The Pip Project

PLT (W4115)
Fall 2008

Frank Wallingford (frw2106@columbia.edu)

Contents

1 Introduction **4**

2 Original Proposal **5**

2.1 Introduction 5

2.2 Description 5

2.2.1 Types 5

2.2.2 Expressions 6

2.2.3 Predicates 6

2.2.4 Actions and Routines 6

2.2.5 Collections 6

2.3 Goal 7

3 Tutorial **8**

3.1 The Game Heading 8

3.2 The Deck 8

3.3 Areas 9

3.4 Actions 9

3.5 Statements 9

3.6 Scoring 10

3.7 The Complete Game 11

4 Reference Manual **12**

4.1 Execution 12

4.2 Lexical Elements 12

4.2.1 ASCII 12

4.2.2 Whitespace 13

4.2.3 Comments 13

4.2.4 Tokens 13

4.3 Type System 14

4.3.1 Properties 14

4.3.2 Types . . . 15
4.4 Syntax and Semantics . . . 18
4.4.1 Game Heading . . . 18
4.4.2 Declarations . . . 18
4.4.3 Statements . . . 21
4.4.4 Expressions . . . 27
5 Project Plan **33**
5.1 Process and Timeline . . . 33
5.2 Style Guide . . . 33
5.3 Roles and Responsibilities . . . 34
5.4 Tools and Languages . . . 34
5.5 Project Log . . . 34
6 Architectural Design **35**
6.1 Components . . . 35
6.2 Interfaces . . . 35
6.3 Highlights . . . 36
7 Test Plan **37**
7.1 Test Automation . . . 37
7.2 Unit Tests . . . 37
7.2.1 Syntactic Tests . . . 37
7.2.2 Semantic Tests . . . 37
7.3 Full Tests . . . 38
7.3.1 Crazy Eights . . . 38
7.3.2 Euchre . . . 42
8 Lessons Learned **74**
8.1 Most Important Lesson . . . 74
8.2 Advice for Future Students . . . 74
9 Appendix **75**
9.1 Interpreter Source . . . 75
9.1.1 Driver . . . 75
9.1.2 Lexer . . . 76
9.1.3 Parser . . . 78
9.1.4 Abstract Syntax Tree . . . 83
9.1.5 Checked Abstract Syntax Tree . . . 89

9.1.6 Semantic Analyzer . . . 92
9.1.7 Interpreter . . . 105
9.1.8 User Interface . . . 117
9.1.9 General Utilities . . . 118
9.2 Unit Tests . . . 120
9.2.1 Syntactic Tests . . . 120
9.2.2 Semantic Tests . . . 125
9.3 Full Tests . . . 148
9.3.1 Crazy Eights . . . 148
9.3.2 Euchre . . . 149
9.4 Build System . . . 151
9.4.1 Top-level Makefile . . . 151
9.4.2 Interpreter Makefiles . . . 152
9.4.3 Support Makefiles . . . 153
9.5 Support Scripts . . . 156
9.5.1 Dependency Fixer . . . 156
9.5.2 Silencer . . . 157
9.5.3 LaTeX Dependency Scanner . . . 158
9.5.4 Testcase Driver . . . 158
9.5.5 OCaml Dependency Ordering Tool . . . 159

Chapter 1

Introduction

The pip language is a programming language that was designed for implementing card games. It contains types, expressions, statements, data structures, and syntactic elements that were all created to make developing card games easier.

This document describes the semester-long project that culminated in the pip interpreter and pip language. The original proposal is reproduced here, followed by a tutorial, the reference manual, and details of the implementation. The paper ends with some reflections and a full source listing of all files produced during the project.

This entire project was written by me (Frank Wallingford) for W4115: Programming Languages and Translators, Fall 2008, Columbia. All of the source files and documents herein are original works that I authored. I worked alone because I am a CVN student.

Chapter 2

Original Proposal

This language proposal is copied here exactly as it was presented at the beginning of the project. The pip language has deviated a bit from that proposal for a few reasons. These changes will become apparent later in the document.

2.1 Introduction

Pip will be a language that can be used to implement card games. It will contain types, expressions, and routines that will include features to facilitate the manipulation of cards, ranks, suits, decks, playing areas, and players.

2.2 Description

The Pip compiler will take as input the rules of a card game written in the Pip language and translate them into instructions for a card game engine to execute. As the card game engine executes the Pip program, one or more users interacting with the engine will be able to play the card game. The card game engine will deal cards, display hands and playing areas, advance through player turns, and allow only valid cards to be played at any given time, based on logic coded up in the Pip program. The card game engine will also be able to determine if the game has ended and who the game winner is.

A simple textual interface will be implemented to facilitate testing and to allow card games to be played in a rudimentary fashion. A graphical interface could be implemented which would allow for a more traditional interface, but that is outside the scope of this project.

There will be no mechanism for implementing artificial intelligence for a player. The Pip language and card game engine will only allow physical players to participate in the card games.

2.2.1 Types

Pip will contain several built-in types. Among them will probably be:

- Rank
- Suit
- Card
- String
- Number
- Boolean
- List (an ordered sequence of any type)
- Collection (a group of name-value properties)

2.2.2 Expressions

Expressions will consist of operators, identifiers, and constants. Common tasks such as list manipulation, testing for list membership, and querying and setting collection properties will done with the built-in operators.

```
card in deck // The "in" operator returns a Boolean
deck[0]      // List member access
deck + card  // List addition; results in new list
deck - card  // List deletion; results in new list
game.name    // Property access; See "Collections" below
```

Cards, including Rank and Suit, will have a direct representation in Pip.

```
A            // Ace, a Rank
S            // Spades, a Suit
A/C          // Ace of Clubs, a Card
2-9/H        // 2 through 9 of Hearts, a List of Cards
*/D          // All Diamonds
3,4,5/*      // All 3's, 4's, and 5's
*/*          // A 52-Card deck
```

The slash combines a Rank and Suit into a Card. If either side of a slash is the asterisk, then the slash results in a list of Cards that match the wildcard.

2.2.3 Predicates

Predicates will be like simple routines that take parameters and evaluate to a Boolean result. They are meant to be short pieces of logic that the card game engine will use at strategic points in the game to decide which cards are legal to play, etc.

```
can_play(Player p, Card c) = c in A/* or c.rank == 8;
```

2.2.4 Actions and Routines

Actions appear in Routines. Actions may move cards from one location to another, show messages to the user, or update properties of players or the game.

Routines will contain control flow, looping constructs, and actions. Routines will be used to advance the state of the game at appropriate times. The card game engine will provide standard actions like deal, display a message, ask for input, and others, and the routines defined in the Pip language will use those actions to control whose turn it is, what message is displayed, and what plays are legal as the game progresses. Routines will not return values.

```
take_turn(Player p) {
  if (p.hand.size < 5) {
    p.hand.add(draw_pile.take(5 - p.hand.size));
  }
  message "Play a card";
  p.get_move();
}
```

2.2.5 Collections

Collections will be used to group name-value pairs together to represent related groups of properties. Collections may describe playing areas, players, and the game itself.

Collections are created to be of a certain kind, and the kind of a collection designates which name-value pairs are valid for the collection. For example, a collection representing a Game will have a name for the game, a list of players, and certain routines and predicates. A collection for a player may also have a name, but it will have different predicates and other properties.

```
a = Area {
  name = "Center";
  show = true;
  stack = false;
```

```
  can_play(Player p, Card c) = ...; // Boolean predicate
}

g = Game {
  name = "Euchre";
  players = 4;
  deck = A-9/*;
  areas = [a];
  take_turn(Player p) { ... } // Action Routine
}
```

The name-value pairs inside collections will be checked by the compiler to ensure that they match the kind of collection they appear in.

2.3 Goal

The goal is to be able to write a few different styles of card games in the Pip language, compile them, and play them using a textual card game engine. The Pip language should be able to easily express a trick-taking game like Euchre and a shedding game like Crazy Eights to be minimally useful.

Chapter 3

Tutorial

Let's build up a simple card game. For this tutorial, I'm going to invent a simple game.

3.1 The Game Heading

Every pip program must start with a Game heading. This tells the pip interpreter what the name of the game is and how many people are required in order to play it.

```
# Let's play a game
Game "Tutorial" requires 2 to 4 players.
```

My game is called "Tutorial" and 2, 3, or 4 players may play it. When pip runs this game, the user will be asked how many players are actually playing, and get names from the user for each player.

You'll also notice that # denotes a comment. Comments extend to the end of the line.

3.2 The Deck

There are expressions in pip that refer to cards by rank and suit or by patterns that expand to entire lists of cards.

```
A~C      # The Ace of Clubs
2~S      # The 2 of Spades
2..4~H   # Three cards - the 2, 3, and 4 of Hearts
%~D      # % is a wildcard. This is all of the Diamonds.
2,3~%    # The 2 and 3 of all suits.
%~%      # All 52 cards.
```

There are also variable declarations that create storage where values can be placed. These are much like variables in most procedural languages.

```
Deck d.
Number count = 2.
```

Each declaration has a type and an identifier. A declaration may also have an optional expression which will become the initial value of the variable.

Putting these together, I can create a deck of cards for my game - my game uses only a few cards. I will also create a counter that I will use later, and start it at 0.

```
Deck d = 2..5~%.
Number count = 0.
```

3.3 Areas

When playing a card game, cards are usually played to some place specific on the table. There may be one or more of these places, and in pip, they are called Areas.

An Area is basically a variable that can hold cards, just like a Deck can. However, an Area also has a label and a few other properties that can be used by the pip interpreter to display the cards in the Area to the user correctly.

```
Area discard labeled "Discard" is faceup.
```

The label is displayed along with any cards in the Area. If the Area is "faceup" then the cards are visible; it may also be "facedown" in which case the cards are not visible.

There are other Area options as well; they can be found in the language reference manual.

3.4 Actions

Now that we've got most of the game set up, we need to describe how the game is played.

Actions are like routines in many languages. Actions are invoked, they execute from beginning to end, and they contain statements that change the state of the game.

A pip program can have any number of actions, but it must contain one called "main". This main action is run when the game begins, and is responsible for doing all of the work.

```
Action main {

}
```

This action needs some statements. They will go between the curly braces.

3.5 Statements

There are many statements available to the pip programmer. Cards can be shuffled and dealt from place to place. Players can be given a chance to play cards or make other decisions. There are also conditional statements and loops to control the flow of execution, statements to display messages to players, and more.

For my game, I need to shuffle the deck, deal cards to the players who are playing, and then play a few hands until there is a winner. Here is the skeleton.

```
Action main {
  shuffle d.
  deal all from d to players.

  forever {
    take_turn().
  }
}
```

This action uses the "shuffle" statement to shuffle the deck we declared earlier, and the "deal" statement to deal out all cards to the players. "players" is a special keyword that contains a list of all of the players currently playing this game.

Finally, the "forever" statement executes its body forever, and in the body we are invoking a new action called "take_turn" that we haven't created yet. After the cards are dealt, "take_turn" will be run over and over until a winner is found.

Let's write up the "take_turn" action one piece at a time.

The first thing we need to know is: whose turn is it? The "players" keyword is a list of players in the game. The first player in the list is the first player to act, and the last player is the dealer.

We can use the "let" statement and a list property expression to make a temporary name for the first player in the list.

```
Action take_turn {
  let p be players->first.
}
```

From that point forward, using "p" as an identifier will get the player that was first in the list.

Note: be sure "take_turn" is earlier in the file than "main" so that "main" can invoke it.

Now, this player needs to play a card. The "play" statement allows a player to play any card from his or her hand to the designated Area so long as it matches a Rule. Rules are special boolean expressions that the pip interpreter will use to decide if a play is legal. Back near our other declarations, we can create a rule that simply says that any card can be played as long as its rank is less than or equal to than the rank of the card on top of the discard pile. In other words, you can only play a card if it is the same or smaller than the card played before you.

```
Rule valid(p, c, cl) = (cl->size == 0) or
                       (c->rank <= cl->top->rank).
```

Any time pip needs to decide if a card "c" can be played from by given player "p" to a given list of cards "cl", pip will evaluate the rule used in the play statement and see if it is true or false.

In my case, if the destination card list is empty, the play is allowed (any card can be played first). Otherwise, the card is only valid if its rank is less than or equal to the rank of the top card on the pile.

Let's create a play statement that uses this rule:

```
Action take_turn {
  let p be players->first.
  play valid from p to discard.
}
```

The last thing we have to do is make sure that a new player gets a turn the next time this action is invoked. The "rotate" statement does just that.

```
Action take_turn {
  let p be players->first.
  play valid from p to discard.
  rotate players.
}
```

After "take_turn" is run once, the first player in the list got a chance to play a card, and then that person ended up at the end of the "players" list. The next time this action is invoked, the next player will get a turn.

3.6 Scoring

Finally, we have to decide on who wins during the game play. The "winner" statement declares a player the winner, but we have to decide on the conditions first.

For my example game, the person who plays the last 2 will win. I'm going to use the "count" variable I created earlier to keep track of how many 2s were played so far.

```
Action take_turn {
  let p be players->first.
  play valid from p to discard.

  if discard->cards->top->rank == 2 {
    count += 1.
    if count == 4 { winner p. }
  }

  rotate players.
}
```

You can see that after a card is played, I check the top of the discard area to see if it was a 2. If so, I add 1 to my count, and if all four 2s have been played, this player (who just played the last one) is the winner.

There is one more small detail to consider for this example game. If the player has no cards he or she can play, the "play" statement won't do anything. In that case we don't want to check to see if a 2 was played, so we first find out if a card could be played at all:

```
Action take_turn {
  let p be players->first.

  if canplay valid from p to discard {
    play valid from p to discard.

    if discard->cards->top->rank == 2 {
      count += 1.
      if count == 4 { winner p. }
    }
  }

  rotate players.
}
```

The "canplay" expression returns true or false depending on whether any cards in the player's hand can be played using the given rule.

3.7 The Complete Game

Here's the complete game:

```
# Let's play a game
Game "Tutorial" requires 2 to 4 players.

Deck d = 2..5~%.
Number count = 0.

Area discard labeled "Discard" is faceup.
Rule valid(p, c, cl) = (cl->size == 0) or
                       (c->rank <= cl->top->rank).

Action take_turn {
  let p be players->first.

  if canplay valid from p to discard {
    play valid from p to discard.

    if discard->cards->top->rank == 2 {
      count += 1.
      if count == 4 { winner p. }
    }
  }

  rotate players.
}

Action main {
  shuffle d.
  deal all from d to players.

  forever {
    take_turn().
  }
}
```

Save that in a file and run it through the pip interpreter. Enjoy!

There are many more expressions, statements, and types available to the pip programmer. Check out the language reference manual for details.

Chapter 4

Reference Manual

A Pip program describes a card game that can be played by one or more players as it is executed. This manual formally describes the syntax and semantics of a Pip program including details about the execution, environment, and type system.

The Pip language is declarative. It consists of types, declarations, statements, and expressions. The statements and declarations follow an English-like natural language made up of many keywords to make it easy to read.

4.1 Execution

A Pip program is executed by the Pip interpreter. During execution, actions are run which contain statements that can manipulate objects, request input from players, produce messages, keep score, and declare a game winner.

The following steps take place during the execution of a Pip program:

1. The Pip source file is loaded and parsed.
2. Players are asked to enter their names, and if the game is played in teams, teams are formed. The interpreter knows how many players to allow based on the `Game Heading`.
3. The game state is initialized. All card objects take on their normal default values.
4. All global variables are initialized.
5. The action named `main` is executed. It may execute statements and other actions, all of which may manipulate the state of the game.

The game is played out according to the `main` action, which should set up the game, advance through turns, keep score, and decide on a winner by executing a `winner` statement.

If the `main` action returns without deciding on a winner, the game is declared to be a tie.

It is an error if no `main` action is declared.

4.2 Lexical Elements

A Pip program is stored in a single file written in the ASCII character set. It consists of a series of tokens separated by whitespace. The tokens are combined to form the semantic elements described later in this document.

4.2.1 ASCII

The basic ASCII character set is defined in ISO/IEC 646, available publicly.

```
char: any of the 128 ASCII characters
```

4.2.2 Whitespace

Whitespace separates tokens. Whitespace is defined as follows:

```
whitespace: space | tab | newline
     space: ASCII 0x20
       tab: ASCII 0x09
   newline: ASCII 0x0A
```

4.2.3 Comments

A comment is a sequence of characters that is ignored by the Pip interpreter. Comments begin with a `#` and continue to the end of the current line. Comments may not begin inside string literals.

```
    comment: hash commentchar*
       hash: ASCII 0x23
commentchar: any char but "\n"
```

4.2.4 Tokens

Tokens are one or more ASCII characters that make up a valid word in a Pip program. A token is matched greedily; at any point in the source program, the longest sequence of characters that makes a valid token is considered as a single token, regardless of whether it would result in an invalid program at some later point.

Punctuation and Expressions

The following tokens are used for punctuation and in expressions:

```
->  =   ==  !=  <   <=  >   >=
+   -   *   /   +=  -=  *=  /=
{   }   (   )   [   ]
,   .   ..  ;   ~   %
```

Keywords

The following tokens are keywords, and may not be used as identifiers:

```
Action      Number      String
Area        Ordering    Suit
Boolean     Player      SuitList
Card        PlayerList  Team
CardList    Rank        TeamList
Deck        RankList
Game        Rule

all         elseif      labeled     requires
and         facedown    leaving     rotate
ask         faceup      let         shuffle
at          for         message     skip
be          forever     not         spreadout
by          from        of          squaredup
canplay     if          or          standard
deal        in          order       starting
defined     is          play        teams
else        label       players     to
                                    winner
```

Identifiers

An identifier is a sequence of alphanumeric ASCII characters that starts with an upper-case or lower-case letter. It may also include underscores.

```
        id: alpha (alpha | num | underscore)*
     alpha: "A" through "Z" | "a" through "z"
       num: "0" through "9"
underscore: ASCII 0x5F
```

Literals

Literal strings, numbers, booleans, ranks, and suits may be specified in part by using the following tokens:

```
    literal: stringlit | numlit | boollit | ranklit | suitlit
  stringlit: '"' stringchar* '"'
 stringchar: ("\" char) | (any char but '"')
     numlit: num+
    boollit: "True" | "False"
    ranklit: "A" | "K" | "Q" | "J"
    suitlit: "C" | "H" | "S" | "D"
```

No identifier may be in the form of a literal.

4.3 Type System

Pip has a complicated type system.

`Number`, `Boolean`, `String`, `Rank`, and `Suit` are *basic types*. Declarations of these types declare an object that can hold a copy of a complete value of these types. Values that are assigned to objects of these types are done by value and copies are made. There is no aliasing of names to these types; every name refers to a unique instance of these values.

`Card`, `Deck`, `Player`, and `Team` are *object types*. Declarations of these types declare a name that references an already-existing object of these types. These names may alias because values that are assigned or passed to actions are passed by reference value, but the object referenced is not copied.

`Area`, `Action`, `Ordering`, and `Rule` are *complex types*. Declarations of these types are each unique and consist of many keywords and values that are needed to initialize the type. These values are not copied or assigned to anything and exist as singletons in the global scope.

Finally, `CardList`, `RankList`, `SuitList`, `PlayerList`, and `TeamList` are *list types*. These each contain elements that match their type, and can be declared and assigned like `object types`. They can also be created by certain expressions.

4.3.1 Properties

All of the types except the `basic types` have properties.

Properties have a name, a type, and a value of their own, and are initialized when the containing type is initialized. Each type below describes the names and types of the properties they contain.

Properties of an object are accessed via the *arrow expression*.

All of the `list types` share the properties described in the general `List` type section. All list properties are read-only.

4.3.2 Types

Number

`Number` is a `basic type`.

The `Number` type represents an integral value of infinite precision. Literals like `0` and `42` have type `Number`.

A `Number` may be implicitly converted to a `Rank` any time one is required. 2 through 10 correspond to the minor ranks. 11 is converted to `J`, 12 is converted to `Q`, 13 is converted to `K`, and 1 is converted to `A`. Any other number will cause a runtime error.

Boolean

`Boolean` is a `basic type`.

The `Boolean` type represents the logical values true and false. The literals `True` and `False` have type `Boolean`.

String

`String` is a `basic type`.

The `String` type represents a sequence of zero or more ASCII characters. Literals like `""` and `"Zaphod Beeblebrox"` have type `String`.

Rank

`Rank` is a `basic type`.

The `Rank` type represents the numeric value of a `Card`. There are thirteen values of type `Rank` which can be written as the literals `2` through `10`, `J`, `Q`, `K`, and `A`.

A `Rank` may be implicitly converted to a `Number` any time one is required. `J` is converted to an 11, `Q` is converted to a 12, `K` is converted to a 13, and `A` is converted to a 1.

Suit

`Suit` is a `basic type`.

The `Suit` type represents the four suits of a `Card`. The literals `C`, `H`, `S`, and `D` all have type `Suit` and represent Clubs, Hears, Spades, and Diamonds, respectively.

Card

`Card` is an `object type`.

There are 52 card objects. They are initialized as a normal 52-card deck with each card having a distinct `Rank` and `Suit` combination before the program begins execution.

Cards are referenced via expressions that name a single card or a group of cards. `A~C` and `%~S` are examples of such expressions.

Cards have the following properties:

Name	Type	Description
rank	Rank	The rank of the card
suit	Suit	The suit of the card
last_played_by	Player	The last player to play this card

If the card has never been played, the value of `last_played_by` is `undefined`.

Deck

The `Deck` type is an alias for the `CardList` type.

Player

`Player` is an `object type`.

An object of type `Player` represents a player in the game and has the following properties:

Name	Type	Description
name	String	The name of the player
hand	CardList	The cards in the player's hand
stash	CardList	A player-specific discard pile
score	Number	The player's current score
team	Team	The team the player is on

There is an object of type `Player` for each player in the game. Each player object is initialized once the interpreter receives input from the player interacting with the program.

A player's `score` starts at `0` and the `hand` and `stash` start out empty. The `team` points to the team the player is on; if the player is not on a team, its value is `undefined`.

Team

`Team` is an `object type`.

An object of type `Team` represents a team in the game and has the following properties:

Name	Type	Description
members	PlayerList	The players on this team
stash	CardList	A team-specific discard pile
score	Number	The team's current score

There is an object of type `Team` for each team in the game. Each team object is initialized once the interpreter receives input from the player(s) interacting with the program.

A team's `score` starts at `0` and the `stash` start out empty.

Area

`Area` is a `complex type`.

Areas are where cards are played. An area contains one or more cards that may or may not have their faces visible to the players. Cards can be transferred from one area to another, or to and from other types objects via certain statements.

