_scope =
    { parent = None;
      vars = all_vars; }
in let env =
    { configs = Stdlib.configs;
      fields = all_fields;
      scope = global_scope;
      unchecked_calls = [];
      func_decls = [];
      can_break = false;
      can_continue = false; }
in
    let checked_prgm =
        { configs = List.map (check_config header_env) prgm.configs;
          field_decls = added_fields;
          vars = added_vars_update;
          funcs = List.map (check_func_decl env) prgm.funcs; }
    in
        if (has_setup_and_round false false checked_prgm.funcs) then begin
            List.iter (check_call env) env.unchecked_calls;
            checked_prgm
            end
        else
            raise (BadProgram("You must have a setup and round procedure" ^
                    " in your program. They must both take 0 arguments."))
```

compile.ml

```
(* compile.ml produces an AST and prints it as a java Game class.
```

```
 *
 * Usage: ./compile program.do *)

open Sast

(* ============================================================================ *)
(*                              Helper Functions                                *)
(* ============================================================================ *)
exception CompilerError of string

(* Finds the java code for a given variable ID from stdlib. If the ID isn't
 * in stdlib, just returns the id. *)
let find_java_for_var var_id =
    let java_code =
        try
            let _, java =
                List.find (fun (vdecl, _) -> (vdecl.var_decl_id = var_id))
                          Stdlib.vars
            in
                java
        with Not_found ->
            var_id
    in
        java_code

(* Returns the java equivalent of a field access. *)
let find_java_for_field field_id =
    let java_code =
        try
            let _, java =
                List.find (fun (field_decl, _) ->
                    (field_decl.field_id = field_id)) Stdlib.fields
            in
                java
        with Not_found ->
            field_id
    in
        java_code

(* ============================================================================ *)
(*                              Java Printing                                   *)
(* ============================================================================ *)
(* This is basically a reimplementation of the pretty print functions in
*  ast.ml, except now we're printing in Java's syntax rather than our own
*  made-up "pretty" syntax.
*
*  The resulting program will be ugly because we don't have information about
```

```
*  spacing or indentation in the SAST. We could maybe keep track of depth and
*  space things that way, but it's not really worth it because nobody is meant
*  to see the java code and java ignores whitespace anyway. *)

(* Converts a type to a string representation of a java type. *)
let java_of_type = function
    | BooleanType -> "boolean"
    | NumberType -> "int"
    | StringType -> "String"
    | CardType -> "Card"
    | SetType -> "Set"
    | PlayerType -> "MyPlayer"

(* Converts an op to the Java equivalent. *)
let java_of_op = function
    | Ast.Add -> "+"
    | Ast.Minus -> "-"
    | Ast.Multiply -> "*"
    | Ast.Divide -> "/"
    | Ast.Equal -> "=="
    | Ast.NotEqual -> "!="
    | Ast.Lt -> "<"
    | Ast.Gt -> ">"
    | Ast.Ltoe -> "<="
    | Ast.Gtoe -> ">="
    | Ast.Disj -> "||"
    | Ast.Conj -> "&&"
    | Ast.Not -> "!"

(* Returns the proper java field names concatenated together. *)
let rec java_of_field_vars field_vars =
    match field_vars with
    | [] -> ""
    | _ -> "." ^ (String.concat "." (List.map find_java_for_field field_vars))

(* Returns the java code for a var. Vars in the stdlib have special java
 * representations, most vars just use the var_id. *)
let java_of_var var_id =
    let id_list =
        Str.split (Str.regexp("[.]")) var_id
    in
        match id_list with
        | [] -> raise (CompilerError("No id given to java_of_var."))
        | hd :: tl -> find_java_for_var hd ^ java_of_field_vars tl

(* Converts a simple expression in our SAST to java code, checking the stdlib
 * for special java code representations. *)
```