An object of type `Area` has the following properties:

Name	Type	Description
name	String	The name of the area
cards	CardList	The cards in the area
is_facedown	Boolean	True if facedown, false if faceup
is_squaredup	Boolean	True if squaredup, false if spreadout

An area's `cards` starts out empty.

Action

`Action` is a `complex type`.

An `Action` is a group of statements that is executed in sequence. Actions are invoked implicitly by the Pip interpreter at certain points, or explicitly via `invocation` statement.

Actions are not manipulated outside of defining and invoking them.

Ordering

`Ordering` is a `complex type`.

An `Ordering` is the type of a declaration that can be used to sort any given `CardList`.

Rule

`Rule` is a `complex type`.

A `Rule` is the type of a declaration that can be used to decide if a certain action is valid.

List

There is no generic `List` type; however, all specific list types below share the following read-only `List` properties:

Name	Type	Description
size	Number	The number of items in the list
first	of item	The first item in the list
last	of item	The last item in the list
top	of item	An alias for last
bottom	of item	An alias for first

CardList

`CardList` is a `list type`.

An object of type `CardList` contains a list of `Card` items.

RankList

`RankList` is a `list type`.

An object of type `RankList` contains a list of `Rank` items.

SuitList

`SuitList` is a `list type`.

An object of type `SuitList` contains a list of `Suit` items.

PlayerList

`PlayerList` is a `list type`.

An object of type `PlayerList` contains a list of `Player` items.

TeamList

`TeamList` is a `list type`.

An object of type `TeamList` contains a list of `Team` items.

4.4 Syntax and Semantics

The main structure of a Pip program is a game heading followed by a series of top-level declarations. These declarations define the game parameters, objects that can be manipulated, cards that are available, and actions that can be executed.

```
pipfile: gameheading declaration*
```

The `Action` declaration contains statements and expressions which manipulate the environment, interact with players, and control the flow of execution.

4.4.1 Game Heading

The game heading must be the first thing in the Pip source file, and there must be only one. It is used to provide general information about the game.

```
gameheading: "Game" name "requires" some "players" "."
           | "Game" name "requires" some "players" "."
           | "Game" name "requires" some "teams" "of" some "."
           | "Game" name "requires" some "teams" "of" some "."
       name: stringlit
       some: numlit ("or" numlit)*
           | numlit "to" numlit
```

Example:

```
Game "Crazy Eights" requires 3 to 6 players.
Game "Foo" requires 2 or 4 teams of 2.
```

Semantics:

The game heading sets up the game's name and possible number of players and/or teams so the Pip interpreter can ask the user for input on the actual number of players and teams as well as the names of those who are playing. This information will be used to initialize the execution environment.

By specifying `x to y`, one is specifying an inclusive range of all valid numbers. By specifying `x or y or ...`, one is specifying that only those elements given are valid.

4.4.2 Declarations

Declarations introduce a new name and create either an object of `basic type`, a reference to an object of `object type` or `list type`, or a singleton object of `complex type`.

The scope of a declaration starts at the end of the declaration and ends at the end of the file. Declarations can only appear at the top level of a pip program.

There are five namespaces: one for areas and variables, one for orderings, one for rules, one for actions, and one for labels.

Declarations that don't specify an initial value start as `undefined`.

It is an error to use an undefined value in any expression.

It is also an error to create multiple declarations of the same name in the same scope and namespace.

Area

The area declaration defines a new object that can hold cards and that displays them to the user in various ways.

```
areadecl: "Area" id "labeled" stringlit "."
        | "Area" id "labeled" stringlit "is" opts "."
    opts: opt ("," opt)*
     opt: "facedown" | "faceup" | "squaredup" | "spreadout"
```

Example:

```
Area drawpile labeled "Draw Pile" is facedown.
```

Semantics:

The literal string is set as the area's `name` property. The name is also used to identify the area visually to the players.

An area that is `facedown` does not have any cards visible to the players. The opposite option is `faceup`, which means the cards are visible to the user. It is an error if both options are given in the same area declaration.

An area that is `squaredup` has all the cards stacked up. If they are `faceup`, only the top card can be seen. The opposite option is `spreadout`, which means all cards are separated. If they are `faceup`, they can all be seen by the players. It is an error if both options are given in the same area declaration.

By default, an area is facedown and squaredup.

Action

The action declaration defines a sequence of statements and names them as a group.

```
actiondecl: "Action" id block
```

Example:

```
Action setup {
  statement.
  statement.
}
```

Semantics:

When an action is invoked via an `invocation` statement, the action executes each statement in turn from top to bottom.

As a side-effect of the rule that the identifier is not in scope until the end of the declaration, actions may not be recursive because they can't refer to themselves.

Rule

The rule declaration defines a rule that can be used to decide if a card can be played at a given point in the game.

```
ruledecl: "Rule" id "(" id, id, id ")" "=" expr "."
```

Example:

```
Rule valid(p, c, cl) = True.
```

Semantics:

When a rule is invoked via a `play` statement, the expression is evaluated and if the result is true, the card can be played. If the result is false, the card can not be played.

The three identifiers are the player, card, and card list, respectively, that are involved in the current play. They are set up by the `play` statement before the rule expression is evaluated.

It is an error if the expression does not have type `Boolean`.

Ordering

The ordering declaration defines a way to put a list of cards in a specific order.

```
orderdecl: "Ordering" id "(" id ")" "=" expr "."
```

Example:

```
Ordering reverse(c) = expr.
```

Semantics:

When an ordering is used in an `order` statement, the expression is evaluated and the result is used to determine the order of the cards being sorted. Each card being ordered is searched for in the resulting list; those cards found closer to the beginning of the list are sorted closer to the front of the result.

The identifier is the cardlist being sorted, and is set up by the `order` statement before the ordering expression is evaluated.

It is an error if the expression does not have type `CardList`.

Generic Variables

Objects of `basic type`, `object type`, and `list type` can be declared and optionally initialized.

```
vardecl: type id "."
       | type id "=" expr "."
```

Example:

```
Player p.
Boolean b = True.
```

Semantics:

The declaration of a variable of `basic type` defines a new variable that can hold a copy of any value of that basic type. Assignments to the variable copy the value being assigned into the variable. Changes to the old variable don't affect the new one.

If a variable of `basic type` is declared with no initial expression, then its initial value is `undefined`.

The declaration of a variable of `object type` or `list type` defines a new variable that can hold a reference to another object of the same type. Assignments to the variable copy the reference; both the old and new variable refer to the same object and changes to that object will be reflected in each. However, a subsequent assignment to one of the variables changes which reference it holds, and that won't affect other variables that still reference the old object.

If a variable of `object type` or `list type` is declared with no initial expression, its initial value is also `undefined`.

If an initial expression is given, it is evaluated and the resulting value becomes the initial value of the new variable. The type of the expression must match the type of the variable or an error will occur.

4.4.3 Statements

Statements are grouped together in `Actions` and can be used to modify the values of objects, manipulate the environment, process input from players, produce messages for players, invoke actions, and control the flow of execution.

blocks

A block is a sequence of zero or more statements.

```
block: "{" statement* "}"
```

Example:

```
{
  statement.
  statement.
}
```

Semantics:

Blocks introduce a new scope that ends at the closing curly brace.

When a block is executed, each statement inside is executed in order.

ask

The ask statement gets input from a player by providing a list of options and letting the player choose one. Each option has an associated block that is executed depending on what the player chooses.

```
  askstmt: "Ask" ref questions
questions: "{" question+ "}"
 question: stringlit (if expr)? block
```

Example:

```
ask player {
  "Do thing A" if expr { statement. }
  "Do thing B"         { statement. }
}
```

Semantics:

When an ask statement is executed, a list of questions is presented to the referenced player.

The set of questions to be asked includes each question with an `if expr`, if the `expr` evaluates to `True`. It also includes each question with no `if expr`; those are always presented.

The player is allowed to select one of the questions. The block associated with the selected question is executed; the other blocks are ignored.

Each `expr` in each `if expr` clause must have type `Boolean` or an error will occur.

assignment

The assignment statement associates a new value or reference with a named object.

```
assignstmt: ref "=" expr "."
```

Example:

```
p = expr.
```

Semantics:

The expression is evaluated. It must produce a value of the same type as the named object, or an error will occur.

For `basic types`, the value is copied into the named object.

For `object types` and `list types`, the result must be a reference to an existing object; the named object is updated to also reference the same existing object as an alias.

compound assignment

A compound assignment statement updates an object with a new value.

```
compoundstmt: ref "+=" expr "."
            | ref "-=" expr "."
            | ref "*=" expr "."
            | ref "/=" expr "."
```

Example:

```
n += 2.
```

Semantics:

The expression is evaluated. The resulting value is combined with the current value of the referenced object to produce a new value. The new value is then stored back into the referenced object.

The expression and the object must be of type `Number` or an error will occur.

deal

The deal statement moves cards from one place to another.

```
dealstmt: "deal" some "from" ref "to" ref "."
    some: "all" | expr
```

Example:

```
deal 1 from p->hand to discardpile.
deal all from deck to drawpile.
deal 2~C from a to b.
```

Semantics:

Some number of cards is moved from the first referenced object to the second referenced object. If an expression is given, it is either the number of cards to move, or a specific card to move; otherwise, if `all` is given, all cards are moved.

If an expression is given, it must have type `Number` or `Card`, or an error will occur. If a numeric expression evaluates to more cards than are available, all of the cards are moved.

Except for the case of a specific card being named, cards are moved one at a time from the top of the source to the top of the destination.

The source reference must have type `CardList`, `Player`, or `Area`, or an error will occur. The destination reference must have type `CardList`, `Player`, `PlayerList`, or `Area`, or an error will occur.

If cards are dealt to or from a player, `player->hand` is implicitly used. If cards are dealt to or from an area, `area->cards` is implicitly used. If cards are dealt to a player list, each player in the list is dealt the given number of cards to their hand. In the latter case, the expression must be numeric or `"all"`.

forever

The forever statement repeats a block over and over.

```
foreverstmt: "forever" block
```

Example:

```
forever {
  statement. statement.
}
```

Semantics:

Executing the forever statement executes the block over and over forever. The only way to exit the forever block is via a `skip to` statement or a `winner` statement.

for

The for statement executes a block once for each item of a list.

```
forstmt: "for" id "in" expr block
       | "for" id "in" expr "starting" "at" expr block
```

Example:

```
for x in list {
  statement. statement.
}
```

Semantics:

For each item of the list, the block is executed. Before the block is executed, a temporary name is introduced and it gets the value of the current list item. The name is no longer in scope after the end of the block.

If the `starting at` form is used, then the given expression is evaluated. It must compare equal (using `==`) to an item in the list or an error will occur. The traversal starts at the first occurrence of that element, walks to the end of the list, starts over at the beginning, and ends one element before the starting element.

If the basic form is used, traversal starts at the beginning of the list and ends at the end of the list.

if

The if statement lets control flow split into optional paths.

```
ifstmt: "if" expr block elseif* else*
elseif: "elseif" expr block
  else: "else" block
```

Example:

```
if expr {
  statement.
} elseif expr {
  statement.
} else {
  statement.
}
```

Semantics:

The first expression is evaluated. If the value of the result is `True`, then the first block is executed and the rest of the statement is ignored.

If the first expression evaluates to `False`, then each `elseif` expression is evaluated in turn, from top to bottom, until one evaluates to `True`. At that point, the corresponding block is executed and the rest of the statement is ignored.

If no expression evaluates to `True` and there is an `else` block, that block is executed. Otherwise, nothing further is done.

All of the expressions must have type `Boolean` or an error will occur.

invoke

The invoke statement calls an action.

```
invokestmt: id "(" ")" "."
```

Example:

```
calculate_score().
```

Semantics:

When an invoke statement is executed, the referenced action is executed, and the control returns to the statement following the invocation statement.

label

The label statement defines a location in a block.

```
labelstmt: "label" id "."
```

Example:

```
label foo.
```

Semantics:

A label statement has no runtime semantics. It exists to mark a location in a block for a `skip` statement to reference. It is an error if more than one label statement with the same name appears in the same action.

let

The let statement temporarily names a value.

```
letstmt: "let" id "be" expr "."
```

Example:

```
let p be players->first.
```

Semantics:

The `let` statement introduces a temporary name for an existing object. The scope of a temporary declaration starts after the `let` statement and ends at the end of the inner-most enclosing scope of the `let` statement.

The expression is evaluated and the resulting value is given the temporary name of an appropriate type.

message

The message statement displays a string.

```
messagestmt: "message" expr "."
           | "message" ref expr "."
```

Example:

```
message "Testing 1 2 3".
message p "You have to act".
```

Semantics:

The expression is evaluated. If the result is not of type `String`, an error will occur.

The first form with no reference displays the string to all players.

If the second form is used, the reference must refer to a player, player list, or team. The message is displayed so only the intended player(s) see it.

order

The order statement sorts a list of cards by following the given ordering.

```
orderstmt: "order" ref "by" id "."
```

Example:

```
order cards by evens_odds.
```

Semantics:

First, a new temporary name is introduced corresponding to the name of the parameter of the ordering identified. Then the ordering's expression is evaluated, which will produce a value of type `CardList`. After the ordering's expression is evaluated, the temporary name is out of scope.

The resulting list of cards is used to sort the referenced list of cards. The given card list is rearranged so that for any two cards a and b, if a comes before b in the result of the ordering, then a is placed before b in the result of this order statement.

The referenced object must have type `CardList` or an error will occur.

The identifier must refer to an object of type `Ordering` or an error will occur.

play

The play statement lets a player play one valid card to a location.

```
playstmt: "play" id "from" ref "to" ref "."
```

Example:

```
play valid from p to discardpile.
```

Semantics:

The referenced player is given the choice to play any of his or her cards to the referenced card list so long as they match the identified rule.

Each card in the player's hand is evaluated against the rule. Temporary names are introduced in which the rule's player, card, and area identifiers are set to refer to the current player, card, and area in question. The rule's expression is evaluated, and if it evaluates to `True`, then the card in question is available for the player to play.

If no cards match the rule, an error occurs. It is wise to guard this statement with a check using `canplay`.

Once the player selects a card for which the rule is true, the card is moved as if it was dealt via a `deal` statement from the player's hand to the referenced card list.

The identifier must reference an object of type `Rule` or an error will occur.

The first reference must reference an object of type `Player` or an error will occur.

The second reference must reference an object of type `CardList`, `Player`, or `Area`, or an error will occur. If a player is given, `player->hand` is implicitly used. If an area is given, `area->cards` is implicitly used.

rotate

The rotate statement rearranges a list in a certain way.

```
rotatestmt: "rotate" ref "."
```

Example:

```
rotate players.
```

Semantics:

The list is rearranged by removing the first item from the list and inserting it at the end of the list so it becomes the last item of the list.

It is an error if the referenced object does not have a list type.

shuffle

The shuffle statement randomly rearranges a list.

```
shufflestmt: "shuffle" ref "."
```

Example:

```
shuffle deck.
```

Semantics:

The referenced list is rearranged randomly. It is an error if the referenced object is not a list type.

skip

The skip statement transfers control to some further point in the current action.

```
skipstmt: "skip" "to" id "."
```

Example:

```
skip to foo.
```

Semantics:

When a skip statement is executed, control transfers to the statement following the corresponding `label` statement. The corresponding `label` statement is the label in the current action with the same identifier.

It is an error if the identifier does not exist as part of some `label` statement at some point further in the current action. It is also an error if there is a `let` statement between the `skip` statement and the corresponding `label` statement.

winner

The winner statement ends the game instantly and displays the winner to all players. Execution of the interpreter stops.

```
winnerstmt: "winner" ref "."
```

Example:

```
winner p.
```

Semantics:

A message is displayed to all players identifying the winner. The game ends and the execution of the interpreter stops.

It is an error if the referenced object is not of type `Player` or `Team`.

4.4.4 Expressions

Expressions can be used to reference objects via identifiers, properties, and card expressions. They are also used to calculate values via arithmetic, compare values, query for information, and create list objects.

Expressions are used in the initialization of declarations and in various statements to provide values to act upon.

If an expression is an identifier, a keyword that represents an object, or a property of an object, it is said to be a `reference`. References are special because they denote objects and can be used in some contexts that require objects where more general expressions are not applicable, like `deal` statements, etc.

References

A reference denotes an object of some type.

```
ref: id
   | ref "->" id
   | "(" expr ")" "->" id
   | builtin
```

Example:

```
player->hand
(J~C)->suit
```

Semantics:

The arrow expression accesses the property in the left-hand sub-expression's resulting object. It is an error if the object is of a type that has no properties, or if it doesn't have the property that the identifier refers to.

The result of a reference expression is an object that may be modified or be used for the value it contains.

Arithmetic

Arithmetic expressions take in two expressions of type `Number` and produce a value of type `Number`.

```
arithexpr: expr "+" expr
         | expr "-" expr
         | expr "*" expr
         | expr "/" expr
```

Example:

```
2 + 2
3 / 10
```

Semantics:

The left and right sub-expression are evaluated. The operation is performed, and its result is the result of the arithmetic expression.

Division is performed using truncation towards zero. It is a runtime error if the right-hand operand is zero.

It is an error if the operands do not have type `Number`.

Comparison Expressions

Logical expressions take in two expressions of the same type and produce a value of type `Boolean`.

```
compexpr: expr "=="  expr
        | expr "!="  expr
        | expr "<"   expr
        | expr "<="  expr
        | expr ">"   expr
        | expr ">="  expr
        | expr "and" expr
        | expr "or"  expr
        | "not" expr
```

Example:

```
S == C
4 < 10
True and False
```

Semantics:

The left and right sub-expression are evaluated. The comparison is performed, and its result is the result of the comparison expression.

For `basic types`, values are compared. For `list types`, each element is compared recursively. For `object types`, comparison is only done by identity, not structurally.

For `==` and `!=`, both operands can be any type except the `complex types`.

For `<`, `<=`, `>`, and `>=`, the operands can be of type `Number` or `Rank`.

For `and`, `or`, and `not`, the operand(s) must be of type `Boolean`. The `and` and `or` perform short-circuit evaluation from left-to-right - as soon as the answer is known, the remaining expressions are not evaluated.

It is an error if the operands do not have the same type.

in

The in expression tests list membership.

```
inexpr: expr "in" expr
```

Example:

```
2 in [2; 3; 4]
```

Semantics:

The left and right sub-expression are evaluated. The left sub-expression should evaluate to an item and the right sub-expression should evaluate to a list. The result if this expression is `True` if the item is in the list (as compared by `==`), and `False` if not.

It is an error if the first operand's type is not the same as the element type of the list type of the second operand.

There is one exception: If the type of the right-hand sub-expression is `CardList`, then the type of the left-hand sub-expression may also be `Rank` or `Suit`. In these cases, the result is `True` if any card in the card list has the rank or suit that the left-hand expression evaluated to.

canplay

The canplay expression tests whether a `play` statement would be possible.

```
canplayexpr: "canplay" id "from" ref "to" ref
```

Example:

```
canplay valid from p to discardpile
```

Semantics:

This expression follows similar semantics to the `play` statement. If any cards in a similar `play` statement could be played following the identified rule, then this expression evaluates to `True`. Otherwise, it evaluates to `False`.

The same errors and restrictions apply here that apply to the `play` statement.

defined

The defined expression tests whether a variable or property has been assigned a value.

```
definedexpr: "defined" ref
```

Example:

```
defined p
```

Semantics:

If the referenced object has the value `undefined`, then this expression evaluates to `False`. Otherwise, it evaluates to `True`.

Card Expressions

Card expressions provide a concise way to refer to cards.

```
cardexpr: rank "~" suit
        | "%"  "~" suit
        | rank "~" "%"
        | "%"  "~" "%"
    rank: ref | numlit | ranklit | rank ".." rank | rank "," rank
    suit: ref | suitlit | suit "," suit
```

Example:

```
J~C
2,3,4~S
4..10~C,H
%~D
```

Semantics:

The expressions are evaluated. For `..`, the left and right side must evaluate to type `Rank`, and the result is a value of type `RankList` of all ranks in the range, inclusive.

For `,`, the left and right sides must evaluate to type `Rank`, `RankList`, `Suit`, or `SuitList`. The result is a new list of the appropriate type with all items from both sub-expressions included.

Finally, the `~` operator constructs a value of type `Card` (if the sub-expressions were not lists) or of type `CardList` (if either of the sub-expressions were of type list). The `Card` or `CardList` will contain references to all of the cards that can be created from all of the combinations of the given ranks and suits.

The token `%` may appear on the left of `~` to represent a wildcard pattern that matches cards of all ranks that also match the given suit, and on the right of `~` to represent a wildcard pattern that matches cards of all suits that also match the given rank. The sequence `%~%` matches all ranks and suits, producing a list of all cards.

It is an error if the rank expressions do not have type `Rank` or `RankList`, or if the suit expressions do not have type `Suit` or `SuitList`.

List Expressions

List expressions provide a convenient way of constructing lists.

```
listexpr: "[" (expr (";" expr)*)? "]"
```

Example:

```
[C; H; S]
```

Semantics:

All of the sub-expressions are evaluated. The value of this expression is a reference to a new list object with appropriate type.

It is an error if all of the sub-expressions don't have the same basic type. Some may be element type and some may be list types but each must have the same basic type.

All of the list items and sub-lists are concatenated to produce the result.

Literals

Literal expressions provide convenient ways of constructing values known at compile-time.

Literals can be of type `Number`, `Boolean`, `Rank`, `Suit`, or `String`.

The syntax for literals was given in the `Literals` section of the `Lexical Elements` portion of this document.

String literals deserve special mention. There are two character sequences inside a string literal that require special treatment, and some types automatically convert to strings when needed.

First, any nested curly braces inside string literals are expanded as identifiers. The identified object is converted to a string (according to the following rules) and the string representation is placed in the string (the curly braces are removed).

Second, any `\n` or `\t` two-character sequence is converted into an ASCII 0x0A and ASCII 0x09, respectively (newline and horizontal tab).

The following types can be converted to strings when interpolated inside curly braces in a string literal:

Type	String Result
Number	The decimal representation
Boolean	"True" or "False"
Rank	"Two", ..., "Ten", "Jack", "Queen", "King", "Ace"
Suit	"Clubs", "Hearts", "Spades", "Diamonds"
Card	"Jack of Clubs", etc
Player	Player's name
Team	"Team (player names)"

Built-in Expressions

The `players` keyword has type `PlayerList` and contains a reference to each player in the game. The current dealer is last in the list, and the person left of the dealer is first in the list. Each player in the list is left of the preceding player in the list.

The `teams` keyword has type `TeamList` and contains a reference to each team in the game, in an unspecified order.

The `standard` keyword has type `CardList` and contains a reference to each card in a standard deck.

It is an error to assign to any of these references. However, one may change the properties or otherwise modify the objects referenced.

Operator Precedence and Associativity

This table describes the precedence and associativity of the operators listed above. The operators closer to the top of the table have a higher precedence than those closer to the bottom of the table.

Operators listed in the same row of the table have the same precedence and are parsed according to their associativity, just as repeated operators are.

Operator(s)	Associativity
`->`	left-to-right
`..`	none (can’t repeat)
`,`	left-to-right
`* /`	left-to-right
`+ -`	left-to-right
`== != < <= > >= in`	none (can’t repeat)
`not`	none (can’t repeat)
`and`	left-to-right
`or`	left-to-right

Chapter 5

Project Plan

5.1 Process and Timeline

Since I worked alone on this project, I didn't set up or follow a rigid process. I set up the weekly timeline based on the deliverables.

- Sept 8-15: Decide on an idea and roughly the syntax
- Sept 15-22: Write Proposal
- Sept 24: Proposal Due
- Sept 24-29: Write representative programs as goals
- Sept 29-Oct 6: Decide on tokens and keywords; write lexer
- Oct 6-13: Write parser without AST and test on programs
- Oct 13-20: Write Reference Manual
- Oct 22: Reference Manual Due
- Oct 27-Nov 3: Write the AST and hook it to the parser
- Nov 3-Nov 10: Test the AST
- Nov 10-Nov 24: Write the Semantic Analyzer
- Nov 24-Dec 1: Test the Semantic Analyzer
- Dec 1-Dec 8: Write the Interpreter
- Dec 8-Dec 15: Test the Interpreter
- Dec 15-19: Write Final Report
- Dec 19: Final Report Due

5.2 Style Guide

Since I worked alone, I followed my own style. I used modules to separate the large elements from each other (the lexer, the parser, the AST, the checked AST, the semantic analyzer, the interpreter, the runtime support for the interpreter, and the user interface).

Inside files I used many functions where they made sense and I commented everything that exists at the top level.

I tried to use a consistent naming scheme so I could understand my own code better as I revisited it.

5.3 Roles and Responsibilities

Since I worked alone, I had all of the roles and responsibilities:

- Environment setup - Makefiles, test scripts, etc
- Design, Implementation, Test, and Debug of all modules
- Documentation

5.4 Tools and Languages

The main program is written in Objective Caml. Since pip is an interpreter, the runtime support is that of Objective Caml plus a Ui module I wrote to handle user input and output so it could be easily replaced with a graphical user interface. Tools like ocamlyacc, ocamllex, ocamlc, ocamldep, and ocamlobjinfo were all used from the Objective Caml toolchain.

The build system used GNU Make and a series of custom Makefiles I wrote for this project. The Makefiles built the pip program, ran the automated tests, and generated the documentation from the LaTeXsource files.

Some scripts were written to aid the build system. A test script was responsible for executing a program and comparing its output to a given log file. A custom script was used to order Objective Caml object files correctly before passing them to the Objective Caml linker because order is important, and it could be done automatically. Yet another script dealt with fixing up the output of ocamldep so it could be used with the rest of the build system seamlessly. These were all written in Perl.

The project was kept entirely under version control using the Subversion program.

Finally, the shell, environment, and misc tools all came from developing on the Linux operating system.

5.5 Project Log

Here are the important milestones as taken from my Subversion log:

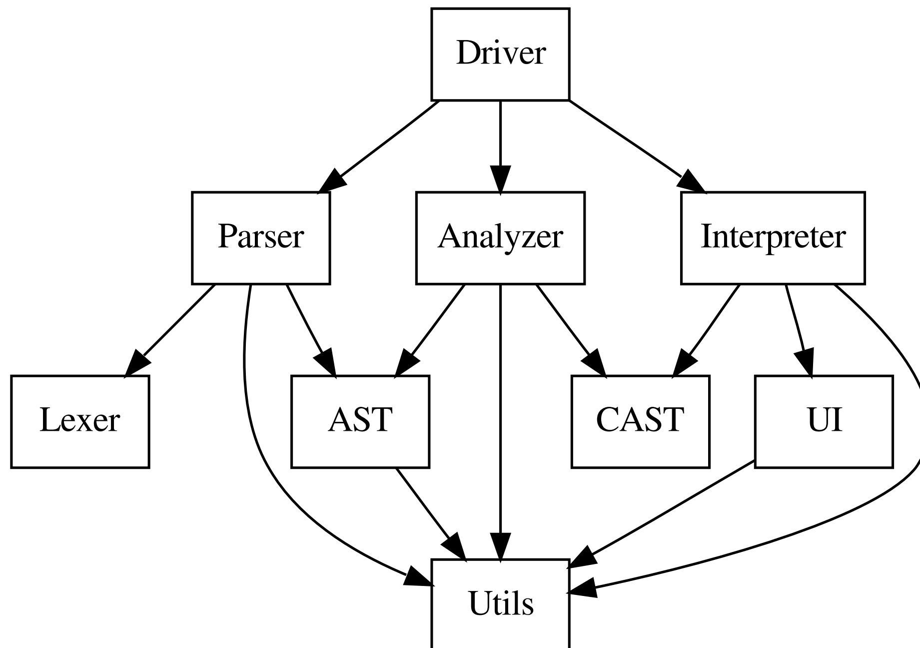
- September 24: Project Proposal completed
- October 17: Build system completed
- October 19: Representative programs completed
- October 21: Lexer and parser completed; test programs parse successfully
- October 22: Reference Manual completed
- October 25: AST completed; programs tested
- November 5: AST pretty-printer completed
- November 16: AST testcases completed
- December 2: Checked AST and semantic analyzer completed
- December 9: Semantic analyzer tests completed
- December 14: Interpreter completed
- December 15: Interpreter tests completed
- December 18: Final Report completed

I slipped on a few dates but the overall structure of my log roughly followed the timeline.

Chapter 6

Architectural Design

6.1 Components



6.2 Interfaces

The Driver sits on top of the whole program, parses the command-line options, and directs the execution of the pip interpreter.

First, the Driver invokes the Parser, which, along with the Lexer, generates the AST data structure defined in the AST module.

The Driver either prints the AST as a string (during testing) or passes the AST to the Semantic Analyzer module.

The Semantic Analyzer module checks for many errors that can be caught at compile-time. The analyzer resolves all identifiers, does bottom-up type checking to catch all type errors, ensures a “main” action exists, checks all “skip” statements

have a corresponding "label" (and that all "label" statements are used), etc. The result of this analysis is a CAST structure ("Checked AST") that is defined in the CAST module, which is passed back to the Driver.

Finally, the Driver invokes the Interpreter, which executes the CAST structure directly. Any time input or output needs to be done, routines from the User Interface module ("Ui") are invoked. This is so that a graphical user interface can be plugged in easily in place of the existing textual interface.

All of the modules use common utilities from the Utils module to manipulate lists, integer ranges, and other minor tasks.

6.3 Highlights

I felt like a few features ended up being especially elegant in their design and/or implementation.

String literals are fully parsed via a few mutually-recursive lexer rules in "scanner.mll". All character escape conversions and embedded curly brace escapes are fully handled before the token even reaches the parser.

Positions for all declarations, statements, and expressions are kept in the AST and Checked AST throughout the program. Errors related to specific constructs are accompanied by their position, making the error messages very useful and errors easy to locate.

Name resolution happens fully during the semantic analysis phase. Global variables end up pointing directly at their runtime storage and local variables end up being offsets into the activation record's array of local variables. Actions, Rules, and Orderings are all resolved directly to their runtime structures. During execution, no hash tables are needed to look up identifiers.

Local variables are resolved to an index in a local variable array. At runtime, a new local variable array is created upon entry to each Action; this is similar to an activation record. Even though variables in nested scopes may have the same name in the same routine, they are resolved to separate indices in this local variable array, much like C would. This alleviates the need to introduce local variables at every scope and only requires they only be maintained when entering and leaving an Action.

Fields are resolved in a similar way: they become an index into an array of fields. Runtime values of object type simply contain an array of values representing the fields, and field references are simply lookups into this array using the resolved index.

Chapter 7

Test Plan

7.1 Test Automation

Unit tests were automated using Makefiles and a support script. The various source files were run through the pip interpreter, and the output was compared against a "golden log" that represented valid output. These golden logs were generated by running pip, examining the output, and blessing it via visual inspection. One "make tests" command executes the entire test suite.

7.2 Unit Tests

Small unit tests were written to test various parts of the syntax and semantics of the pip interpreter. Full code listings for all of the unit tests can be found in the appendix.

Some of the unit tests were designed to ensure the lexer was tokenizing the files correctly. For example, one test ensures that invalid characters in the input file are diagnosed correctly with a file and position.

Some of the tests are valid pip programs, and they exist to ensure that pip doesn't raise any errors where none should occur. These were written to test all of the constructs that the reference manual claims are supported by the pip language.

Other unit tests are invalid pip programs, and they exist to ensure that pip raises the appropriate error when one should occur. These were written to test all of the syntactic and semantic errors that the reference manual implicitly or explicitly dictated.

7.2.1 Syntactic Tests

The pip interpreter has a few options to aid testing. If the interpreter is run with the "–print-ast" option, then processing stops after the parsing phase as soon as the AST is built. The AST is converted back into a string representing the source file and printed out; the interpreter then exits.

This string representation was designed to closely resemble the original program. The automated tests would run a source program through the interpreter using the "–print-ast" option; then, the output, which should represent the same program in source form again, was run through the interpreter a second time to ensure that it produced the same result when printed right back out again. This fix point test ensured that the AST was constructed correctly and the AST-printing routines were also correct.

7.2.2 Semantic Tests

The pip interpreter also has an "–analyze" option. This option causes the pip interpreter to stop after the semantic analyzer converts the AST to the Checked AST (CAST). Once the CAST has been successfully constructed, the interpreter exits.

Unit tests run with this option were designed to test name resolution, type checking, skip/label matching, and other various semantic properties.

7.3 Full Tests

The goal of the pip interpreter was to write card games, so two full card games were written in the pip language: Crazy Eights and Euchre. Full code listings for these two programs can be found in the appendix.

7.3.1 Crazy Eights

Description

Crazy Eights is a shedding game where players try to get rid of cards by playing them to a central area in turn until one player runs out of cards. This is a simple game and was chosen to flush out the basic ideas of the language: cards, a standard deck, one area to play cards, players but no teams, and no scoring. Advanced features like ordering a list of cards, using teams, using different decks, and keeping score are not required for Crazy Eights.

Demonstration

Here is a log of me playing Crazy Eights. The source listing for this pip program is in the appendix.

There are a few points to mention:

- The user input is the text following "Enter a name:" and "Your choice?" - the rest is output produced by the interpreter.
- At one point I enter invalid input. The interpreter responds accordingly and requests valid input again.
- The only cards that are offered to play are those that are legal plays.
- A player must draw until he has a valid card to play.
- An 8 may be played on any card. Otherwise, the card played must match the top card in rank or suit.
- The winner is the first to get rid of his cards.

Keep in mind that the ideal interface would be graphical - the user would be clicking on cards to play, the discard pile would be shown as a stack of cards with the top card visible, and messages would be presented directly to the appropriate user. For this project, a textual interface was enough to exercise the interpreter and language.

Listing 7.1: Interactive Crazy Eights Run

```
$ ./bin/pip ./tests/full/crazy_eights.pip
----------------------------------------
Welcome to Crazy Eights
----------------------------------------
How many players are playing?
  2
  3
  4
Your choice? 2
This game has 2 players.
Enter a name for Player 1: Frank
Enter a name for Player 2: Not Frank

Dealing 7 cards to everyone.

It is Frank's turn
Frank: You are holding the following cards:
  Ace of Clubs
  Seven of Diamonds
  Two of Spades
  Three of Diamonds
  Queen of Diamonds
  Five of Spades
  Six of Hearts
Frank: The top card of the discard pile is Queen of Spades.
Frank: Choose a card:
  Two of Spades
```

```
  Queen of Diamonds
  Five of Spades
Your choice? Queen of Diamonds
Frank played Queen of Diamonds.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Ten of Diamonds
  Ten of Spades
  Two of Hearts
  Four of Clubs
  Ten of Clubs
  Ace of Spades
  King of Spades
Not Frank: The top card of the discard pile is Queen of Diamonds.
Not Frank: Choose a card:
  Ten of Diamonds
Your choice? Ten of Diamonds
Not Frank played Ten of Diamonds.

It is Frank's turn
Frank: You are holding the following cards:
  Ace of Clubs
  Seven of Diamonds
  Two of Spades
  Three of Diamonds
  Five of Spades
  Six of Hearts
Frank: The top card of the discard pile is Ten of Diamonds.
Frank: Choose a card:
  Seven of Diamonds
  Three of Diamonds
Your choice? Seven of Diamonds
Frank played Seven of Diamonds.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Ten of Spades
  Two of Hearts
  Four of Clubs
  Ten of Clubs
  Ace of Spades
  King of Spades
Not Frank: The top card of the discard pile is Seven of Diamonds.
Not Frank: You have nothing you can play.
Not Frank: Choose an action:
  Draw
Your choice? Seven of Diamonds
I'm sorry, I didn't understand.
Not Frank: Choose an action:
  Draw
Your choice? Draw
Not Frank: You picked up Seven of Hearts.
Not Frank: You are holding the following cards:
  Seven of Hearts
  Ten of Spades
  Two of Hearts
  Four of Clubs
  Ten of Clubs
  Ace of Spades
  King of Spades
Not Frank: The top card of the discard pile is Seven of Diamonds.
Not Frank: Choose a card:
  Seven of Hearts
Your choice? Seven of Hearts
Not Frank played Seven of Hearts.

It is Frank's turn
Frank: You are holding the following cards:
  Ace of Clubs
  Two of Spades
  Three of Diamonds
  Five of Spades
  Six of Hearts
Frank: The top card of the discard pile is Seven of Hearts.
Frank: Choose a card:
```

```
  Six of Hearts
Your choice? Six of Hearts
Frank played Six of Hearts.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Ten of Spades
  Two of Hearts
  Four of Clubs
  Ten of Clubs
  Ace of Spades
  King of Spades
Not Frank: The top card of the discard pile is Six of Hearts.
Not Frank: Choose a card:
  Two of Hearts
Your choice? Two of Hearts
Not Frank played Two of Hearts.

It is Frank's turn
Frank: You are holding the following cards:
  Ace of Clubs
  Two of Spades
  Three of Diamonds
  Five of Spades
Frank: The top card of the discard pile is Two of Hearts.
Frank: Choose a card:
  Two of Spades
Your choice? Two of Spades
Frank played Two of Spades.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Ten of Spades
  Four of Clubs
  Ten of Clubs
  Ace of Spades
  King of Spades
Not Frank: The top card of the discard pile is Two of Spades.
Not Frank: Choose a card:
  Ten of Spades
  Ace of Spades
  King of Spades
Your choice? Ace of Spades
Not Frank played Ace of Spades.

It is Frank's turn
Frank: You are holding the following cards:
  Ace of Clubs
  Three of Diamonds
  Five of Spades
Frank: The top card of the discard pile is Ace of Spades.
Frank: Choose a card:
  Ace of Clubs
  Five of Spades
Your choice? Ace of Clubs
Frank played Ace of Clubs.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Ten of Spades
  Four of Clubs
  Ten of Clubs
  King of Spades
Not Frank: The top card of the discard pile is Ace of Clubs.
Not Frank: Choose a card:
  Four of Clubs
  Ten of Clubs
Your choice? Ten of Clubs
Not Frank played Ten of Clubs.

It is Frank's turn
Frank: You are holding the following cards:
  Three of Diamonds
  Five of Spades
Frank: The top card of the discard pile is Ten of Clubs.
Frank: You have nothing you can play.
```

```
Frank: Choose an action:
  Draw
Your choice? Draw
Frank: You picked up Ten of Hearts.
Frank: You are holding the following cards:
  Ten of Hearts
  Three of Diamonds
  Five of Spades
Frank: The top card of the discard pile is Ten of Clubs.
Frank: Choose a card:
  Ten of Hearts
Your choice? Ten of Hearts
Frank played Ten of Hearts.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Ten of Spades
  Four of Clubs
  King of Spades
Not Frank: The top card of the discard pile is Ten of Hearts.
Not Frank: Choose a card:
  Ten of Spades
Your choice? Ten of Spades
Not Frank played Ten of Spades.

It is Frank's turn
Frank: You are holding the following cards:
  Three of Diamonds
  Five of Spades
Frank: The top card of the discard pile is Ten of Spades.
Frank: Choose a card:
  Five of Spades
Your choice? Five of Spades
Frank played Five of Spades.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Four of Clubs
  King of Spades
Not Frank: The top card of the discard pile is Five of Spades.
Not Frank: Choose a card:
  King of Spades
Your choice? King of Spades
Not Frank played King of Spades.

It is Frank's turn
Frank: You are holding the following cards:
  Three of Diamonds
Frank: The top card of the discard pile is King of Spades.
Frank: You have nothing you can play.
Frank: Choose an action:
  Draw
Your choice? Draw
Frank: You picked up Eight of Hearts.
Frank: You are holding the following cards:
  Eight of Hearts
  Three of Diamonds
Frank: The top card of the discard pile is King of Spades.
Frank: Choose a card:
  Eight of Hearts
Your choice? Eight of Hearts
Frank played Eight of Hearts.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Four of Clubs
Not Frank: The top card of the discard pile is Eight of Hearts.
Not Frank: You have nothing you can play.
Not Frank: Choose an action:
  Draw
Your choice? Draw
Not Frank: You picked up Eight of Clubs.
Not Frank: You are holding the following cards:
  Eight of Clubs
  Four of Clubs
Not Frank: The top card of the discard pile is Eight of Hearts.
```

```
Not Frank: Choose a card:
  Eight of Clubs
Your choice? Eight of Clubs
Not Frank played Eight of Clubs.

It is Frank's turn
Frank: You are holding the following cards:
  Three of Diamonds
Frank: The top card of the discard pile is Eight of Clubs.
Frank: You have nothing you can play.
Frank: Choose an action:
  Draw
Your choice? Draw
Frank: You picked up Queen of Hearts.
Frank: You are holding the following cards:
  Queen of Hearts
  Three of Diamonds
Frank: The top card of the discard pile is Eight of Clubs.
Frank: You have nothing you can play.
Frank: Choose an action:
  Draw
Your choice? Draw
Frank: You picked up Nine of Diamonds.
Frank: You are holding the following cards:
  Nine of Diamonds
  Queen of Hearts
  Three of Diamonds
Frank: The top card of the discard pile is Eight of Clubs.
Frank: You have nothing you can play.
Frank: Choose an action:
  Draw
Your choice? Draw
Frank: You picked up Three of Spades.
Frank: You are holding the following cards:
  Three of Spades
  Nine of Diamonds
  Queen of Hearts
  Three of Diamonds
Frank: The top card of the discard pile is Eight of Clubs.
Frank: You have nothing you can play.
Frank: Choose an action:
  Draw
Your choice? Draw
Frank: You picked up Queen of Clubs.
Frank: You are holding the following cards:
  Queen of Clubs
  Three of Spades
  Nine of Diamonds
  Queen of Hearts
  Three of Diamonds
Frank: The top card of the discard pile is Eight of Clubs.
Frank: Choose a card:
  Queen of Clubs
Your choice? Queen of Clubs
Frank played Queen of Clubs.

It is Not Frank's turn
Not Frank: You are holding the following cards:
  Four of Clubs
Not Frank: The top card of the discard pile is Queen of Clubs.
Not Frank: Choose a card:
  Four of Clubs
Your choice? Four of Clubs
Not Frank played Four of Clubs.
The game was won by Not Frank
```

7.3.2 Euchre

Description

Euchre is a trick-taking game that is a bit more complicated. Teams are formed, a special deck is used, and score must be kept. Cards also have to be ordered to determine the trick winner.

Demonstration

Here is a log of me playing Euchre. The source listing for this pip program is in the appendix.

Some quick notes about the complicated rules that are represented and demonstrated here:

- Euchre is played with 9 through A of all suits.
- Euchre is played in teams of 2.
- Cards must "follow suit" when played into tricks.
- Score are team-based and depend on who "made the trump suit" and who collected the tricks.
- The Jacks are special and behave differently depending on if their color is trump or not.
- If no one makes trump, the dealer is forced to. He has no "Pass" option the last time around.
- A team wins when they make 10 points.