```
let rec java_of_expr = function
    | Number(num), _ -> string_of_int num
    | String(str), _ -> str
    | Boolean(boolean), _ ->
       if boolean then
           "true"
       else
           "false"
    | Var(var), _ -> java_of_var var
    | Unop(op, e), _ -> java_of_op op ^ java_of_expr e
    | Binop(e1, op, e2), _  ->
       let _, expr_types =
           e1
       in
           begin match expr_types with
           | NumberType | BooleanType ->
              "(" ^ java_of_expr e1 ^ " " ^ java_of_op op ^ " " ^
              java_of_expr e2 ^ ")"
           | StringType ->
              begin match op with
              | Ast.Add ->
                 "(" ^ java_of_expr e1 ^ " " ^ java_of_op op ^ " " ^
                 java_of_expr e2 ^ ")"
              | Ast.Equal ->
                 "(Utility.compareString(" ^ java_of_expr e1 ^ ", " ^
                 java_of_expr e2 ^ "))"
              | Ast.NotEqual ->
                 "(!Utility.compareString(" ^ java_of_expr e1 ^ ", " ^
                 java_of_expr e2 ^ "))"
              | _ -> raise (CompilerError("Invalid Op."))
              end
           | CardType ->
              begin match op with
              | Ast.Equal -> "(Utility.cardEqual(" ^ java_of_expr e1  ^
                     ", " ^ java_of_expr e2 ^  "))"
              | Ast.NotEqual -> "(Utility.cardNotEqual(" ^ java_of_expr e1  ^
                     ", " ^ java_of_expr e2 ^ "))"
              | Ast.Lt -> "(Utility.cardLessThan(" ^ java_of_expr e1  ^
                     ", " ^ java_of_expr e2 ^ "))"
              | Ast.Gt -> "(Utility.cardGreaterThan(" ^ java_of_expr e1  ^
                     ", " ^ java_of_expr e2 ^ "))"
              | Ast.Ltoe -> "(Utility.cardLessOrEqualThan(" ^ java_of_expr e1
                        ^ ", " ^ java_of_expr e2 ^ "))"
              | Ast.Gtoe -> "(Utility.cardGreaterOrEqualThan(" ^
                       java_of_expr e1  ^ ", " ^
                       java_of_expr e2 ^ "))"
              | _ -> raise (CompilerError("Invalid Op."))
```

```
            end
        | SetType | PlayerType ->
            begin match op with
            | Ast.Equal | Ast.NotEqual ->
                "(" ^ java_of_expr e1 ^ " " ^ java_of_op op ^ " " ^
                java_of_expr e2 ^ ")"
            | _ -> raise (CompilerError("Invalid Op."))
            end
        end

(* Converts a config_decl to a java assignment. Only numbers, booleans, or
 * variables can be used for configure statements. *)
let java_of_config config =
    match config.config_value with
    | Number(n), _ ->
        let num =
            if (config.config_id = "highestCard") then
                n - 1
            else
                n
        in
            config.config_id ^ " = " ^ string_of_int (num) ^ ";"
    | Boolean(b), _ -> config.config_id ^ " = " ^ string_of_bool b ^ ";"
    | Var(var), _-> config.config_id ^ " = " ^ java_of_var var ^ ";"
    | _ -> raise (CompilerError("Invalid type used for configure statement."))

(* Converts a field decl to a java instance var declaration. *)
let java_of_field_decl field_decl =
    (string_of_type field_decl.field_type) ^ " " ^ field_decl.field_id ^ ";\n"

(* Converts a field decl to a java var assignment, which will appear in the
 * constructor of the MyPlayer class. *)
let java_of_field_decl_assign field_decl =
    match field_decl.field_type with
    | SetType -> field_decl.field_id ^ " = new Set();"
    | StringType -> field_decl.field_id ^ " = \"\";"
    | BooleanType -> field_decl.field_id ^ " = false;"
    | NumberType -> field_decl.field_id ^ " = 0;"
    (* Should be caught by semantic analysis, here to prevent warning. *)
    | _ -> raise (CompilerError("You can't have a field declaration with type "
                 ^ "\"" ^ string_of_type field_decl.field_type ^ ".\""))