Listing 7.2: Interactive Euchre Run

```
$ ./bin/pip ./tests/full/euchre.pip
——————————————————————————————————————————
Welcome to Euchre
——————————————————————————————————————————
This game has 2 players per team.
How many teams are playing?
  2
Your choice? 2
This game has 2 teams.
Enter a name for Player 1 on team 1: Al
Enter a name for Player 2 on team 1: Bob
Enter a name for Player 1 on team 2: Carl
Enter a name for Player 2 on team 2: David

Dealing 5 cards to everyone.
Dealer is David.
Top of kitty is Jack of Clubs. Choosing trump.
Al: Your hand contains:
  King of Hearts
  King of Clubs
  Nine of Spades
  Jack of Diamonds
  Queen of Diamonds
Al: Choose an action:
  Order up Jack of Clubs to David's hand
  Pass
Your choice? Pass
Carl: Your hand contains:
  Ten of Clubs
  Ace of Spades
  Ten of Spades
  Ace of Hearts
  Nine of Clubs
Carl: Choose an action:
  Order up Jack of Clubs to David's hand
  Pass
Your choice? Pass
Bob: Your hand contains:
  Ace of Clubs
  Queen of Hearts
  King of Diamonds
  Queen of Spades
  Nine of Diamonds
Bob: Choose an action:
  Order up Jack of Clubs to David's hand
  Pass
Your choice? Jack of Clubs to David's hand
I'm sorry, I didn't understand.
Bob: Choose an action:
  Order up Jack of Clubs to David's hand
  Pass
```

```
Your choice? Order up Jack of Clubs to David's hand
David: You acquired Jack of Clubs. You must discard something.
David: Choose a card:
  Jack of Spades
  Ten of Hearts
  King of Spades
  Jack of Hearts
  Ace of Diamonds
Your choice? King of Spades
David played King of Spades.

Trump is Clubs.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  King of Hearts
  King of Clubs
  Nine of Spades
  Jack of Diamonds
  Queen of Diamonds
Your choice? Jack of Diamonds
Al played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Al
Carl: Choose a card:
  Ten of Clubs
  Ace of Spades
  Ten of Spades
  Ace of Hearts
  Nine of Clubs
Your choice? Ace of Hearts
Carl played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by Carl
  Jack of Diamonds, played by Al
Bob: Choose a card:
  King of Diamonds
  Nine of Diamonds
Your choice? King of Diamonds
Bob played King of Diamonds.
Cards played so far:
  King of Diamonds, played by Bob
  Ace of Hearts, played by Carl
  Jack of Diamonds, played by Al
David: Choose a card:
  Ace of Diamonds
Your choice? Ace of Diamonds
David played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by David
  King of Diamonds, played by Bob
  Ace of Hearts, played by Carl
  Jack of Diamonds, played by Al
David took the trick with Ace of Diamonds.

Trump is Clubs.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
David starts the trick.
David: Choose a card:
  Jack of Clubs
  Jack of Spades
  Ten of Hearts
  Jack of Hearts
Your choice? Jack of Hearts
David played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by David
Al: Choose a card:
  King of Hearts
Your choice? King of Hearts
Al played King of Hearts.
Cards played so far:
```

```
  King of Hearts, played by Al
  Jack of Hearts, played by David
Carl: Choose a card:
  Ten of Clubs
  Ace of Spades
  Ten of Spades
  Nine of Clubs
Your choice? Ace of Spades
Carl played Ace of Spades.
Cards played so far:
  Ace of Spades, played by Carl
  King of Hearts, played by Al
  Jack of Hearts, played by David
Bob: Choose a card:
  Queen of Hearts
Your choice? Queen of Hearts
Bob played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Bob
  Ace of Spades, played by Carl
  King of Hearts, played by Al
  Jack of Hearts, played by David
Al took the trick with King of Hearts.

Trump is Clubs.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Al starts the trick.
Al: Choose a card:
  King of Clubs
  Nine of Spades
  Queen of Diamonds
Your choice? Nine of Spades
Al played Nine of Spades.
Cards played so far:
  Nine of Spades, played by Al
Carl: Choose a card:
  Ten of Spades
Your choice? Ten of Spades
Carl played Ten of Spades.
Cards played so far:
  Ten of Spades, played by Carl
  Nine of Spades, played by Al
Bob: Choose a card:
  Queen of Spades
Your choice? Queen of Spades
Bob played Queen of Spades.
Cards played so far:
  Queen of Spades, played by Bob
  Ten of Spades, played by Carl
  Nine of Spades, played by Al
David: Choose a card:
  Jack of Clubs
  Jack of Spades
  Ten of Hearts
Your choice? Jack of Spades
David played Jack of Spades.
Cards played so far:
  Jack of Spades, played by David
  Queen of Spades, played by Bob
  Ten of Spades, played by Carl
  Nine of Spades, played by Al
David took the trick with Jack of Spades.

Trump is Clubs.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 2 tricks so far.
David starts the trick.
David: Choose a card:
  Jack of Clubs
  Ten of Hearts
Your choice? Jack of Clubs
David played Jack of Clubs.
Cards played so far:
```

```
  Jack of Clubs, played by David
Al: Choose a card:
  King of Clubs
Your choice? King of Clubs
Al played King of Clubs.
Cards played so far:
  King of Clubs, played by Al
  Jack of Clubs, played by David
Carl: Choose a card:
  Ten of Clubs
  Nine of Clubs
Your choice? Nine of Clubs
Carl played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Carl
  King of Clubs, played by Al
  Jack of Clubs, played by David
Bob: Choose a card:
  Ace of Clubs
Your choice? Ace of Clubs
Bob played Ace of Clubs.
Cards played so far:
  Ace of Clubs, played by Bob
  Nine of Clubs, played by Carl
  King of Clubs, played by Al
  Jack of Clubs, played by David
David took the trick with Jack of Clubs.

Trump is Clubs.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 3 tricks so far.
David starts the trick.
David: Choose a card:
  Ten of Hearts
Your choice? Ten of Hearts
David played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by David
Al: Choose a card:
  Queen of Diamonds
Your choice? Queen of Diamonds
Al played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by Al
  Ten of Hearts, played by David
Carl: Choose a card:
  Ten of Clubs
Your choice? Ten of Clubs
Carl played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Carl
  Queen of Diamonds, played by Al
  Ten of Hearts, played by David
Bob: Choose a card:
  Nine of Diamonds
Your choice? Nine of Diamonds
Bob played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by Bob
  Ten of Clubs, played by Carl
  Queen of Diamonds, played by Al
  Ten of Hearts, played by David
Carl took the trick with Ten of Clubs.
Team (Al, Bob) took 1 tricks.
Team (Carl, David) took 4 tricks.
  Team (Carl, David) scores 2 points.
The scores:
  Team (Al, Bob) has 0 points.
  Team (Carl, David) has 2 points.

Dealing 5 cards to everyone.
Dealer is Al.
Top of kitty is Queen of Clubs. Choosing trump.
Carl: Your hand contains:
  Queen of Spades
```

```
  Ten of Diamonds
  Queen of Diamonds
  Queen of Hearts
  King of Diamonds
Carl: Choose an action:
  Order up Queen of Clubs to Al's hand
  Pass
Your choice? Pass
Bob: Your hand contains:
  King of Hearts
  Jack of Hearts
  Jack of Clubs
  King of Clubs
  King of Spades
Bob: Choose an action:
  Order up Queen of Clubs to Al's hand
  Pass
Your choice? Pass
David: Your hand contains:
  Ace of Spades
  Nine of Hearts
  Nine of Diamonds
  Ace of Diamonds
  Jack of Spades
David: Choose an action:
  Order up Queen of Clubs to Al's hand
  Pass
Your choice? Pass
Al: Your hand contains:
  Ten of Hearts
  Ten of Spades
  Ace of Hearts
  Ten of Clubs
  Jack of Diamonds
Al: Choose an action:
  Order up Queen of Clubs to Al's hand
  Pass
Your choice? Pass
No one called up Queen of Clubs. Pick a suit to be trump.
Carl: Your hand contains:
  Queen of Spades
  Ten of Diamonds
  Queen of Diamonds
  Queen of Hearts
  King of Diamonds
Carl: Choose an action:
  Spades
  Hearts
  Diamonds
  Pass
Your choice? Pass
Bob: Your hand contains:
  King of Hearts
  Jack of Hearts
  Jack of Clubs
  King of Clubs
  King of Spades
Bob: Choose an action:
  Spades
  Hearts
  Diamonds
  Pass
Your choice? Diamonds

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Queen of Spades
  Ten of Diamonds
  Queen of Diamonds
  Queen of Hearts
  King of Diamonds
Your choice? Queen of Hearts
```

```
Carl played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Carl
Bob: Choose a card:
  King of Hearts
Your choice? King of Hearts
Bob played King of Hearts.
Cards played so far:
  King of Hearts, played by Bob
  Queen of Hearts, played by Carl
David: Choose a card:
  Nine of Hearts
Your choice? Nine of Hearts
David played Nine of Hearts.
Cards played so far:
  Nine of Hearts, played by David
  King of Hearts, played by Bob
  Queen of Hearts, played by Carl
Al: Choose a card:
  Ten of Hearts
  Ace of Hearts
Your choice? Ace of Hearts
Al played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by Al
  Nine of Hearts, played by David
  King of Hearts, played by Bob
  Queen of Hearts, played by Carl
Al took the trick with Ace of Hearts.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  Ten of Hearts
  Ten of Spades
  Ten of Clubs
  Jack of Diamonds
Your choice? Jack of Diamonds
Al played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Al
Carl: Choose a card:
  Ten of Diamonds
  Queen of Diamonds
  King of Diamonds
Your choice? King of Diamonds
Carl played King of Diamonds.
Cards played so far:
  King of Diamonds, played by Carl
  Jack of Diamonds, played by Al
Bob: Choose a card:
  Jack of Hearts
Your choice? Jack of Hearts
Bob played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by Bob
  King of Diamonds, played by Carl
  Jack of Diamonds, played by Al
David: Choose a card:
  Nine of Diamonds
  Ace of Diamonds
Your choice? Ace of Diamonds
David played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by David
  Jack of Hearts, played by Bob
  King of Diamonds, played by Carl
  Jack of Diamonds, played by Al
Al took the trick with Jack of Diamonds.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
```

```
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  Ten of Hearts
  Ten of Spades
  Ten of Clubs
Your choice? Ten of Clubs
Al played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Al
Carl: Choose a card:
  Queen of Spades
  Ten of Diamonds
  Queen of Diamonds
Your choice? Queen of Diamonds
Carl played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by Carl
  Ten of Clubs, played by Al
Bob: Choose a card:
  Jack of Clubs
  King of Clubs
Your choice? King of Clubs
Bob played King of Clubs.
Cards played so far:
  King of Clubs, played by Bob
  Queen of Diamonds, played by Carl
  Ten of Clubs, played by Al
David: Choose a card:
  Ace of Spades
  Nine of Diamonds
  Jack of Spades
Your choice? Jack of Spades
David played Jack of Spades.
Cards played so far:
  Jack of Spades, played by David
  King of Clubs, played by Bob
  Queen of Diamonds, played by Carl
  Ten of Clubs, played by Al
Carl took the trick with Queen of Diamonds.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Queen of Spades
  Ten of Diamonds
Your choice? Ten of Diamonds
Carl played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by Carl
Bob: Choose a card:
  Jack of Clubs
  King of Spades
Your choice? King of Spades
Bob played King of Spades.
Cards played so far:
  King of Spades, played by Bob
  Ten of Diamonds, played by Carl
David: Choose a card:
  Nine of Diamonds
Your choice? Nine of Diamonds
David played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by David
  King of Spades, played by Bob
  Ten of Diamonds, played by Carl
Al: Choose a card:
  Ten of Hearts
  Ten of Spades
Your choice? Ten of Spades
Al played Ten of Spades.
Cards played so far:
  Ten of Spades, played by Al
```

```
  Nine of Diamonds, played by David
  King of Spades, played by Bob
  Ten of Diamonds, played by Carl
Carl took the trick with Ten of Diamonds.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 2 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Queen of Spades
Your choice? Queen of Spades
Carl played Queen of Spades.
Cards played so far:
  Queen of Spades, played by Carl
Bob: Choose a card:
  Jack of Clubs
Your choice? Jack of Clubs
Bob played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by Bob
  Queen of Spades, played by Carl
David: Choose a card:
  Ace of Spades
Your choice? Ace of Spades
David played Ace of Spades.
Cards played so far:
  Ace of Spades, played by David
  Jack of Clubs, played by Bob
  Queen of Spades, played by Carl
Al: Choose a card:
  Ten of Hearts
Your choice? Ten of Hearts
Al played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by Al
  Ace of Spades, played by David
  Jack of Clubs, played by Bob
  Queen of Spades, played by Carl
David took the trick with Ace of Spades.
Team (Al, Bob) took 2 tricks.
Team (Carl, David) took 3 tricks.
  Team (Carl, David) scores 2 points.
The scores:
  Team (Al, Bob) has 0 points.
  Team (Carl, David) has 4 points.

Dealing 5 cards to everyone.
Dealer is Carl.
Top of kitty is Nine of Spades. Choosing trump.
Bob: Your hand contains:
  Ten of Clubs
  Ace of Hearts
  Ten of Hearts
  Nine of Diamonds
  Jack of Diamonds
Bob: Choose an action:
  Order up Nine of Spades to Carl's hand
  Pass
Your choice? Pass
David: Your hand contains:
  Ten of Diamonds
  King of Clubs
  Ten of Spades
  King of Spades
  Queen of Spades
David: Choose an action:
  Order up Nine of Spades to Carl's hand
  Pass
Your choice? Pass
Al: Your hand contains:
  Ace of Diamonds
  Ace of Clubs
  Jack of Clubs
  Nine of Hearts
```

```
  Ace of Spades
Al: Choose an action:
  Order up Nine of Spades to Carl's hand
  Pass
Your choice? Pass
Carl: Your hand contains:
  Queen of Hearts
  Queen of Diamonds
  King of Diamonds
  Jack of Spades
  Queen of Clubs
Carl: Choose an action:
  Order up Nine of Spades to Carl's hand
  Pass
Your choice? Pass
No one called up Nine of Spades. Pick a suit to be trump.
Bob: Your hand contains:
  Ten of Clubs
  Ace of Hearts
  Ten of Hearts
  Nine of Diamonds
  Jack of Diamonds
Bob: Choose an action:
  Clubs
  Hearts
  Diamonds
  Pass
Your choice? Pass
David: Your hand contains:
  Ten of Diamonds
  King of Clubs
  Ten of Spades
  King of Spades
  Queen of Spades
David: Choose an action:
  Clubs
  Hearts
  Diamonds
  Pass
Your choice? Pass
Al: Your hand contains:
  Ace of Diamonds
  Ace of Clubs
  Jack of Clubs
  Nine of Hearts
  Ace of Spades
Al: Choose an action:
  Clubs
  Hearts
  Diamonds
  Pass
Your choice? Pass
Carl: Your hand contains:
  Queen of Hearts
  Queen of Diamonds
  King of Diamonds
  Jack of Spades
  Queen of Clubs
Carl: Choose an action:
  Clubs
  Hearts
  Diamonds
Your choice? Hearts

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Ten of Clubs
  Ace of Hearts
  Ten of Hearts
  Nine of Diamonds
  Jack of Diamonds
Your choice? Ace of Hearts
```

```
Bob played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by Bob
David: Choose a card:
  Ten of Diamonds
  King of Clubs
  Ten of Spades
  King of Spades
  Queen of Spades
Your choice? Queen of Spades
David played Queen of Spades.
Cards played so far:
  Queen of Spades, played by David
  Ace of Hearts, played by Bob
Al: Choose a card:
  Nine of Hearts
Your choice? Nine of Hearts
Al played Nine of Hearts.
Cards played so far:
  Nine of Hearts, played by Al
  Queen of Spades, played by David
  Ace of Hearts, played by Bob
Carl: Choose a card:
  Queen of Hearts
Your choice? Queen of Hearts
Carl played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Carl
  Nine of Hearts, played by Al
  Queen of Spades, played by David
  Ace of Hearts, played by Bob
Bob took the trick with Ace of Hearts.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Ten of Clubs
  Ten of Hearts
  Nine of Diamonds
  Jack of Diamonds
Your choice? Jack of Diamonds
Bob played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Bob
David: Choose a card:
  Ten of Diamonds
  King of Clubs
  Ten of Spades
  King of Spades
Your choice? King of Spades
David played King of Spades.
Cards played so far:
  King of Spades, played by David
  Jack of Diamonds, played by Bob
Al: Choose a card:
  Ace of Diamonds
  Ace of Clubs
  Jack of Clubs
  Ace of Spades
Your choice? Ace of Spades
Al played Ace of Spades.
Cards played so far:
  Ace of Spades, played by Al
  King of Spades, played by David
  Jack of Diamonds, played by Bob
Carl: Choose a card:
  Queen of Diamonds
  King of Diamonds
  Jack of Spades
  Queen of Clubs
Your choice? Queen of Clubs
Carl played Queen of Clubs.
Cards played so far:
```

```
  Queen of Clubs, played by Carl
  Ace of Spades, played by Al
  King of Spades, played by David
  Jack of Diamonds, played by Bob
Bob took the trick with Jack of Diamonds.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Ten of Clubs
  Ten of Hearts
  Nine of Diamonds
Your choice? Nine of Diamonds
Bob played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by Bob
David: Choose a card:
  Ten of Diamonds
Your choice? Ten of Diamonds
David played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by David
  Nine of Diamonds, played by Bob
Al: Choose a card:
  Ace of Diamonds
Your choice? Ace of Diamonds
Al played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by Al
  Ten of Diamonds, played by David
  Nine of Diamonds, played by Bob
Carl: Choose a card:
  Queen of Diamonds
  King of Diamonds
Your choice? King of Diamonds
Carl played King of Diamonds.
Cards played so far:
  King of Diamonds, played by Carl
  Ace of Diamonds, played by Al
  Ten of Diamonds, played by David
  Nine of Diamonds, played by Bob
Al took the trick with Ace of Diamonds.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 3 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  Ace of Clubs
  Jack of Clubs
Your choice? Jack of Clubs
Al played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by Al
Carl: Choose a card:
  Queen of Diamonds
  Jack of Spades
Your choice? Jack of Spades
Carl played Jack of Spades.
Cards played so far:
  Jack of Spades, played by Carl
  Jack of Clubs, played by Al
Bob: Choose a card:
  Ten of Clubs
Your choice? Ten of Clubs
Bob played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Bob
  Jack of Spades, played by Carl
  Jack of Clubs, played by Al
David: Choose a card:
  King of Clubs
```

```
Your choice? King of Clubs
David played King of Clubs.
Cards played so far:
  King of Clubs, played by David
  Ten of Clubs, played by Bob
  Jack of Spades, played by Carl
  Jack of Clubs, played by Al
David took the trick with King of Clubs.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 3 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
David starts the trick.
David: Choose a card:
  Ten of Spades
Your choice? Ten of Spades
David played Ten of Spades.
Cards played so far:
  Ten of Spades, played by David
Al: Choose a card:
  Ace of Clubs
Your choice? Ace of Clubs
Al played Ace of Clubs.
Cards played so far:
  Ace of Clubs, played by Al
  Ten of Spades, played by David
Carl: Choose a card:
  Queen of Diamonds
Your choice? Queen of Diamonds
Carl played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by Carl
  Ace of Clubs, played by Al
  Ten of Spades, played by David
Bob: Choose a card:
  Ten of Hearts
Your choice? Ten of Hearts
Bob played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by Bob
  Queen of Diamonds, played by Carl
  Ace of Clubs, played by Al
  Ten of Spades, played by David
Bob took the trick with Ten of Hearts.
Team (Al, Bob) took 4 tricks.
  Team (Al, Bob) scores 2 points.
Team (Carl, David) took 1 tricks.
The scores:
  Team (Al, Bob) has 2 points.
  Team (Carl, David) has 4 points.

Dealing 5 cards to everyone.
Dealer is Bob.
Top of kitty is Jack of Spades. Choosing trump.
David: Your hand contains:
  Jack of Hearts
  Ace of Diamonds
  Queen of Spades
  Nine of Spades
  Ace of Hearts
David: Choose an action:
  Order up Jack of Spades to Bob's hand
  Pass
Your choice? Pass
Al: Your hand contains:
  King of Diamonds
  Queen of Diamonds
  Ten of Spades
  Ace of Clubs
  Ten of Clubs
Al: Choose an action:
  Order up Jack of Spades to Bob's hand
  Pass
Your choice? Order up Jack of Spades to Bob's hand
Bob: You acquired Jack of Spades. You must discard something.
```

```
Bob: Choose a card:
  Nine of Diamonds
  Queen of Clubs
  King of Clubs
  Ten of Hearts
  Ace of Spades
Your choice? Ace of Spades
Bob played Ace of Spades.

Trump is Spades.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
David starts the trick.
David: Choose a card:
  Jack of Hearts
  Ace of Diamonds
  Queen of Spades
  Nine of Spades
  Ace of Hearts
Your choice? Nine of Spades
David played Nine of Spades.
Cards played so far:
  Nine of Spades, played by David
Al: Choose a card:
  Ten of Spades
Your choice? Ten of Spades
Al played Ten of Spades.
Cards played so far:
  Ten of Spades, played by Al
  Nine of Spades, played by David
Carl: Choose a card:
  Jack of Clubs
Your choice? Jack of Clubs
Carl played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by Carl
  Ten of Spades, played by Al
  Nine of Spades, played by David
Bob: Choose a card:
  Jack of Spades
Your choice? Jack of Spades
Bob played Jack of Spades.
Cards played so far:
  Jack of Spades, played by Bob
  Jack of Clubs, played by Carl
  Ten of Spades, played by Al
  Nine of Spades, played by David
Bob took the trick with Jack of Spades.

Trump is Spades.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Nine of Diamonds
  Queen of Clubs
  King of Clubs
  Ten of Hearts
Your choice? King of Clubs
Bob played King of Clubs.
Cards played so far:
  King of Clubs, played by Bob
David: Choose a card:
  Jack of Hearts
  Ace of Diamonds
  Queen of Spades
  Ace of Hearts
Your choice? Ace of Hearts
David played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by David
  King of Clubs, played by Bob
Al: Choose a card:
  Ace of Clubs
```

```
  Ten of Clubs
Your choice? Ace of Clubs
Al played Ace of Clubs.
Cards played so far:
  Ace of Clubs, played by Al
  Ace of Hearts, played by David
  King of Clubs, played by Bob
Carl: Choose a card:
  Nine of Clubs
Your choice? Nine of Clubs
Carl played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Carl
  Ace of Clubs, played by Al
  Ace of Hearts, played by David
  King of Clubs, played by Bob
Al took the trick with Ace of Clubs.

Trump is Spades.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  King of Diamonds
  Queen of Diamonds
  Ten of Clubs
Your choice? Ten of Clubs
Al played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Al
Carl: Choose a card:
  Queen of Hearts
  Ten of Diamonds
  Jack of Diamonds
Your choice? Ten of Diamonds
Carl played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by Carl
  Ten of Clubs, played by Al
Bob: Choose a card:
  Queen of Clubs
Your choice? Queen of Clubs
Bob played Queen of Clubs.
Cards played so far:
  Queen of Clubs, played by Bob
  Ten of Diamonds, played by Carl
  Ten of Clubs, played by Al
David: Choose a card:
  Jack of Hearts
  Ace of Diamonds
  Queen of Spades
Your choice? Jack of Hearts
David played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by David
  Queen of Clubs, played by Bob
  Ten of Diamonds, played by Carl
  Ten of Clubs, played by Al
Bob took the trick with Queen of Clubs.

Trump is Spades.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 3 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Nine of Diamonds
  Ten of Hearts
Your choice? Ten of Hearts
Bob played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by Bob
David: Choose a card:
  Ace of Diamonds
  Queen of Spades
```

```
Your choice? Ace of Diamonds
David played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by David
  Ten of Hearts, played by Bob
Al: Choose a card:
  King of Diamonds
  Queen of Diamonds
Your choice? Queen of Diamonds
Al played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by Al
  Ace of Diamonds, played by David
  Ten of Hearts, played by Bob
Carl: Choose a card:
  Queen of Hearts
Your choice? Queen of Hearts
Carl played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Carl
  Queen of Diamonds, played by Al
  Ace of Diamonds, played by David
  Ten of Hearts, played by Bob
Carl took the trick with Queen of Hearts.

Trump is Spades.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 3 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Jack of Diamonds
Your choice? Jack of Diamonds
Carl played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Carl
Bob: Choose a card:
  Nine of Diamonds
Your choice? Nine of Diamonds
Bob played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by Bob
  Jack of Diamonds, played by Carl
David: Choose a card:
  Queen of Spades
Your choice? Queen of Spades
David played Queen of Spades.
Cards played so far:
  Queen of Spades, played by David
  Nine of Diamonds, played by Bob
  Jack of Diamonds, played by Carl
Al: Choose a card:
  King of Diamonds
Your choice? King of Diamonds
Al played King of Diamonds.
Cards played so far:
  King of Diamonds, played by Al
  Queen of Spades, played by David
  Nine of Diamonds, played by Bob
  Jack of Diamonds, played by Carl
David took the trick with Queen of Spades.
Team (Al, Bob) took 3 tricks.
  Team (Al, Bob) scores 1 point.
Team (Carl, David) took 2 tricks.
The scores:
  Team (Al, Bob) has 3 points.
  Team (Carl, David) has 4 points.

Dealing 5 cards to everyone.
Dealer is David.
Top of kitty is Queen of Diamonds. Choosing trump.
Al: Your hand contains:
  King of Spades
  Nine of Clubs
  Queen of Spades
  Jack of Spades
```

```
  Ten of Spades
Al: Choose an action:
  Order up Queen of Diamonds to David's hand
  Pass
Your choice? Order up Queen of Diamonds to David's hand
David: You acquired Queen of Diamonds. You must discard something.
David: Choose a card:
  King of Diamonds
  Ten of Clubs
  Jack of Hearts
  King of Hearts
  Nine of Spades
Your choice? Ten of Clubs
David played Ten of Clubs.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  King of Spades
  Nine of Clubs
  Queen of Spades
  Jack of Spades
  Ten of Spades
Your choice? Queen of Spades
Al played Queen of Spades.
Cards played so far:
  Queen of Spades, played by Al
Carl: Choose a card:
  Ace of Spades
Your choice? Ace of Spades
Carl played Ace of Spades.
Cards played so far:
  Ace of Spades, played by Carl
  Queen of Spades, played by Al
Bob: Choose a card:
  Ten of Diamonds
  King of Clubs
  Ace of Diamonds
  Queen of Clubs
  Jack of Diamonds
Your choice? Jack of Diamonds
Bob played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Bob
  Ace of Spades, played by Carl
  Queen of Spades, played by Al
David: Choose a card:
  Nine of Spades
Your choice? Nine of Spades
David played Nine of Spades.
Cards played so far:
  Nine of Spades, played by David
  Jack of Diamonds, played by Bob
  Ace of Spades, played by Carl
  Queen of Spades, played by Al