(* Takes a list of field_decls and converts them to a MyPlayer class. *)
let java_of_player field_decls =
    let instance_vars =
        String.concat "\n" (List.map java_of_field_decl field_decls)
    in let assigns =
```

```
        String.concat "\n" (List.map java_of_field_decl_assign field_decls)
    in
        "public class MyPlayer extends Player {\n" ^
        instance_vars ^
        "public MyPlayer(String playerName) {\n"
        ^   "super(playerName);\n" ^
        assigns ^
        "}\n}"

let java_of_args args =
    String.concat ", " (List.map java_of_expr args)

let java_of_call call =
    match call.fname with
    | "output" -> "System.out.println(" ^ java_of_args call.args ^ ")"
    | "input" ->
        let expr, _type =
            List.hd call.args
        in let var_id =
            begin match expr with
            | Var(id) -> id
            | _ -> raise (CompilerError("Bad type passed to input()."))
            end
        in
            begin match _type with
            | BooleanType -> var_id ^ " = Utility.inputBool()"
            | NumberType -> var_id ^ " = Utility.inputInt()"
            | StringType -> var_id ^ " = Utility.inputString()"
            | _ -> raise (CompilerError("Bad type passed to input()."))
            end
    | "quit" ->
        "System.exit(0)"
    | _ -> call.fname ^ "(" ^ java_of_args call.args ^ ")"

let java_of_update = function
    | Assign(id, e) -> java_of_var id ^ " = " ^ java_of_expr e
    | VarDecl(var) -> java_of_type var.var_decl_type ^ " " ^
                      var.var_decl_id ^ " = " ^
                      java_of_expr var.var_decl_value

let rec java_of_stmt stmt =
    match stmt with
    | Call(call) -> java_of_call call ^ ";"
    | Update(update) -> java_of_update update ^ ";"
    | If(e, tb, fb) -> "if (" ^ java_of_expr e ^ ")\n" ^ java_of_block tb
                       ^ "else\n" ^ java_of_block fb
    | While(e, b) -> "while (" ^ java_of_expr e ^ ")\n" ^ java_of_block b
```

```
    | Break -> "break;"
    | Continue -> "continue;"
    | For(a, e, u, b) -> "for (" ^ java_of_update a ^ "; " ^ java_of_expr e
                         ^ "; " ^ java_of_update u ^ ")\n" ^
                         java_of_block b
    | TimesLoop(stmt, expr) -> "for (int i = 0; i < " ^ java_of_expr expr ^
                               "; i++)\n{\n" ^ java_of_stmt stmt ^ "\n}\n"
    | Prepend(e1, e2, draw_source) ->
          let source =
              match draw_source with
              | Ast.Top -> "Set.TOP"
              | Ast.Bottom -> "Set.BOTTOM"
          in
              "Set.prepend(" ^ java_of_expr e1 ^ ", "  ^ source ^ ", " ^
              java_of_expr e2 ^ ");\n"
    | Append(e1, e2, draw_source) ->
          let source =
              match draw_source with
              | Ast.Top -> "Set.TOP"
              | Ast.Bottom -> "Set.BOTTOM"
          in
              "Set.append(" ^ java_of_expr e1 ^ ", " ^ source ^ ", " ^
              java_of_expr e2 ^ ");\n"
and java_of_block block =
    let value =
        String.concat "\n" (List.map java_of_stmt block)
    in
    "{\n" ^ value ^ "}\n"

let java_of_function func =
    let access =
        match func.decl_name with
        | "setup"
        | "round" -> "public"
        | _ -> "private"
    in let formals =
        String.concat ", " (List.map (fun formal ->
                            " " ^ (java_of_type formal.formal_type) ^ " " ^
                            formal.formal_id) func.formals)
    in
        access ^ " void " ^ func.decl_name ^ "(" ^ formals ^ ")\n" ^
        java_of_block func.body