Bob took the trick with Jack of Diamonds.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Ten of Diamonds
  King of Clubs
  Ace of Diamonds
  Queen of Clubs
Your choice? Ace of Diamonds
Bob played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by Bob
David: Choose a card:
  Queen of Diamonds
```

```
  King of Diamonds
  Jack of Hearts
Your choice? King of Diamonds
David played King of Diamonds.
Cards played so far:
  King of Diamonds, played by David
  Ace of Diamonds, played by Bob
Al: Choose a card:
  King of Spades
  Nine of Clubs
  Jack of Spades
  Ten of Spades
Your choice? Jack of Spades
Al played Jack of Spades.
Cards played so far:
  Jack of Spades, played by Al
  King of Diamonds, played by David
  Ace of Diamonds, played by Bob
Carl: Choose a card:
  Queen of Hearts
  Ace of Hearts
  Nine of Hearts
  Jack of Clubs
Your choice? Nine of Hearts
Carl played Nine of Hearts.
Cards played so far:
  Nine of Hearts, played by Carl
  Jack of Spades, played by Al
  King of Diamonds, played by David
  Ace of Diamonds, played by Bob
Bob took the trick with Ace of Diamonds.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Ten of Diamonds
  King of Clubs
  Queen of Clubs
Your choice? Queen of Clubs
Bob played Queen of Clubs.
Cards played so far:
  Queen of Clubs, played by Bob
David: Choose a card:
  Queen of Diamonds
  Jack of Hearts
  King of Hearts
Your choice? Jack of Hearts
David played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by David
  Queen of Clubs, played by Bob
Al: Choose a card:
  Nine of Clubs
Your choice? Nine of Clubs
Al played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Al
  Jack of Hearts, played by David
  Queen of Clubs, played by Bob
Carl: Choose a card:
  Jack of Clubs
Your choice? Jack of Clubs
Carl played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by Carl
  Nine of Clubs, played by Al
  Jack of Hearts, played by David
  Queen of Clubs, played by Bob
David took the trick with Jack of Hearts.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
```

```
Team (Carl, David) has taken 1 tricks so far.
David starts the trick.
David: Choose a card:
  Queen of Diamonds
  King of Hearts
Your choice? King of Hearts
David played King of Hearts.
Cards played so far:
  King of Hearts, played by David
Al: Choose a card:
  King of Spades
  Ten of Spades
Your choice? Ten of Spades
Al played Ten of Spades.
Cards played so far:
  Ten of Spades, played by Al
  King of Hearts, played by David
Carl: Choose a card:
  Queen of Hearts
  Ace of Hearts
Your choice? Ace of Hearts
Carl played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by Carl
  Ten of Spades, played by Al
  King of Hearts, played by David
Bob: Choose a card:
  Ten of Diamonds
  King of Clubs
Your choice? King of Clubs
Bob played King of Clubs.
Cards played so far:
  King of Clubs, played by Bob
  Ace of Hearts, played by Carl
  Ten of Spades, played by Al
  King of Hearts, played by David
Carl took the trick with Ace of Hearts.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 2 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Queen of Hearts
Your choice? Queen of Hearts
Carl played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Carl
Bob: Choose a card:
  Ten of Diamonds
Your choice? Ten of Diamonds
Bob played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by Bob
  Queen of Hearts, played by Carl
David: Choose a card:
  Queen of Diamonds
Your choice? Queen of Diamonds
David played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by David
  Ten of Diamonds, played by Bob
  Queen of Hearts, played by Carl
Al: Choose a card:
  King of Spades
Your choice? King of Spades
Al played King of Spades.
Cards played so far:
  King of Spades, played by Al
  Queen of Diamonds, played by David
  Ten of Diamonds, played by Bob
  Queen of Hearts, played by Carl
David took the trick with Queen of Diamonds.
Team (Al, Bob) took 2 tricks.
Team (Carl, David) took 3 tricks.
```

```
  Team (Carl, David) scores 2 points.
The scores:
  Team (Al, Bob) has 3 points.
  Team (Carl, David) has 6 points.

Dealing 5 cards to everyone.
Dealer is Al.
Top of kitty is King of Hearts. Choosing trump.
Carl: Your hand contains:
  Ace of Spades
  Queen of Hearts
  Jack of Clubs
  Ten of Diamonds
  Queen of Clubs
Carl: Choose an action:
  Order up King of Hearts to Al's hand
  Pass
Your choice? Order up King of Hearts to Al's hand
Al: You acquired King of Hearts. You must discard something.
Al: Choose a card:
  Queen of Spades
  Ace of Clubs
  King of Clubs
  Nine of Diamonds
  King of Diamonds
Your choice? King of Diamonds
Al played King of Diamonds.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Ace of Spades
  Queen of Hearts
  Jack of Clubs
  Ten of Diamonds
  Queen of Clubs
Your choice? Ten of Diamonds
Carl played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by Carl
Bob: Choose a card:
  Ace of Diamonds
Your choice? Ace of Diamonds
Bob played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by Bob
  Ten of Diamonds, played by Carl
David: Choose a card:
  Queen of Diamonds
Your choice? Queen of Diamonds
David played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by David
  Ace of Diamonds, played by Bob
  Ten of Diamonds, played by Carl
Al: Choose a card:
  Nine of Diamonds
Your choice? Nine of Diamonds
Al played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by Al
  Queen of Diamonds, played by David
  Ace of Diamonds, played by Bob
  Ten of Diamonds, played by Carl
Bob took the trick with Ace of Diamonds.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Nine of Spades
```

```
  Nine of Clubs
  Nine of Hearts
  Jack of Hearts
Your choice? Nine of Hearts
Bob played Nine of Hearts.
Cards played so far:
  Nine of Hearts, played by Bob
David: Choose a card:
  Ace of Hearts
  Ten of Hearts
Your choice? Ace of Hearts
David played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by David
  Nine of Hearts, played by Bob
Al: Choose a card:
  King of Hearts
Your choice? King of Hearts
Al played King of Hearts.
Cards played so far:
  King of Hearts, played by Al
  Ace of Hearts, played by David
  Nine of Hearts, played by Bob
Carl: Choose a card:
  Queen of Hearts
Your choice? Queen of Hearts
Carl played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Carl
  King of Hearts, played by Al
  Ace of Hearts, played by David
  Nine of Hearts, played by Bob
David took the trick with Ace of Hearts.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
David starts the trick.
David: Choose a card:
  King of Spades
  Ten of Hearts
  Jack of Spades
Your choice? King of Spades
David played King of Spades.
Cards played so far:
  King of Spades, played by David
Al: Choose a card:
  Queen of Spades
Your choice? Queen of Spades
Al played Queen of Spades.
Cards played so far:
  Queen of Spades, played by Al
  King of Spades, played by David
Carl: Choose a card:
  Ace of Spades
Your choice? Ace of Spades
Carl played Ace of Spades.
Cards played so far:
  Ace of Spades, played by Carl
  Queen of Spades, played by Al
  King of Spades, played by David
Bob: Choose a card:
  Nine of Spades
Your choice? Nine of Spades
Bob played Nine of Spades.
Cards played so far:
  Nine of Spades, played by Bob
  Ace of Spades, played by Carl
  Queen of Spades, played by Al
  King of Spades, played by David
Carl took the trick with Ace of Spades.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 1 tricks so far.
```

```
Team (Carl, David) has taken 2 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Jack of Clubs
  Queen of Clubs
Your choice? Queen of Clubs
Carl played Queen of Clubs.
Cards played so far:
  Queen of Clubs, played by Carl
Bob: Choose a card:
  Nine of Clubs
Your choice? Nine of Clubs
Bob played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Bob
  Queen of Clubs, played by Carl
David: Choose a card:
  Ten of Hearts
  Jack of Spades
Your choice? Ten of Hearts
David played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by David
  Nine of Clubs, played by Bob
  Queen of Clubs, played by Carl
Al: Choose a card:
  Ace of Clubs
  King of Clubs
Your choice? King of Clubs
Al played King of Clubs.
Cards played so far:
  King of Clubs, played by Al
  Ten of Hearts, played by David
  Nine of Clubs, played by Bob
  Queen of Clubs, played by Carl
David took the trick with Ten of Hearts.

Trump is Hearts.
Maker is Team (Carl, David).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 3 tricks so far.
David starts the trick.
David: Choose a card:
  Jack of Spades
Your choice? Jack of Spades
David played Jack of Spades.
Cards played so far:
  Jack of Spades, played by David
Al: Choose a card:
  Ace of Clubs
Your choice? Ace of Clubs
Al played Ace of Clubs.
Cards played so far:
  Ace of Clubs, played by Al
  Jack of Spades, played by David
Carl: Choose a card:
  Jack of Clubs
Your choice? Jack of Clubs
Carl played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by Carl
  Ace of Clubs, played by Al
  Jack of Spades, played by David
Bob: Choose a card:
  Jack of Hearts
Your choice? Jack of Hearts
Bob played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by Bob
  Jack of Clubs, played by Carl
  Ace of Clubs, played by Al
  Jack of Spades, played by David
Bob took the trick with Jack of Hearts.
Team (Al, Bob) took 2 tricks.
Team (Carl, David) took 3 tricks.
  Team (Carl, David) scores 1 point.
```

```
The scores:
  Team (Al, Bob) has 3 points.
  Team (Carl, David) has 7 points.

Dealing 5 cards to everyone.
Dealer is Carl.
Top of kitty is Ace of Hearts. Choosing trump.
Bob: Your hand contains:
  Jack of Diamonds
  Ace of Spades
  Jack of Hearts
  Queen of Clubs
  Ten of Clubs
Bob: Choose an action:
  Order up Ace of Hearts to Carl's hand
  Pass
Your choice? Order up Ace of Hearts to Carl's hand
Carl: You acquired Ace of Hearts. You must discard something.
Carl: Choose a card:
  Ace of Clubs
  Ace of Diamonds
  King of Clubs
  Ten of Spades
  Nine of Clubs
Your choice? Ten of Spades
Carl played Ten of Spades.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Jack of Diamonds
  Ace of Spades
  Jack of Hearts
  Queen of Clubs
  Ten of Clubs
Your choice? Queen of Clubs
Bob played Queen of Clubs.
Cards played so far:
  Queen of Clubs, played by Bob
David: Choose a card:
  Jack of Clubs
Your choice? Jack of Clubs
David played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by David
  Queen of Clubs, played by Bob
Al: Choose a card:
  Ten of Diamonds
  King of Spades
  King of Diamonds
  Queen of Spades
  Queen of Hearts
Your choice? Queen of Spades
Al played Queen of Spades.
Cards played so far:
  Queen of Spades, played by Al
  Jack of Clubs, played by David
  Queen of Clubs, played by Bob
Carl: Choose a card:
  Ace of Clubs
  King of Clubs
  Nine of Clubs
Your choice? King of Clubs
Carl played King of Clubs.
Cards played so far:
  King of Clubs, played by Carl
  Queen of Spades, played by Al
  Jack of Clubs, played by David
  Queen of Clubs, played by Bob
Carl took the trick with King of Clubs.

Trump is Hearts.
Maker is Team (Al, Bob).
```

```
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Ace of Hearts
  Ace of Clubs
  Ace of Diamonds
  Nine of Clubs
Your choice? Nine of Clubs
Carl played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Carl
Bob: Choose a card:
  Ten of Clubs
Your choice? Ten of Clubs
Bob played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Bob
  Nine of Clubs, played by Carl
David: Choose a card:
  Jack of Spades
  Nine of Diamonds
  Nine of Hearts
  Ten of Hearts
Your choice? Ten of Hearts
David played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by David
  Ten of Clubs, played by Bob
  Nine of Clubs, played by Carl
Al: Choose a card:
  Ten of Diamonds
  King of Spades
  King of Diamonds
  Queen of Hearts
Your choice? Queen of Hearts
Al played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Al
  Ten of Hearts, played by David
  Ten of Clubs, played by Bob
  Nine of Clubs, played by Carl
Al took the trick with Queen of Hearts.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Al starts the trick.
Al: Choose a card:
  Ten of Diamonds
  King of Spades
  King of Diamonds
Your choice? King of Diamonds
Al played King of Diamonds.
Cards played so far:
  King of Diamonds, played by Al
Carl: Choose a card:
  Ace of Diamonds
Your choice? Ace of Diamonds
Carl played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by Carl
  King of Diamonds, played by Al
Bob: Choose a card:
  Jack of Diamonds
  Ace of Spades
  Jack of Hearts
Your choice? Jack of Hearts
Bob played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by Bob
  Ace of Diamonds, played by Carl
  King of Diamonds, played by Al
David: Choose a card:
  Nine of Diamonds
```

```
Your choice? Nine of Diamonds
David played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by David
  Jack of Hearts, played by Bob
  Ace of Diamonds, played by Carl
  King of Diamonds, played by Al
Bob took the trick with Jack of Hearts.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 2 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Jack of Diamonds
  Ace of Spades
Your choice? Jack of Diamonds
Bob played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Bob
David: Choose a card:
  Nine of Hearts
Your choice? Nine of Hearts
David played Nine of Hearts.
Cards played so far:
  Nine of Hearts, played by David
  Jack of Diamonds, played by Bob
Al: Choose a card:
  Ten of Diamonds
  King of Spades
Your choice? King of Spades
Al played King of Spades.
Cards played so far:
  King of Spades, played by Al
  Nine of Hearts, played by David
  Jack of Diamonds, played by Bob
Carl: Choose a card:
  Ace of Hearts
Your choice? Ace of Hearts
Carl played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by Carl
  King of Spades, played by Al
  Nine of Hearts, played by David
  Jack of Diamonds, played by Bob
Bob took the trick with Jack of Diamonds.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 3 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Bob starts the trick.
Bob: Choose a card:
  Ace of Spades
Your choice? Ace of Spades
Bob played Ace of Spades.
Cards played so far:
  Ace of Spades, played by Bob
David: Choose a card:
  Jack of Spades
Your choice? Jack of Spades
David played Jack of Spades.
Cards played so far:
  Jack of Spades, played by David
  Ace of Spades, played by Bob
Al: Choose a card:
  Ten of Diamonds
Your choice? Ten of Diamonds
Al played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by Al
  Jack of Spades, played by David
  Ace of Spades, played by Bob
Carl: Choose a card:
  Ace of Clubs
```

```
Your choice? Ace of Clubs
Carl played Ace of Clubs.
Cards played so far:
  Ace of Clubs, played by Carl
  Ten of Diamonds, played by Al
  Jack of Spades, played by David
  Ace of Spades, played by Bob
Bob took the trick with Ace of Spades.
Team (Al, Bob) took 4 tricks.
  Team (Al, Bob) scores 1 point.
Team (Carl, David) took 1 tricks.
The scores:
  Team (Al, Bob) has 4 points.
  Team (Carl, David) has 7 points.

Dealing 5 cards to everyone.
Dealer is Bob.
Top of kitty is Jack of Hearts. Choosing trump.
David: Your hand contains:
  Ace of Spades
  Queen of Spades
  Queen of Clubs
  King of Diamonds
  Ten of Diamonds
David: Choose an action:
  Order up Jack of Hearts to Bob's hand
  Pass
Your choice? Pass
Al: Your hand contains:
  Jack of Diamonds
  King of Clubs
  Ten of Clubs
  Nine of Diamonds
  Ace of Hearts
Al: Choose an action:
  Order up Jack of Hearts to Bob's hand
  Pass
Your choice? Order up Jack of Hearts to Bob's hand
Bob: You acquired Jack of Hearts. You must discard something.
Bob: Choose a card:
  Ace of Diamonds
  Nine of Hearts
  King of Hearts
  King of Spades
  Ten of Spades
Your choice? Nine of Hearts
Bob played Nine of Hearts.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
David starts the trick.
David: Choose a card:
  Ace of Spades
  Queen of Spades
  Queen of Clubs
  King of Diamonds
  Ten of Diamonds
Your choice? Ten of Diamonds
David played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by David
Al: Choose a card:
  Nine of Diamonds
Your choice? Nine of Diamonds
Al played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by Al
  Ten of Diamonds, played by David
Carl: Choose a card:
  Ace of Clubs
  Nine of Clubs
  Nine of Spades
  Ten of Hearts
  Queen of Hearts
```

```
Your choice? Ten of Hearts
Carl played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by Carl
  Nine of Diamonds, played by Al
  Ten of Diamonds, played by David
Bob: Choose a card:
  Ace of Diamonds
Your choice? Ace of Diamonds
Bob played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by Bob
  Ten of Hearts, played by Carl
  Nine of Diamonds, played by Al
  Ten of Diamonds, played by David
Carl took the trick with Ten of Hearts.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Ace of Clubs
  Nine of Clubs
  Nine of Spades
  Queen of Hearts
Your choice? Ace of Clubs
Carl played Ace of Clubs.
Cards played so far:
  Ace of Clubs, played by Carl
Bob: Choose a card:
  Jack of Hearts
  King of Hearts
  King of Spades
  Ten of Spades
Your choice? King of Spades
Bob played King of Spades.
Cards played so far:
  King of Spades, played by Bob
  Ace of Clubs, played by Carl
David: Choose a card:
  Queen of Clubs
Your choice? Queen of Clubs
David played Queen of Clubs.
Cards played so far:
  Queen of Clubs, played by David
  King of Spades, played by Bob
  Ace of Clubs, played by Carl
Al: Choose a card:
  King of Clubs
  Ten of Clubs
Your choice? King of Clubs
Al played King of Clubs.
Cards played so far:
  King of Clubs, played by Al
  Queen of Clubs, played by David
  King of Spades, played by Bob
  Ace of Clubs, played by Carl
Carl took the trick with Ace of Clubs.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 2 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Nine of Clubs
  Nine of Spades
  Queen of Hearts
Your choice? Nine of Spades
Carl played Nine of Spades.
Cards played so far:
  Nine of Spades, played by Carl
Bob: Choose a card:
  Ten of Spades
```

```
Your choice? Ten of Spades
Bob played Ten of Spades.
Cards played so far:
  Ten of Spades, played by Bob
  Nine of Spades, played by Carl
David: Choose a card:
  Ace of Spades
  Queen of Spades
Your choice? Ace of Spades
David played Ace of Spades.
Cards played so far:
  Ace of Spades, played by David
  Ten of Spades, played by Bob
  Nine of Spades, played by Carl
Al: Choose a card:
  Jack of Diamonds
  Ten of Clubs
  Ace of Hearts
Your choice? Ten of Clubs
Al played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Al
  Ace of Spades, played by David
  Ten of Spades, played by Bob
  Nine of Spades, played by Carl
David took the trick with Ace of Spades.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 3 tricks so far.
David starts the trick.
David: Choose a card:
  Queen of Spades
  King of Diamonds
Your choice? King of Diamonds
David played King of Diamonds.
Cards played so far:
  King of Diamonds, played by David
Al: Choose a card:
  Jack of Diamonds
  Ace of Hearts
Your choice? Jack of Diamonds
Al played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by Al
  King of Diamonds, played by David
Carl: Choose a card:
  Nine of Clubs
  Queen of Hearts
Your choice? Nine of Clubs
Carl played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Carl
  Jack of Diamonds, played by Al
  King of Diamonds, played by David
Bob: Choose a card:
  Jack of Hearts
  King of Hearts
Your choice? King of Hearts
Bob played King of Hearts.
Cards played so far:
  King of Hearts, played by Bob
  Nine of Clubs, played by Carl
  Jack of Diamonds, played by Al
  King of Diamonds, played by David
Al took the trick with Jack of Diamonds.

Trump is Hearts.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 1 tricks so far.
Team (Carl, David) has taken 3 tricks so far.
Al starts the trick.
Al: Choose a card:
  Ace of Hearts
Your choice? Ace of Hearts
```

```
Al played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by Al
Carl: Choose a card:
  Queen of Hearts
Your choice? Queen of Hearts
Carl played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by Carl
  Ace of Hearts, played by Al
Bob: Choose a card:
  Jack of Hearts
Your choice? Jack of Hearts
Bob played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by Bob
  Queen of Hearts, played by Carl
  Ace of Hearts, played by Al
David: Choose a card:
  Queen of Spades
Your choice? Queen of Spades
David played Queen of Spades.
Cards played so far:
  Queen of Spades, played by David
  Jack of Hearts, played by Bob
  Queen of Hearts, played by Carl
  Ace of Hearts, played by Al
Bob took the trick with Jack of Hearts.
Team (Al, Bob) took 2 tricks.
Team (Carl, David) took 3 tricks.
  Team (Carl, David) scores 2 points.
The scores:
  Team (Al, Bob) has 4 points.
  Team (Carl, David) has 9 points.

Dealing 5 cards to everyone.
Dealer is David.
Top of kitty is Ace of Diamonds. Choosing trump.
Al: Your hand contains:
  Ten of Diamonds
  Ace of Spades
  Jack of Clubs
  King of Diamonds
  King of Hearts
Al: Choose an action:
  Order up Ace of Diamonds to David's hand
  Pass
Your choice? Order up Ace of Diamonds to David's hand
David: You acquired Ace of Diamonds. You must discard something.
David: Choose a card:
  Queen of Hearts
  Ace of Hearts
  Nine of Diamonds
  Jack of Diamonds
  Nine of Spades
Your choice? Nine of Spades
David played Nine of Spades.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 0 tricks so far.
Al starts the trick.
Al: Choose a card:
  Ten of Diamonds
  Ace of Spades
  Jack of Clubs
  King of Diamonds
  King of Hearts
Your choice? Ten of Diamonds
Al played Ten of Diamonds.
Cards played so far:
  Ten of Diamonds, played by Al
Carl: Choose a card:
  Jack of Hearts
Your choice? Jack of Hearts
```

```
Carl played Jack of Hearts.
Cards played so far:
  Jack of Hearts, played by Carl
  Ten of Diamonds, played by Al
Bob: Choose a card:
  Queen of Diamonds
Your choice? Queen of Diamonds
Bob played Queen of Diamonds.
Cards played so far:
  Queen of Diamonds, played by Bob
  Jack of Hearts, played by Carl
  Ten of Diamonds, played by Al
David: Choose a card:
  Ace of Diamonds
  Nine of Diamonds
  Jack of Diamonds
Your choice? Nine of Diamonds
David played Nine of Diamonds.
Cards played so far:
  Nine of Diamonds, played by David
  Queen of Diamonds, played by Bob
  Jack of Hearts, played by Carl
  Ten of Diamonds, played by Al
Carl took the trick with Jack of Hearts.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 1 tricks so far.
Carl starts the trick.
Carl: Choose a card:
  Nine of Clubs
  Nine of Hearts
  Ten of Hearts
  Ten of Clubs
Your choice? Ten of Hearts
Carl played Ten of Hearts.
Cards played so far:
  Ten of Hearts, played by Carl
Bob: Choose a card:
  Jack of Spades
  King of Spades
  Queen of Clubs
  King of Clubs
Your choice? Jack of Spades
Bob played Jack of Spades.
Cards played so far:
  Jack of Spades, played by Bob
  Ten of Hearts, played by Carl
David: Choose a card:
  Queen of Hearts
  Ace of Hearts
Your choice? Ace of Hearts
David played Ace of Hearts.
Cards played so far:
  Ace of Hearts, played by David
  Jack of Spades, played by Bob
  Ten of Hearts, played by Carl
Al: Choose a card:
  King of Hearts
Your choice? King of Hearts
Al played King of Hearts.
Cards played so far:
  King of Hearts, played by Al
  Ace of Hearts, played by David
  Jack of Spades, played by Bob
  Ten of Hearts, played by Carl
David took the trick with Ace of Hearts.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 2 tricks so far.
David starts the trick.
David: Choose a card:
  Ace of Diamonds
```

```
  Queen of Hearts
  Jack of Diamonds
Your choice? Jack of Diamonds
David played Jack of Diamonds.
Cards played so far:
  Jack of Diamonds, played by David
Al: Choose a card:
  King of Diamonds
Your choice? King of Diamonds
Al played King of Diamonds.
Cards played so far:
  King of Diamonds, played by Al
  Jack of Diamonds, played by David
Carl: Choose a card:
  Nine of Clubs
  Nine of Hearts
  Ten of Clubs
Your choice? Ten of Clubs
Carl played Ten of Clubs.
Cards played so far:
  Ten of Clubs, played by Carl
  King of Diamonds, played by Al
  Jack of Diamonds, played by David
Bob: Choose a card:
  King of Spades
  Queen of Clubs
  King of Clubs
Your choice? King of Clubs
Bob played King of Clubs.
Cards played so far:
  King of Clubs, played by Bob
  Ten of Clubs, played by Carl
  King of Diamonds, played by Al
  Jack of Diamonds, played by David
David took the trick with Jack of Diamonds.

Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 3 tricks so far.
David starts the trick.
David: Choose a card:
  Ace of Diamonds
  Queen of Hearts
Your choice? Queen of Hearts
David played Queen of Hearts.
Cards played so far:
  Queen of Hearts, played by David
Al: Choose a card:
  Ace of Spades
  Jack of Clubs
Your choice? Jack of Clubs
Al played Jack of Clubs.
Cards played so far:
  Jack of Clubs, played by Al
  Queen of Hearts, played by David
Carl: Choose a card:
  Nine of Hearts
Your choice? Nine of Hearts
Carl played Nine of Hearts.
Cards played so far:
  Nine of Hearts, played by Carl
  Jack of Clubs, played by Al
  Queen of Hearts, played by David
Bob: Choose a card:
  King of Spades
  Queen of Clubs
Your choice? Queen of Clubs
Bob played Queen of Clubs.
Cards played so far:
  Queen of Clubs, played by Bob
  Nine of Hearts, played by Carl
  Jack of Clubs, played by Al
  Queen of Hearts, played by David
David took the trick with Queen of Hearts.
```

```
Trump is Diamonds.
Maker is Team (Al, Bob).
Team (Al, Bob) has taken 0 tricks so far.
Team (Carl, David) has taken 4 tricks so far.
David starts the trick.
David: Choose a card:
  Ace of Diamonds
Your choice? Ace of Diamonds
David played Ace of Diamonds.
Cards played so far:
  Ace of Diamonds, played by David
Al: Choose a card:
  Ace of Spades
Your choice? Ace of Spades
Al played Ace of Spades.
Cards played so far:
  Ace of Spades, played by Al
  Ace of Diamonds, played by David
Carl: Choose a card:
  Nine of Clubs
Your choice? Nine of Clubs
Carl played Nine of Clubs.
Cards played so far:
  Nine of Clubs, played by Carl
  Ace of Spades, played by Al
  Ace of Diamonds, played by David
Bob: Choose a card:
  King of Spades
Your choice? King of Spades
Bob played King of Spades.
Cards played so far:
  King of Spades, played by Bob
  Nine of Clubs, played by Carl
  Ace of Spades, played by Al
  Ace of Diamonds, played by David
David took the trick with Ace of Diamonds.
Team (Al, Bob) took 0 tricks.
Team (Carl, David) took 5 tricks.
  Team (Carl, David) scores 4 points.
The scores:
  Team (Al, Bob) has 4 points.
  Team (Carl, David) has 13 points.
The game was won by Team (Carl, David)
```

Chapter 8

Lessons Learned

8.1 Most Important Lesson

The most important lesson I learned was that the devil is in the details.

When I initially decided on the scope of this card game language interpreter, things seemed fairly simple in my head. I understood the C language at a “Lawyer” level, having become very familiar with the C language standard over the years, and I was also familiar with lexing and parsing before this class.

However, the semantic analysis phase was much larger in scope than I imagined. The static type checking, skip and label resolution, identifier resolution, and the structure of the Checked AST were all more complicated than I thought. Once I finally finished the analyzer it all seemed so simple, but working on it for the first time definitely opened my eyes to something new.

8.2 Advice for Future Students

I have a few pieces of advice for future students.

- Write the lexer and parser (without the AST) before deciding on the formal syntax. Make sure you can parse a few substantial example programs without parse errors or ambiguities.
- Write test cases as you write the reference manual. Every time you describe something as an error in the manual, write a test to make sure good programs don’t error, and equally important, bad programs don’t pass without error. It’s easy to forget the corner cases if you write the tests later.
- Don’t bite off more than you can chew. It’s easy to get lost in details. Keep features simple and orthogonal.

Chapter 9

Appendix

This is a full code listing of all source files, build files, test files, and support scripts used for this project.

9.1 Interpreter Source

9.1.1 Driver

Listing 9.1: pip.ml

```
(**************)
(* The Driver *)

let usage_msg = "Usage: pip [ --print-ast | --analyze ] input_file"

(* Command-line options and setters for Arg.parse *)
let print_ast = ref false
let analyze = ref false
let input_file = ref ""

let set_print_ast () = print_ast := true
let set_analyze ()   = analyze := true
let set_input_file f = input_file := f

(* Main *)
let main () =
  let desc =
    [("--print-ast", Arg.Unit(set_print_ast), "Print the AST and exit.");
     ("--analyze",   Arg.Unit(set_analyze),   "Semantic analysis and exit.")]
  in
  Arg.parse desc set_input_file usage_msg;

  if !input_file = "" then (print_endline usage_msg; exit 1);

  (* Syntax *)
  let input = open_in !input_file in
  let lexbuf = Lexing.from_channel input in
  let ast = Parser.game Scanner.token lexbuf in
  close_in input;
  if !print_ast then (print_string (Ast.str_game ast); exit 0);

  (* Semantics *)
  let cast = Semantic.sem_game ast in
  if !analyze then exit 0;

  (* Interpretation *)
  Interp.run_game cast;
  exit 0

(* Run main *)
let _ = try main () with Failure(s) -> (print_endline s; exit 1)
```

9.1.2 Lexer

Listing 9.2: scanner.mll

```
{
  open Parser

  (* Increment position based on seeing a newline *)
  let newline lexbuf =
    let pos = lexbuf.Lexing.lex_curr_p in
    lexbuf.Lexing.lex_curr_p <- {
      pos with Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
               Lexing.pos_bol = pos.Lexing.pos_cnum;
    };
    Utils.line_num := lexbuf.Lexing.lex_curr_p.Lexing.pos_lnum

  (* Get the position from the lexbuf. Back up n chars since they
     were already scanned. *)
  let pos lexbuf n = let p = lexbuf.Lexing.lex_curr_p in
                     let (l, c) = Utils.lexpos p in
                     (l, c - n)
}

let identifier = ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*

rule token = parse
  eof { EOF }

(* Grouping *)

| "{" { LCURLY }
| "}" { RCURLY }
| "(" { LPAREN }
| ")" { RPAREN }
| "[" { LSQUARE }
| "]" { RSQUARE }

(* Operators and punctuation *)
(* Note: and, or, in, not in, defined, not defined, canplay are below in keywords *)

| "->" { ARROW }
| ","  { COMMA }
| "."  { DOT }
| ".." { DOTDOT }
| ";"  { SEMI }
| "~"  { TILDE }
| "%"  { PERCENT }

| "="  { ASSIGN }

| "=="  { EQ }
| "!="  { NOTEQ }
| "<"   { LT }
| "<="  { LTEQ }
| ">"   { GT }
| ">="  { GTEQ }

| "+" { PLUS }
| "-" { MINUS }
| "*" { TIMES }
| "/" { DIVIDE }

| "+=" { PLUSEQ }
| "-=" { MINUSEQ }
| "*=" { TIMESEQ }
| "/=" { DIVIDEEQ }

(* Types *)

| "Action"      { ACTION }
| "Area"        { AREA }
| "Boolean"     { BOOLEAN }
| "Card"        { CARD }
| "CardList"    { CARDLIST }
| "Deck"        { DECK }
| "Game"        { GAME }
```

```
| "Number"     { NUMBER }
| "Ordering"   { ORDERING }
| "Player"     { PLAYER }
| "PlayerList" { PLAYERLIST }
| "Rank"       { RANK }
| "RankList"   { RANKLIST }
| "Rule"       { RULE }
| "String"     { STRING }
| "Suit"       { SUIT }
| "SuitList"   { SUITLIST }
| "Team"       { TEAM }
| "TeamList"   { TEAMLIST }

(* Keywords *)

| "all"       { ALL }
| "and"       { AND }
| "ask"       { ASK }
| "at"        { AT }
| "be"        { BE }
| "by"        { BY }
| "canplay"   { CANPLAY }
| "deal"      { DEAL }
| "defined"   { DEFINED }
| "else"      { ELSE }
| "elseif"    { ELSEIF }
| "facedown"  { FACEDOWN }
| "faceup"    { FACEUP }
| "for"       { FOR }
| "forever"   { FOREVER }
| "from"      { FROM }
| "if"        { IF }
| "in"        { IN }
| "is"        { IS }
| "label"     { LABEL }
| "labeled"   { LABELED }
| "leaving"   { LEAVING }
| "let"       { LET }
| "message"   { MESSAGE }
| "not"       { NOT }
| "of"        { OF }
| "or"        { OR }
| "order"     { ORDER }
| "play"      { PLAY }
| "players"   { PLAYERS }
| "requires"  { REQUIRES }
| "rotate"    { ROTATE }
| "shuffle"   { SHUFFLE }
| "skip"      { SKIP }
| "spreadout" { SPREADOUT }
| "squaredup" { SQUAREDUP }
| "standard"  { STANDARD }
| "starting"  { STARTING }
| "teams"     { TEAMS }
| "to"        { TO }
| "winner"    { WINNER }

(* Literals *)

| 'A' { ACE }
| 'K' { KING }
| 'Q' { QUEEN }
| 'J' { JACK }

| 'C' { CLUBS }
| 'H' { HEARTS }
| 'S' { SPADES }
| 'D' { DIAMONDS }

| "True"            { TRUE }
| "False"           { FALSE }
| ['0'-'9']+ as num { NUMLIT(int_of_string num) }
| '"'               { STRINGLIT(str_lit [] [] lexbuf) }

(* Identifiers *)
```

```
| identifier as lit { ID(lit) }

(* Whitespace *)

| '\n'          { newline lexbuf ; token lexbuf } (* Newline *)
| [' ' '\t']    { token lexbuf }                  (* Whitespace *)
| "#" [^ '\n']* { token lexbuf }                  (* Line Comments *)

(* Anything else is an error *)

| _ as c { let ce = Char.escaped c in
           Utils.pos_error (pos lexbuf 1) ("Illegal character: " ^ ce) }


(*
 * String literals can contain nested references to identifiers, wrapped
 * in { and }. This rule returns an Ast.str_lit.
 *
 * For this sequence of characters:
 *
 *   "ab{c}{d}"
 *
 * this returns
 *
 *   [StrLit("ab"); StrRef(Id("c")); StrLit(""); StrRef(Id("d")); StrLit("")]
 *)

and str_lit cs ls = parse
  (* End of string *)
  '"'              { let s = Utils.str_implode (List.rev cs)
                     in List.rev (Ast.StrLit(s) :: ls) }

  (* Escapes we convert *)
| '\\' 'n'         { str_lit ('\n' :: cs) ls lexbuf }
| '\\' 't'         { str_lit ('\t' :: cs) ls lexbuf }

  (* Escapes that pass through *)
| '\\' ('\n' as c) { newline lexbuf ; str_lit (c :: cs) ls lexbuf }
| '\\' (_ as c)    { str_lit (c :: cs) ls lexbuf }

  (* Identifier start *)
| '{'              { let s = Utils.str_implode (List.rev cs)
                     in str_id [] (Ast.StrLit(s) :: ls) lexbuf }

  (* Regular characters *)
| '\n' as c        { newline lexbuf ; str_lit (c :: cs) ls lexbuf }
| _ as c           { str_lit (c :: cs) ls lexbuf }

and str_id cs ls = parse
  (* End of identifier *)
  '}'                { str_lit cs ls lexbuf }

| (* Identifier characters *)
  identifier as lit { let p = pos lexbuf (String.length lit) in
                      str_id cs (Ast.StrRef(Ast.Id(lit, p)) :: ls) lexbuf }

  (* Error *)
| _ as c { let ce = Char.escaped c in
           Utils.pos_error (pos lexbuf 1)
             ("Nested string identifier expected; found: " ^ ce) }
```