(* Default config values yet to be implemented *)
let java_of_game program =
    let config_vars =
        String.concat "\n" (List.map java_of_config program.configs)
```

```
    in let instance_vars =
        String.concat "" (List.map (fun vd -> java_of_update vd ^ ";")
                            program.vars)
    in let funcs =
        String.concat "\n" (List.map java_of_function program.funcs)
    in
        "import java.util.Scanner;\n" ^
        "import java.util.ArrayList;\n\n" ^
        "public class Game {\n" ^
        "ArrayList<MyPlayer> players;\n" ^
        "Set deck;\n" ^
        "int numberOfPlayers = 4;\n" ^
        "int highestCard = 12;\n" ^
        "boolean ascendingOrder = true;\n" ^
        instance_vars ^ "\n" ^
        "public Game() {\n" ^
        config_vars ^
        "deck = new Deck(highestCard, ascendingOrder);\n" ^
        "deck.shuffle();\n" ^
        "players = new ArrayList<MyPlayer>();\n" ^
        "for(int i = 0; i < numberOfPlayers; i++) {\n" ^
        "players.add(new MyPlayer(\"Player \" + (i+1)));\n" ^
        "}\n" ^
        "}\n\n" ^
        funcs ^
        "}"

let compile lexbuf =
    let program = (* This is our SAST. *)
        Semantic.check_prgm (Parser.program Cache.process lexbuf)
    in let game_file =
        open_out "Game.java"
    in let player_file =
        open_out "MyPlayer.java"
    in
        output_string game_file (java_of_game program);
        output_string player_file (java_of_player program.field_decls)

let _ =
    let input_file =
        open_in Sys.argv.(1)
    in
        compile (Lexing.from_channel input_file)
```

Makefile

```
# a simple "make" command builds the compiler
# "make printer" builds the AST printer

CFLAGS = -c
YACCFLAGS = -v

OBJ = ast.cmo indent.cmo scanner.cmo parser.cmo cache.cmo sast.cmo\
         stdlib.cmo semantic.cmo

compile : $(OBJ) compile.cmo
       ocamlc -o $@ str.cma $(OBJ) compile.cmo

compile.cmo : compile.ml ast.cmi
       ocamlc $(CFLAGS) $<

indent.cmo : indent.ml
       ocamlc $(CFLAGS) indent.ml

cache.cmo : cache.ml scanner.ml parser.ml
       ocamlc $(CFLAGS) cache.ml

semantic.cmo : semantic.ml
       ocamlc $(CFLAGS) semantic.ml

scanner.cmo : scanner.ml parser.cmi indent.cmi
       ocamlc $(CFLAGS) scanner.ml

parser.cmo : parser.ml parser.cmi
       ocamlc $(CFLAGS) parser.ml

parser.cmi : parser.mli ast.cmi
       ocamlc $(CFLAGS) parser.mli

stdlib.cmo : stdlib.ml
       ocamlc $(CFLAGS) stdlib.ml

sast.cmi : sast.ml
       ocamlc $(CFLAGS) $^

sast.cmo: sast.ml
       ocamlc $(CFLAGS) $^

ast.cmi : ast.ml
       ocamlc $(CFLAGS) $^
```

```
ast.cmo : ast.ml
	ocamlc $(CFLAGS) $^

parser.ml parser.mli: parser.mly
	ocamlyacc $(YACCFLAGS) $^

scanner.ml : scanner.mll
	ocamllex $^

.PHONY : printer
printer : $(OBJ) printer.cmo
	ocamlc -o $@ str.cma $(OBJ) printer.cmo

printer.cmo : printer.ml ast.cmo cache.cmo
	ocamlc $(CFLAGS) printer.ml

.PHONY : clean
clean:
	rm compile printer *.cmi *.cmo scanner.ml parser.ml parser.mli *.output *.java
```

do

```
#!/bin/bash

RunJava() {
	mv Game.java MyPlayer.java runtime/
	(cd runtime && make run && make clean)
}

make
./compile $1 && RunJava
make clean
```