9.1.3 Parser

Listing 9.3: parser.mly

```
%{
  (* Called in rules - return the start of the given item *)
  let pos n = let pos = rhs_start_pos n in
              Utils.lexpos pos

  (* Called during parse errors - show the line number *)
  let parse_error m =
    print_endline ("Error: " ^ m ^ " on line "
```

```
                    ^ (string_of_int !Utils.line_num))

%}

%token EOF

/* Operators and punctuation */

%token RCURLY LCURLY RPAREN LPAREN RSQUARE LSQUARE
%token ARROW DOT DOTDOT SEMI TILDE PERCENT
%token ASSIGN COMMA DIVIDE DIVIDEEQ EQ
%token GT GTEQ LT LTEQ MINUS MINUSEQ NOTEQ
%token PLUS PLUSEQ TIMES TIMESEQ

/* Types */

%token BOOLEAN CARD CARDLIST DECK NUMBER
%token PLAYER PLAYERLIST RANK RANKLIST STRING
%token SUIT SUITLIST TEAM TEAMLIST

%token ACTION AREA GAME ORDERING RULE

/* Keywords */

%token ALL ASK AT BE BY CANPLAY DEAL
%token ELSE ELSEIF FACEDOWN FACEUP FOR FOREVER FROM
%token IF IS LABEL LABELED LEAVING LET MESSAGE
%token OF ORDER PLAY REQUIRES ROTATE
%token SHUFFLE SKIP SPREADOUT SQUAREDUP
%token STARTING SUIT TO WINNER

%token AND OR NOT IN DEFINED
%token PLAYERS STANDARD TEAMS

/* Literals */

%token ACE KING QUEEN JACK
%token CLUBS HEARTS SPADES DIAMONDS
%token TRUE FALSE
%token <int> NUMLIT
%token <Ast.str_lit> STRINGLIT
%token <Ast.id> ID

/* Precedence and associativity */

%left OR
%left AND
%nonassoc NOT
%nonassoc EQ NOTEQ LT LTEQ GT GTEQ IN
%left PLUS MINUS
%left TIMES DIVIDE
%left COMMA
%nonassoc DOTDOT
%left ARROW

/* Start symbol and type */

%start game
%type <Ast.game> game

%%

/* game */
game: gameheading decls EOF
        { (fst $1, snd $1, List.rev $2, pos 1) }

/* (string, ptcount) */
gameheading: GAME STRINGLIT REQUIRES count DOT  { ($2, $4) }

/* pcount */
count: playernum PLAYERS           { Ast.PlayerCount($1, pos 1) }
     | playernum TEAMS OF NUMLIT  { Ast.TeamCount($1, $4, pos 1) }

/* int list */
playernum: NUMLIT TO NUMLIT  { Utils.int_range $1 $3 }
         | playernumlist     { List.rev $1 }
```

```
/* int list reversed */
playernumlist: NUMLIT                    { [$1] }
             | playernumlist OR NUMLIT  { $3 :: $1 }


/* Declarations */

/* decl list reversed */
decls: /* Empty */  { [] }
     | decls decl   { $2 :: $1 }

/* decl */
decl: vardecl     { $1 }
    | areadecl    { $1 }
    | actiondecl  { $1 }
    | ruledecl    { $1 }
    | orderdecl   { $1 }

/* decl */
vardecl: dtype ID DOT              { Ast.Var($1, $2, Ast.EmptyExpr, pos 1) }
       | dtype ID ASSIGN expr DOT  { Ast.Var($1, $2, $4, pos 1) }

/* decl */
areadecl: AREA ID LABELED STRINGLIT DOT
            { Ast.Area($2, $4, [], pos 1) }

        | AREA ID LABELED STRINGLIT IS areaopts DOT
            { Ast.Area($2, $4, List.rev $6, pos 1) }

/* areaopts reversed */
areaopts: areaopt                  { [$1] }
        | areaopts COMMA areaopt  { $3 :: $1 }

/* areaopt */
areaopt: FACEDOWN   { (Ast.Facedown,  pos 1) }
       | FACEUP     { (Ast.Faceup,    pos 1) }
       | SQUAREDUP  { (Ast.Squaredup, pos 1) }
       | SPREADOUT  { (Ast.Spreadout, pos 1) }

/* decl */
actiondecl: ACTION ID block  { Ast.Action($2, $3, pos 1) }

/* decl */
ruledecl: RULE ID LPAREN ID COMMA ID COMMA ID RPAREN ASSIGN expr DOT
            { Ast.Rule($2, [$4; $6; $8], $11, pos 1) }

/* decl */
orderdecl: ORDERING ID LPAREN ID RPAREN ASSIGN expr DOT
            { Ast.Ordering($2, $4, $7, pos 1) }


/* Statements */

/* block */
block: LCURLY stmts RCURLY  { List.rev $2 }

/* stmt list reversed */
stmts: /* Empty */ { [] }
     | stmts stmt  { $2 :: $1 }

/* stmt */
stmt: askstmt       { $1 }
    | assignstmt    { $1 }
    | compoundstmt  { $1 }
    | dealstmt      { $1 }
    | foreverstmt   { $1 }
    | forstmt       { $1 }
    | ifstmt        { $1 }
    | invokestmt    { $1 }
    | labelstmt     { $1 }
    | letstmt       { $1 }
    | messagestmt   { $1 }
    | orderstmt     { $1 }
    | playstmt      { $1 }
    | rotatestmt    { $1 }
```

```
    | shufflestmt   { $1 }
    | skipstmt      { $1 }
    | winnerstmt    { $1 }

/* stmt */
askstmt: ASK ref questionblock  { Ast.Ask($2, $3, pos 1) }

/* question list */
questionblock: LCURLY questions RCURLY  { List.rev $2 }

/* question list reversed */
questions: question            { [$1] }
         | questions question  { $2 :: $1 }

/* question */
question: STRINGLIT block          { Ast.Uncond($1, $2, pos 1) }
        | STRINGLIT IF expr block  { Ast.Cond($1, $3, $4, pos 1) }

/* stmt */
assignstmt: ref ASSIGN expr DOT  { Ast.Assign($1, $3, pos 1) }

/* stmt */
compoundstmt: ref PLUSEQ   expr DOT
                { Ast.Compound($1, Ast.Plus, $3, pos 1) }

            | ref MINUSEQ  expr DOT
                { Ast.Compound($1, Ast.Minus, $3, pos 1) }

            | ref TIMESEQ  expr DOT
                { Ast.Compound($1, Ast.Times, $3, pos 1) }

            | ref DIVIDEEQ expr DOT
                { Ast.Compound($1, Ast.Divide, $3, pos 1) }

/* stmt */
dealstmt: DEAL ALL FROM ref TO ref DOT
            { Ast.Deal(Ast.EmptyExpr, $4, $6, pos 1) }

        | DEAL expr FROM ref TO ref DOT
            { Ast.Deal($2, $4, $6, pos 1) }

/* stmt */
foreverstmt: FOREVER block  { Ast.Forever($2, pos 1) }

/* stmt */
forstmt: FOR ID IN expr block
           { Ast.For($2, $4, Ast.EmptyExpr, $5, pos 1) }

       | FOR ID IN expr STARTING AT expr block
           { Ast.For($2, $4, $7, $8, pos 1) }


/* stmt */
ifstmt: IF expr block elseifs elseopt
          { Ast.If([($2, $3)] @ (List.rev $4) @ $5, pos 1) }

/* (expr, block) list reversed */
elseifs: /* Empty */                 { [] }
       | elseifs ELSEIF expr block  { ($3, $4) :: $1 }

/* (expr, block) list */
elseopt: /* Empty */ { [] }
       | ELSE block  { [(Ast.EmptyExpr, $2)] }

/* stmt */
invokestmt: ID LPAREN RPAREN DOT  { Ast.Invoke($1, pos 1) }

/* stmt */
labelstmt: LABEL ID DOT  { Ast.Label($2, pos 1) }

/* stmt */
letstmt: LET ID BE expr DOT  { Ast.Let($2, $4, pos 1) }

/* stmt */
messagestmt: MESSAGE expr DOT
               { Ast.Message(Ast.EmptyRef, $2, pos 1) }
```

```
          | MESSAGE ref expr DOT
              { Ast.Message($2, $3, pos 1) }

/* stmt */
orderstmt: ORDER ref BY ID DOT  { Ast.Order($2, $4, pos 1) }

/* stmt */
playstmt: PLAY ID FROM ref TO ref DOT  { Ast.Play($2, $4, $6, pos 1) }

/* stmt */
rotatestmt: ROTATE ref DOT  { Ast.Rotate($2, pos 1) }

/* stmt */
shufflestmt: SHUFFLE ref DOT  { Ast.Shuffle($2, pos 1) }

/* stmt */
skipstmt: SKIP TO ID DOT  { Ast.Skip($3, pos 1) }

/* stmt */
winnerstmt: WINNER ref DOT  { Ast.Winner($2, pos 1) }


/* Declarable Types */

/* dtype */
dtype: BOOLEAN     { Ast.Boolean }
     | CARD        { Ast.Card }
     | CARDLIST    { Ast.CardList }
     | DECK        { Ast.Deck }
     | NUMBER      { Ast.Number }
     | PLAYER      { Ast.Player }
     | PLAYERLIST  { Ast.PlayerList }
     | RANK        { Ast.Rank }
     | RANKLIST    { Ast.RankList }
     | STRING      { Ast.String }
     | SUIT        { Ast.Suit }
     | SUITLIST    { Ast.SuitList }
     | TEAM        { Ast.Team }
     | TEAMLIST    { Ast.TeamList }


/* Expressions */

/* expr */
expr: LPAREN expr RPAREN  { $2 }
    | expr PLUS expr      { Ast.Binary($1, Ast.Plus, $3, pos 2) }
    | expr MINUS expr     { Ast.Binary($1, Ast.Minus, $3, pos 2) }
    | expr TIMES expr     { Ast.Binary($1, Ast.Times, $3, pos 2) }
    | expr DIVIDE expr    { Ast.Binary($1, Ast.Divide, $3, pos 2) }
    | expr EQ expr        { Ast.Binary($1, Ast.Eq, $3, pos 2) }
    | expr NOTEQ expr     { Ast.Binary($1, Ast.NotEq, $3, pos 2) }
    | expr LT expr        { Ast.Binary($1, Ast.Lt, $3, pos 2) }
    | expr LTEQ expr      { Ast.Binary($1, Ast.LtEq, $3, pos 2) }
    | expr GT expr        { Ast.Binary($1, Ast.Gt, $3, pos 2) }
    | expr GTEQ expr      { Ast.Binary($1, Ast.GtEq, $3, pos 2) }
    | expr AND expr       { Ast.Binary($1, Ast.And, $3, pos 2) }
    | expr OR expr        { Ast.Binary($1, Ast.Or, $3, pos 2) }
    | NOT expr            { Ast.Not($2, pos 1) }
    | expr IN expr        { Ast.Binary($1, Ast.In, $3, pos 2) }
    | DEFINED ref         { Ast.Defined($2, pos 1) }

    | CANPLAY ID FROM ref TO ref { Ast.CanPlay($2, $4, $6, pos 1) }

    | rankexpr TILDE suitexpr { Ast.CardExpr($1, pos 2, $3) }
    | PERCENT  TILDE suitexpr { Ast.WildRank(pos 1, pos 2, $3) }
    | rankexpr TILDE PERCENT  { Ast.WildSuit($1, pos 2, pos 3) }
    | PERCENT  TILDE PERCENT  { Ast.WildCard(pos 1, pos 2, pos 3) }

    | list  { Ast.List($1, pos 1) }
    | ref   { Ast.Ref($1, pos 1) }
    | lit   { $1 }

/* expr */
rankexpr:
      rankexpr DOTDOT rankexpr  { Ast.RangeExpr($1, $3, pos 2) }
```

```
    | rankexpr COMMA rankexpr   { Ast.SeqExpr($1, $3, pos 2) }
    | ref                       { Ast.Ref($1, pos 1) }
    | ranklit                   { $1 }
    | NUMLIT                    { Ast.NumLit($1, pos 1) }

/* expr */
suitexpr:
      suitexpr COMMA suitexpr  { Ast.SeqExpr($1, $3, pos 2) }
    | ref                      { Ast.Ref($1, pos 1) }
    | suitlit                  { $1 }

/* expr list */
list: LSQUARE listitemsopt RSQUARE  { $2 }

/* expr list */
listitemsopt: /* Empty */ { [] }
            | listitems   { List.rev $1 }

/* expr list reversed */
listitems: expr                 { [$1] }
         | listitems SEMI expr  { $3 :: $1 }

/* ref */
ref: ID                           { Ast.Id($1, pos 1) }
   | ref ARROW ID                 { Ast.Prop(Ast.Ref($1, pos 1), $3, pos 3) }
   | LPAREN expr RPAREN ARROW ID { Ast.Prop($2, $5, pos 5) }
   | builtin                      { $1 }

/* ref */
builtin: PLAYERS   { Ast.Players(pos 1) }
       | TEAMS     { Ast.Teams(pos 1) }
       | STANDARD  { Ast.Standard(pos 1) }


/* Literals */

/* expr */
lit: ranklit    { $1 }
   | suitlit    { $1 }
   | STRINGLIT  { Ast.StringLit($1, pos 1) }
   | NUMLIT     { Ast.NumLit($1, pos 1) }
   | TRUE       { Ast.BoolLit(true, pos 1) }
   | FALSE      { Ast.BoolLit(false, pos 1) }

/* expr */
ranklit: ACE     { Ast.RankLit(Ast.Ace, pos 1) }
       | KING    { Ast.RankLit(Ast.King, pos 1) }
       | QUEEN   { Ast.RankLit(Ast.Queen, pos 1) }
       | JACK    { Ast.RankLit(Ast.Jack, pos 1) }

/* expr */
suitlit: CLUBS     { Ast.SuitLit(Ast.Clubs, pos 1) }
       | HEARTS    { Ast.SuitLit(Ast.Hearts, pos 1) }
       | SPADES    { Ast.SuitLit(Ast.Spades, pos 1) }
       | DIAMONDS  { Ast.SuitLit(Ast.Diamonds, pos 1) }
```

9.1.4 Abstract Syntax Tree

Listing 9.4: ast.ml

```
(*
 * This module defines the type of the basic AST
 * that is created by the parser.
 *
 * There are also utilities to turn AST nodes
 * into strings that can be re-parsed back into
 * AST nodes.
 *
 * This represents a syntactically valid program.
 *)

(***********)
(* Helpers *)
```

```
let str_indent ind = String.make ind ' '


(**********)
(* Basics *)

(* Position: Line and Column *)
type pos = int * int

(* Identifiers are strings in the AST *)
type id = string


(***************)
(* Expressions *)

(* Ranks *)
(* Minor must be 2..10 *)
type rank = Ace | King | Queen | Jack | Minor of int

(* Unparse a rank *)
let str_rank = function
    Ace      -> "A"
  | King     -> "K"
  | Queen    -> "Q"
  | Jack     -> "J"
  | Minor(i) -> (string_of_int i)


(* Suits *)
type suit = Clubs | Hearts | Spades | Diamonds

(* Un-parse a suit *)
let str_suit = function
    Clubs    -> "C"
  | Hearts   -> "H"
  | Spades   -> "S"
  | Diamonds -> "D"


(* Operators *)
type op = Plus | Minus | Times | Divide | Eq | NotEq
        | Lt | LtEq | Gt | GtEq | And | Or | In

(* Un-parse an operator *)
let str_op = function
    Plus   -> "+"
  | Minus  -> "-"
  | Times  -> "*"
  | Divide -> "/"
  | Eq     -> "=="
  | NotEq  -> "!="
  | Lt     -> "<"
  | LtEq   -> "<="
  | Gt     -> ">"
  | GtEq   -> ">="
  | And    -> "and"
  | Or     -> "or"
  | In     -> "in"


(* Expressions and References *)
(* Some have positions for keywords or operators *)
type expr =
  | Not of expr * pos
  | Binary of expr * op * expr * pos
  | Defined of reference * pos
  | CanPlay of id * reference * reference * pos
  | CardExpr of expr * pos * expr
  | RangeExpr of expr * expr * pos
  | SeqExpr of expr * expr * pos
  | WildRank of pos * pos * expr
  | WildSuit of expr * pos * pos
  | WildCard of pos * pos * pos
  | List of expr list * pos
```

```
  | Ref of reference * pos
  | BoolLit of bool * pos
  | NumLit of int * pos
  | StringLit of str_lit * pos
  | RankLit of rank * pos
  | SuitLit of suit * pos
  | EmptyExpr

and reference =
    Id of id * pos
  | Prop of expr * id * pos
  | Players of pos
  | Teams of pos
  | Standard of pos
  | EmptyRef  (* Used in a few places for opt ref *)

and str_part =
    StrRef of reference
  | StrLit of string

and str_lit = str_part list


(* Unparse an expression *)
let rec str_expr = function
    Not(e, _) ->
      "not " ^ (str_pexpr e)

  | Binary(e1, o, e2, _) ->
      (str_pexpr e1) ^ " " ^ (str_op o) ^ " " ^ (str_pexpr e2)

  | Defined(r, _) ->
      "defined " ^ (str_ref r)

  | CanPlay(s, r1, r2, _) ->
      "canplay " ^ s ^ " from " ^ (str_ref r1) ^ " to " ^ (str_ref r2)

  | CardExpr(e1, _, e2) -> (str_expr e1) ^ "~" ^ (str_expr e2)

  | RangeExpr(e1, e2, _) -> (str_expr e1) ^ ".." ^ (str_expr e2)

  | SeqExpr(e1, e2, _) -> (str_expr e1) ^ "," ^ (str_expr e2)

  | WildRank(_, _, e) -> "%~" ^ (str_expr e)

  | WildSuit(e, _, _) -> (str_expr e) ^ "~%"

  | WildCard(_, _, _) -> "%~%"

  | List(es, _) ->
      "[" ^ (Utils.catmap "; " str_expr es) ^ "]"

  | Ref(r, _) -> str_ref r

  | BoolLit(b, _) ->
      if b then "True" else "False"

  | NumLit(n, _) -> string_of_int n

  | StringLit(s, _) -> str_escape s
  | RankLit(r, _) -> str_rank r
  | SuitLit(s, _) -> str_suit s
  | EmptyExpr -> ""

and str_ref = function
    Id(s, _)      -> s
  | Prop(e, s, _) -> (str_pexpr e) ^ "->" ^ s
  | Players(_)    -> "players"
  | Teams(_)      -> "teams"
  | Standard(_)   -> "standard"
  | EmptyRef      -> ""

and str_escape lit =
  let str_cescape = function
      '\\' -> "\\\\"
    | '\n' -> "\\n"
```

```
    | '\t' -> "\\t"
    | '"'  -> "\\\""
    | '{'  -> "\\{"
    | '}'  -> "\\}"
    | x    -> String.make 1 x
  in
  let lit_str = function
      StrRef(r) -> "{" ^ (str_ref r) ^ "}"
    | StrLit(s) -> Utils.str_map str_cescape s
  in
  "\"" ^ (Utils.catmap "" lit_str lit) ^ "\""

and str_pexpr e = "(" ^ (str_expr e) ^ ")"


(**************)
(* Statements *)

(* The position marks the line/col of the beginning of the statement *)
type stmt =
    Ask of reference * question list * pos
  | Assign of reference * expr * pos

    (* Compound: Op can only be plus, minus, times, or divide *)
  | Compound of reference * op * expr * pos

    (* Deal: First expr empty for "all" *)
  | Deal of expr * reference * reference * pos

  | Forever of block * pos

    (* For: Second expr could be empty - start at beginning of list *)
  | For of id * expr * expr * block * pos

    (* If: Each expr/block pair is if..elseif..elseif.
           Last pair may have EmptyExpr for unconditional else. *)
  | If of (expr * block) list * pos

  | Invoke of id * pos
  | Label of id * pos
  | Let of id * expr * pos

    (* Message: Empty ref means message to all *)
  | Message of reference * expr * pos

  | Order of reference * id * pos
  | Play of id * reference * reference * pos
  | Rotate of reference * pos
  | Shuffle of reference * pos
  | Skip of id * pos
  | Winner of reference * pos

and question =
    Uncond of str_lit * block * pos
  | Cond of str_lit * expr * block * pos

and block = stmt list

(* Un-parse a statement *)
let rec str_stmt ind stmt =
  (str_indent ind) ^
  (match stmt with
    Ask(r, qs, _) ->
      "ask " ^ (str_ref r) ^ " " ^ (str_quests ind qs)

  | Assign(r, e, _) ->
      (str_ref r) ^ " = " ^ (str_expr e) ^ ".\n"

  | Compound(r, o, e, _) ->
      (str_ref r) ^ " " ^ (str_op o) ^ "= " ^ (str_expr e) ^ ".\n"

  | Deal(e, r1, r2, _) ->
      let what = function
          EmptyExpr -> "all"
        | e         -> str_expr e
      in
```

```
      "deal " ^ (what e) ^ " from " ^ (str_ref r1) ^
                           " to " ^ (str_ref r2) ^ ".\n"

  | Forever(ss, _) ->
      "forever " ^ (str_block ind ss)

  | For(s, e1, e2, ss, _) ->
      let starting = function
          EmptyExpr -> ""
        | e         -> " starting at " ^ (str_expr e2)
      in
      "for " ^ s ^ " in " ^ (str_expr e1) ^ (starting e2) ^
                     " " ^ (str_block ind ss)

  | If(ess, _) -> str_ifs ind ess

  | Invoke(s, _) -> s ^ "().\n"

  | Label(s, _) -> "label " ^ s ^ ".\n"

  | Let(s, e, _) -> "let " ^ s ^ " be " ^ (str_expr e) ^ ".\n"

  | Message(r, e, _) ->
      let who = function
          EmptyRef -> ""
        | r        -> (str_ref r) ^ " "
      in
      "message " ^ (who r) ^ (str_expr e) ^ ".\n"

  | Order(r, s, _) ->
      "order " ^ (str_ref r) ^ " by " ^ s ^ ".\n"

  | Play(s, r1, r2, _) ->
      "play " ^ s ^ " from " ^ (str_ref r1) ^ " to " ^ (str_ref r2) ^ ".\n"

  | Rotate(r, _) -> "rotate " ^ (str_ref r) ^ ".\n"

  | Shuffle(r, _) -> "shuffle " ^ (str_ref r) ^ ".\n"

  | Skip(s, _) -> "skip to " ^ s ^ ".\n"

  | Winner(r, _) -> "winner " ^ (str_ref r) ^ ".\n"

  )

and str_quests ind qs =
  "{\n" ^ (Utils.catmap "" (str_quest (ind + 2)) qs) ^ (str_indent ind) ^ "}\n"

and str_quest ind q =
  (str_indent ind) ^
  (match q with
    Uncond(s, ss, _) -> (str_escape s) ^ " " ^ (str_block ind ss)
  | Cond(s, e, ss, _) -> (str_escape s) ^ " if " ^ (str_expr e)
                                              ^ " " ^ (str_block ind ss)
  )

and str_block ind ss =
  "{\n" ^ (Utils.catmap "" (str_stmt (ind + 2)) ss) ^ (str_indent ind) ^ "}\n"

and str_ifs ind ess =
  match ess with
    [] -> raise(Failure("Unexpected empty if/elseif/else list"))

  | ((e, ss) :: rest) ->
      "if " ^ (str_expr e) ^ " " ^ (str_block ind ss) ^
              (Utils.catmap "" (str_elses ind) rest)

and str_elses ind (e, ss) =
  match e with
      EmptyExpr -> (str_indent ind) ^ "else " ^ (str_block ind ss)
    | e         -> (str_indent ind) ^ "elseif " ^ (str_expr e) ^ " " ^ (str_block ind ss)


(****************)
(* Declarations *)
```

```
(* Declarable types *)
type dtype = Boolean
           | Card
           | CardList
           | Deck
           | Number
           | Player
           | PlayerList
           | Rank
           | RankList
           | String
           | Suit
           | SuitList
           | Team
           | TeamList

(* Un-parse a type *)
let str_dtype = function
    Boolean    -> "Boolean"
  | Card       -> "Card"
  | CardList   -> "CardList"
  | Deck       -> "Deck"
  | Number     -> "Number"
  | Player     -> "Player"
  | PlayerList -> "PlayerList"
  | Rank       -> "Rank"
  | RankList   -> "RankList"
  | String     -> "String"
  | Suit       -> "Suit"
  | SuitList   -> "SuitList"
  | Team       -> "Team"
  | TeamList   -> "TeamList"


(* For Area decl *)
type areaopt = Faceup | Facedown | Squaredup | Spreadout

type areaopts = (areaopt * pos) list

(* Un-parse area options *)
let str_areaopt = function
    (Faceup, _)    -> "faceup"
  | (Facedown, _)  -> "facedown"
  | (Squaredup, _) -> "squaredup"
  | (Spreadout, _) -> "spreadout"

let str_areaopts opts = Utils.catmap ", " str_areaopt opts


(* Declarations *)
type decl =
    Area of id * str_lit * areaopts * pos  (* Id, Display, Opts *)
  | Action of id * block * pos             (* Id, Body *)
  | Rule of id * id list * expr * pos      (* Id, Args, Body *)
  | Ordering of id * id * expr * pos       (* Id, Arg, Body *)
  | Var of dtype * id * expr * pos         (* Type, Id, Init *)

(* Un-parse a declaration *)
let rec str_decl = function
    Area(s1, s2, [], _) ->
      "Area " ^ s1 ^ " labeled " ^ (str_escape s2) ^ ".\n"

  | Area(s1, s2, opts, _) ->
      "Area " ^ s1 ^ " labeled " ^ (str_escape s2) ^ " is "
                   ^ (str_areaopts opts) ^ ".\n"

  | Action(s, ss, _) ->
      "Action " ^ s ^ " " ^ (str_block 0 ss)

  | Rule(s, ss, e, _) ->
      "Rule " ^ s ^ (str_args ss) ^ " = " ^ (str_expr e) ^ ".\n"

  | Ordering(s, a, e, _) ->
      "Ordering " ^ s ^ (str_args [a]) ^ " = " ^ (str_expr e) ^ ".\n"

  | Var(t, s, e, _) ->
```

```
      let init = function
          EmptyExpr -> ""
        | e         -> " = " ^ (str_expr e)
      in
      (str_dtype t) ^ " " ^ s ^ (init e) ^ ".\n"

and str_args ss = "(" ^ (String.concat ", " ss) ^ ")"

(* Un-parse a list of declarations *)
let str_decls ds = Utils.catmap "" str_decl ds


(************)
(* The Game *)

(* Player/team designations *)
type ptcount =
    PlayerCount of int list * pos      (* 2, 3, ... players *)
  | TeamCount of int list * int * pos  (* 2, 3, ... teams of 2 *)

(* Game name, player/team count, list of decls, position of Game keyword *)
type game = str_lit * ptcount * decl list * pos

(* Un-parse a game *)
let rec str_game (s, c, ds, _) =
  "Game " ^ (str_escape s) ^ " requires " ^
      (str_ptcount c) ^ ".\n" ^ (str_decls ds)

and str_ptcount = function
    PlayerCount(ns, _)  ->
      (Utils.catmap " or " string_of_int ns) ^ " players"

  | TeamCount(ns, n, _) ->
      (Utils.catmap " or " string_of_int ns) ^
                   " teams of " ^ (string_of_int n)
```

9.1.5 Checked Abstract Syntax Tree

Listing 9.5: cast.ml

```
(*
 * This module defines the checked AST structure.
 *
 * This is like the AST but with all identifiers
 * resolved and all types checked.
 *
 * This represents a semantically valid program and builds
 * on a lot of the AST nodes.
 *)

(**********)
(* Basics *)

(* Handle - Refers to a variable's location in the environment *)
type handle = int

(* Offset - Refers to a property in an Object value's prop array *)
type offset = int

(* Position *)
type pos = Ast.pos

(* Expression and Types *)
type t = Area | Boolean | Card | Number
       | Player | Rank | String | Suit | Team
       | L of t
       | EmptyType

let str_type = function
    Area      -> "Area"
  | Boolean   -> "Boolean"
  | Card      -> "Card"
  | Number    -> "Number"
```

```
  | Player    -> "Player"
  | Rank      -> "Rank"
  | String    -> "String"
  | Suit      -> "Suit"
  | Team      -> "Team"

  | L(Boolean) -> "BooleanList"
  | L(Card)    -> "CardList"
  | L(Number)  -> "NumberList"
  | L(Player)  -> "PlayerList"
  | L(Rank)    -> "RankList"
  | L(String)  -> "StringList"
  | L(Suit)    -> "SuitList"
  | L(Team)    -> "TeamList"

  | EmptyType -> "EmptyType"
  | _ -> raise(Failure("Internal Error: Unexpected type"))


(* Rank and Suit *)
type rank = Ast.rank
type suit = Ast.suit

(* Values *)
type value =
    BoolVal of bool
  | NumVal of int
  | StrVal of string
  | RankVal of rank
  | SuitVal of suit
  | ObjVal of value array * t      (* Properties *)
  | ListVal of value list ref * t  (* Elements *)
  | UndefVal


(******************************************)
(* Expressions, References, and Declarations *)

type op = Ast.op

type expr =
  | Not of expr * pos
  | Binary of expr * op * expr * pos
  | Defined of reference * pos
  | CanPlay of rule * reference * reference * pos
  | CardExpr of expr * pos * expr
  | RangeExpr of expr * expr * pos
  | RankSeqExpr of expr * expr * pos
  | SuitSeqExpr of expr * expr * pos
  | WildRank of pos * pos * expr
  | WildSuit of expr * pos * pos
  | WildCard of pos * pos * pos
  | List of expr list * t * pos
  | Ref of reference * pos
  | Literal of value * pos
  | StrLiteral of str_lit * pos
  | EmptyExpr

and reference =
    Var of variable * pos
  | Prop of expr * prop * pos
  | EmptyRef

and variable =
    Global of value ref * t
  | Local of handle * t
  | PlayersVar
  | TeamsVar
  | StandardVar

and lprop = Size | First | Last | Top | Bottom

and prop =
    ObjProp of offset * t
  | ListProp of lprop * t
```

```
and str_part =
    StrRef of reference
  | StrLit of string

and str_lit = str_part list

and rule = expr * pos

and ordering = expr * pos


(***************)
(* Declarations *)

type var_decl =
    VarDecl of variable * expr * pos
  | AreaDecl of variable * str_lit * bool * bool * pos


(*************)
(* Statements *)

type stmt =
    Ask of reference * question list * pos
  | Assign of reference * expr * pos

    (* Compound: Op can only be plus, minus, times, or divide *)
  | Compound of reference * op * expr * pos

    (* DealCard: First expr is Card *)
  | DealCard of expr * reference * reference * pos

    (* DealCards: First expr empty for "all" or Numeric *)
  | DealCards of expr * reference * reference * pos

  | Forever of block * pos

    (* For: Second expr could be empty - start at beginning of list *)
  | For of variable * expr * expr * block * pos

    (* If: Each expr/block pair is if..elseif..elseif.
           Last pair may have EmptyExpr for unconditional else. *)
  | If of (expr * block) list * pos

  | Invoke of action * pos
  | Label of string * pos
  | Let of var_decl * pos

    (* Message: Empty ref means message to all *)
  | Message of reference * expr * pos

  | Order of reference * ordering * pos
  | Play of rule * reference * reference * pos
  | Rotate of reference * pos
  | Shuffle of reference * pos
  | Skip of string * pos
  | Winner of reference * pos

and question =
    Uncond of str_lit * block * pos
  | Cond of str_lit * expr * block * pos

and block = stmt list

  (* Action: Number of locals and a body *)
and action = int * block * pos


(***********)
(* The Game *)

type ptcount = Ast.ptcount

(* Name, number of players, global variables, and the main action *)
type game = str_lit * ptcount * var_decl list * action * pos
```

9.1.6 Semantic Analyzer

Listing 9.6: semantic.ml

```
(*
 * This module performs semantic analysis on the Ast resolving
 * symbols and type checking the program. The result is a
 * Cast ("Checked Ast") that can be interpreted directly with
 * no potential unresolved symbol errors or type errors.
 *)

module StringMap = Map.Make(String)

(*
 * Utils
 *)

(*
 * Merge two StringMaps. If a key exists in both, the value
 * in the second map will win.
 *)
let map_merge : 'a StringMap.t -> 'a StringMap.t -> 'a StringMap.t =
  fun m1 m2 -> StringMap.fold StringMap.add m2 m1

(*
 * When given a StringMap and a string, this raises an error if
 * the given key is already in the map. Otherwise, it does nothing.
 *)
let assert_no_dup : 'a StringMap.t -> string -> Cast.pos
                                    -> string -> unit =
  fun map id p s ->
    if StringMap.mem id map
      then Utils.pos_error p ("Duplicate " ^ s ^ ": " ^ id)
      else ()


(*
 * Scopes
 *)

(*
 * There is one scope for global variables and
 * one scope for each new Cast.block that is entered.
 *
 * Each scope contains a lookup table for resolving
 * variables referenced in that scope.
 *)
type scope = Cast.variable StringMap.t

(*
 * Add a variable to the given scope, returning a new scope.
 * Ensure the scope doesn't contain a variable of the given name.
 *)
let scope_add_var : scope -> string -> Cast.variable -> Cast.pos -> scope =
  fun sc id var p ->
    assert_no_dup sc id p "Variable";
    StringMap.add id var sc


(*
 * Environment
 *)

(*
 * The environment contains a list of scopes for variables,
 * which are resolved differently depending on the current
 * scope, and a single lookup table for actions, rules, and
 * orderings, which are always global.
 *
 * The scope list contains the current scope at the head.
 * Successive scopes in the list represent successively
 * nested scopes in the program with the last scope in the
 * list being the global scope.
 *)
type env = { e_vars      : scope list;
             e_actions   : Cast.action StringMap.t;
```

```
             e_rules      : Cast.rule StringMap.t;
             e_orderings : Cast.ordering StringMap.t; }

(* Push a new, empty scope on to the env, returning a new env. *)
let env_push_scope : env -> env =
  fun ev -> { ev with e_vars = StringMap.empty :: ev.e_vars }

(* Add a variable to env's current scope, returning a new env. *)
let env_add_var : env -> string -> Cast.variable -> Cast.pos -> env =
  fun ev id var p ->
    match ev.e_vars with
      []       -> Utils.ie "No current scope"
    | (h :: t) -> { ev with e_vars = scope_add_var h id var p :: t }

(*
 * Add an action to the env, returning a new env.
 * Ensure the env doesn't contain an action of the given name.
 *)
let env_add_action : env -> string -> Cast.action -> Cast.pos -> env =
  fun ev id act p ->
    let acts = ev.e_actions in
    assert_no_dup acts id p "Action";
    { ev with e_actions = StringMap.add id act acts }

(*
 * Add a rule to the env, returning a new env.
 *
 * It is a semantic error if the env already contains a rule
 * of the given name.
 *)
let env_add_rule : env -> string -> Cast.rule -> Cast.pos -> env =
  fun ev id rule p ->
    let rules = ev.e_rules in
    assert_no_dup rules id p "Rule";
    { ev with e_rules = StringMap.add id rule rules }

(*
 * Add an ordering to the env, returning a new env.
 *
 * It is a semantic error if the env already contains an ordering
 * of the given name.
 *)
let env_add_ordering : env -> string -> Cast.ordering -> Cast.pos -> env =
  fun ev id ord p ->
    let ords = ev.e_orderings in
    assert_no_dup ords id p "Ordering";
    { ev with e_orderings = StringMap.add id ord ords }


(*
 * Semantic analysis of types
 *)

(*
 * Given an Ast.dtype, this results in the corresponding Cast.t
 *)
let sem_type : Ast.dtype -> Cast.t =
  function
    Ast.Boolean    -> Cast.Boolean
  | Ast.Card       -> Cast.Card
  | Ast.Number     -> Cast.Number
  | Ast.Player     -> Cast.Player
  | Ast.Rank       -> Cast.Rank
  | Ast.String     -> Cast.String
  | Ast.Suit       -> Cast.Suit
  | Ast.Team       -> Cast.Team

  | Ast.CardList   -> Cast.L(Cast.Card)
  | Ast.Deck       -> Cast.L(Cast.Card)
  | Ast.PlayerList -> Cast.L(Cast.Player)
  | Ast.RankList   -> Cast.L(Cast.Rank)
  | Ast.SuitList   -> Cast.L(Cast.Suit)
  | Ast.TeamList   -> Cast.L(Cast.Team)

(* Return the element type if t is a list type; otherwise return t *)
let element_type : Cast.t -> Cast.t =
```

```
  function Cast.L(t) -> t
         | t -> t

(* Decide if the second type can be used where the first is expected. *)
let rec ok_type : Cast.t -> Cast.t -> bool =
  fun exp act ->
    match (exp, act) with
      (_, Cast.EmptyType)       -> true
    | (Cast.EmptyType, _)       -> true
    | (Cast.Rank, Cast.Number) -> true
    | (Cast.Number, Cast.Rank) -> true
    | (Cast.L(x), Cast.L(y))   -> ok_type x y
    | (x, y)                     -> x = y

(*
 * Ensure the second type can be used where the first type
 * was expected.
 *)
let assert_ok_type : Cast.t -> Cast.t -> Cast.pos -> unit =
  fun t1 t2 p ->
    if not (ok_type t1 t2)
      then Utils.pos_error p
             ("Expected type " ^ (Cast.str_type t1) ^
              " but found type " ^ (Cast.str_type t2))

(*
 * Ensure the type is a list type. Return the
 * type of its elements.
 *)
let assert_list_type : Cast.t -> Cast.pos -> Cast.t =
  fun t p ->
    let elt = element_type t in
    if elt = t
      then Utils.pos_error p
             ("Expected a list type but found type " ^ (Cast.str_type t));
    elt

(*
 * Ensure the second type is ok in a list literal that
 * already contains elements of the first type.
 *
 * This returns the new element type of the list, which
 * may be different if the current element type is
 * unknown (Cast.EmptyType).
 *)
let assert_listlit_type : Cast.t -> Cast.t -> Cast.pos -> Cast.t =
  fun old t p ->
    let oldelt = element_type old in
    let elt    = element_type t in
    assert_ok_type oldelt elt p;
    elt

(* Ensure the type matches one in the list using assert_ok_type. *)
let assert_type_opts : Cast.t list -> Cast.t -> Cast.pos -> unit =
  fun ts t p ->
    let to_str ts = Utils.catmap " or " Cast.str_type ts in
    if not (List.exists (fun t' -> ok_type t' t) ts)
      then Utils.pos_error p
             ("Expected type to be one of " ^ (to_str ts) ^
              " but found type " ^ (Cast.str_type t))

(* Ensure the type is valid to deal from. *)
let assert_deal_from_type : Cast.t -> Cast.pos -> unit =
  fun t p ->
    let ts = [Cast.L(Cast.Card); Cast.Player; Cast.Area] in
    assert_type_opts ts t p

(* Ensure the type is valid to deal to. *)
let assert_deal_to_type : Cast.t -> Cast.pos -> unit =
  fun t p ->
    let ts = [Cast.L(Cast.Card); Cast.Player;
              Cast.L(Cast.Player); Cast.Area] in
    assert_type_opts ts t p

(* Ensure the type is valid for a "play" and "canplay" destination. *)
let assert_play_type : Cast.t -> Cast.pos -> unit =
```

```
  fun t p ->
    let ts = [Cast.L(Cast.Card); Cast.Player; Cast.Area] in
    assert_type_opts ts t p

(* Ensure the type is valid for a "message" statement. *)
let assert_msg_type : Cast.t -> Cast.pos -> unit =
  fun t p ->
    let ts = [Cast.Player; Cast.L(Cast.Player); Cast.Team] in
    assert_type_opts ts t p

(* Ensure the type is valid for a "winner" statement. *)
let assert_winner_type : Cast.t -> Cast.pos -> unit =
  fun t p ->
    let ts = [Cast.Player; Cast.Team] in
    assert_type_opts ts t p

(*
 * Ensure the types are valid for the binary operator.
 * Return the result type.
 *)
let assert_op_types : Cast.op -> Cast.t -> Cast.t
                   -> Cast.pos -> Cast.pos -> Cast.t =
  fun op t1 t2 p1 p2 ->
    match op with
      Ast.Plus ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Number
    | Ast.Minus ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Number
    | Ast.Times ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Number
    | Ast.Divide ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Number
    | Ast.Eq ->
        assert_ok_type t1 t2 p2;
        if t1 = Cast.Area
          then Utils.pos_error p1 "Invalid type Area for comparison";
        Cast.Boolean
    | Ast.NotEq ->
        assert_ok_type t1 t2 p2;
        if t1 = Cast.Area
          then Utils.pos_error p1 "Invalid type Area for comparison";
        Cast.Boolean
    | Ast.Lt ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Boolean
    | Ast.LtEq ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Boolean
    | Ast.Gt ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Boolean
    | Ast.GtEq ->
        assert_ok_type Cast.Number t1 p1;
        assert_ok_type Cast.Number t2 p2;
        Cast.Boolean
    | Ast.And ->
        assert_ok_type Cast.Boolean t1 p1;
        assert_ok_type Cast.Boolean t2 p2;
        Cast.Boolean
    | Ast.Or ->
        assert_ok_type Cast.Boolean t1 p1;
        assert_ok_type Cast.Boolean t2 p2;
        Cast.Boolean
    | Ast.In ->
        let lt = assert_list_type t2 p2 in
```

```
        if lt = Cast.Card
          then assert_type_opts [Cast.Card;
                                 Cast.Rank;
                                 Cast.Suit] t1 p1
          else assert_ok_type lt t1 p1;
        Cast.Boolean

(*
 * Ensure the types are valid for the left and right operand
 * of a Card expression. Return the result type.
 *)
let assert_cardexpr_types : Cast.t -> Cast.t
                         -> Cast.pos -> Cast.pos -> Cast.t =
  fun t1 t2 p1 p2 ->
    assert_type_opts [Cast.Rank; Cast.L(Cast.Rank)] t1 p1;
    assert_type_opts [Cast.Suit; Cast.L(Cast.Suit)] t2 p2;
    match (t1, t2) with
        (Cast.Rank, Cast.Suit)   -> Cast.Card
      | (Cast.Number, Cast.Suit) -> Cast.Card
      | _                        -> Cast.L(Cast.Card)

(*
 * Ensure the types are valid for the left and right operand
 * of a sequence expression. Return the result type.
 *)
let assert_seq_types : Cast.t -> Cast.t
                    -> Cast.pos -> Cast.pos -> Cast.t =
  fun t1 t2 p1 p2 ->
    let elt1 = element_type t1 in
    let elt2 = element_type t2 in
    let valid = [Cast.Rank; Cast.Suit] in
    assert_type_opts valid elt1 p1;
    assert_type_opts valid elt2 p2;
    assert_ok_type elt1 elt2 p2;
    Cast.L(elt1)

(* Given a variable, this results in its type. *)
let var_type : Cast.variable -> Cast.t =
  function
      Cast.Global(_, t) -> t
    | Cast.Local(_, t)  -> t
    | Cast.PlayersVar   -> Cast.L(Cast.Player)
    | Cast.TeamsVar     -> Cast.L(Cast.Team)
    | Cast.StandardVar  -> Cast.L(Cast.Card)

(* Given a property, this results in its type. *)
let prop_type : Cast.prop -> Cast.t =
  function
      Cast.ObjProp(_, t) -> t
    | Cast.ListProp(_, t) -> t

(* Ensure the type can be converted to a Cast.String. *)
let assert_str_conv_ok : Cast.t -> Cast.pos -> unit =
  fun t p ->
    let str_conv_ok = function
        Cast.Area       -> false
      | Cast.Boolean    -> true
      | Cast.Card       -> true
      | Cast.Number     -> true
      | Cast.Player     -> true
      | Cast.Rank       -> true
      | Cast.String     -> true
      | Cast.Suit       -> true
      | Cast.Team       -> true
      | Cast.L(_)       -> false
      | Cast.EmptyType  -> false
    in
    if str_conv_ok t
      then ()
      else Utils.pos_error p
             ("Can't convert " ^ (Cast.str_type t) ^ " to a String")


(*
 * Name resolution
 *)
```

```
(* This resolves the variable's name to a variable. *)
let rec res_var : scope list -> Ast.id -> Ast.pos -> Cast.variable =
  fun sl id p ->
    match sl with
      []       -> Utils.pos_error p ("Unknown identifier '" ^ id ^ "'")
    | (h :: t) -> try StringMap.find id h
                  with Not_found -> res_var t id p

(*
 * This resolves the type and property name into a property.
 *
 * A property consists of its offset into the array of
 * properties that object values are represented by,
 * along with its type.
 *
 * List properties are represented symbolically and are
 * calculated at run-time. They are read-only and have
 * no storage.
 *)
let res_prop : Cast.t -> string -> Cast.pos -> Cast.prop =
  fun t id p ->
    match (t, id) with
      (Cast.Card, "rank")           -> Cast.ObjProp(0, Cast.Rank)
    | (Cast.Card, "suit")           -> Cast.ObjProp(1, Cast.Suit)
    | (Cast.Card, "last_played_by") -> Cast.ObjProp(2, Cast.Player)

    | (Cast.Player, "name")  -> Cast.ObjProp(0, Cast.String)
    | (Cast.Player, "hand")  -> Cast.ObjProp(1, Cast.L(Cast.Card))
    | (Cast.Player, "stash") -> Cast.ObjProp(2, Cast.L(Cast.Card))
    | (Cast.Player, "score") -> Cast.ObjProp(3, Cast.Number)
    | (Cast.Player, "team")  -> Cast.ObjProp(4, Cast.Team)

    | (Cast.Team, "members") -> Cast.ObjProp(0, Cast.L(Cast.Player))
    | (Cast.Team, "stash")   -> Cast.ObjProp(1, Cast.L(Cast.Card))
    | (Cast.Team, "score")   -> Cast.ObjProp(2, Cast.Number)

    | (Cast.Area, "name")          -> Cast.ObjProp(0, Cast.String)
    | (Cast.Area, "cards")         -> Cast.ObjProp(1, Cast.L(Cast.Card))
    | (Cast.Area, "is_facedown")  -> Cast.ObjProp(2, Cast.Boolean)
    | (Cast.Area, "is_squaredup") -> Cast.ObjProp(3, Cast.Boolean)

    | (Cast.L(_), "size")   -> Cast.ListProp(Cast.Size, Cast.Number)
    | (Cast.L(t), "first")  -> Cast.ListProp(Cast.First, t)
    | (Cast.L(t), "last")   -> Cast.ListProp(Cast.Last, t)
    | (Cast.L(t), "top")    -> Cast.ListProp(Cast.Top, t)
    | (Cast.L(t), "bottom") -> Cast.ListProp(Cast.Bottom, t)

    | _ -> Utils.pos_error p ("Type " ^ (Cast.str_type t) ^
                              " has no property named " ^
                              "'" ^ id ^ "'")

(* This resolves the action's name to an action. *)
let res_action : env -> Ast.id -> Ast.pos -> Cast.action =
  fun ev id p -> try StringMap.find id ev.e_actions
                 with Not_found ->
                   Utils.pos_error p ("Unknown action '" ^ id ^ "'")

(* This resolves an ordering's name to an ordering. *)
let res_ordering : env -> Ast.id -> Ast.pos -> Cast.ordering =
  fun ev id p -> try StringMap.find id ev.e_orderings
                 with Not_found ->
                   Utils.pos_error p ("Unknown ordering '" ^ id ^ "'")

(* This resolves a rule's name to a rule. *)
let res_rule : env -> Ast.id -> Ast.pos -> Cast.rule =
  fun ev id p -> try StringMap.find id ev.e_rules
                 with Not_found ->
                   Utils.pos_error p ("Unknown rule '" ^ id ^ "'")



(*
 * Semantic analysis of expressions and references
 *)

(*
```

```
 * Given a reference, this results in its position.
 * For a property, this is the position of the property's name.
 *)
let ref_pos : Cast.reference -> Cast.pos =
  function
      Cast.Var(_, p)      -> p
    | Cast.Prop(_, _, p) -> p
    | Cast.EmptyRef       -> (-1, -1)

(* Given an expr, this results in its position. *)
let rec expr_pos : Cast.expr -> Cast.pos =
  function
      Cast.Not(_, p)             -> p
    | Cast.Binary(e, _, _, _)    -> expr_pos e
    | Cast.Defined(_, p)         -> p
    | Cast.CanPlay(_, _, _, p)   -> p
    | Cast.CardExpr(e, _, _)     -> expr_pos e
    | Cast.RangeExpr(e, _, _)    -> expr_pos e
    | Cast.RankSeqExpr(e, _, _) -> expr_pos e
    | Cast.SuitSeqExpr(e, _, _) -> expr_pos e
    | Cast.WildRank(p, _, _)     -> p
    | Cast.WildSuit(e, _, _)     -> expr_pos e
    | Cast.WildCard(p, _, _)     -> p
    | Cast.List(_, _, p)         -> p
    | Cast.Ref(_, p)             -> p
    | Cast.Literal(_, p)         -> p
    | Cast.StrLiteral(_, p)      -> p
    | Cast.EmptyExpr             -> (-1, -1)

(*
 * Analyze the Ast.reference, creating a Cast.reference
 * and its type.
 *)
let rec sem_ref : env -> Ast.reference -> Cast.reference * Cast.t =
  fun ev -> function
    Ast.Id(id, p)       -> let v = res_var ev.e_vars id p
                           in Cast.Var(v, p), var_type v
  | Ast.Prop(e, id, p) -> let (e, t) = sem_expr ev e in
                           let f = res_prop t id p in
                           Cast.Prop(e, f, p), prop_type f
  | Ast.Players(p)      -> let v = Cast.PlayersVar
                           in Cast.Var(v, p), var_type v
  | Ast.Teams(p)        -> let v = Cast.TeamsVar
                           in Cast.Var(v, p), var_type v
  | Ast.Standard(p)     -> let v = Cast.StandardVar
                           in Cast.Var(v, p), var_type v
  | Ast.EmptyRef        -> Cast.EmptyRef, Cast.EmptyType

(*
 * Analyze the Ast.expr, creating the Cast.expr and
 * its type.
 *)
and sem_expr : env -> Ast.expr -> Cast.expr * Cast.t =
  fun ev -> function
      Ast.Not(e, p) ->
        let (e, t) = sem_expr ev e in
        assert_ok_type Cast.Boolean t (expr_pos e);
        Cast.Not(e, p), Cast.Boolean
    | Ast.Binary(e1, op, e2, p) ->
        let (e1, t1) = sem_expr ev e1 in
        let (e2, t2) = sem_expr ev e2 in
        let p1 = expr_pos e1 in
        let p2 = expr_pos e2 in
        let t = assert_op_types op t1 t2 p1 p2 in
        Cast.Binary(e1, op, e2, p), t
    | Ast.Defined(r, p) ->
        let (r, _) = sem_ref ev r in
        Cast.Defined(r, p), Cast.Boolean
    | Ast.CanPlay(id, r1, r2, p) ->
        let rule = res_rule ev id p in
        let (r1, t1) = sem_ref ev r1 in
        assert_ok_type Cast.Player t1 (ref_pos r1);
        let (r2, t2) = sem_ref ev r2 in
        assert_play_type t2 (ref_pos r2);
        Cast.CanPlay(rule, r1, r2, p), Cast.Boolean
    | Ast.CardExpr(e1, p, e2) ->
```

```
        let (e1, t1) = sem_expr ev e1 in
        let (e2, t2) = sem_expr ev e2 in
        let p1 = expr_pos e1 in
        let p2 = expr_pos e2 in
        let t = assert_cardexpr_types t1 t2 p1 p2 in
        Cast.CardExpr(e1, p, e2), t
    | Ast.RangeExpr(e1, e2, p) ->
        let (e1, t1) = sem_expr ev e1 in
        let (e2, t2) = sem_expr ev e2 in
        let p1 = expr_pos e1 in
        let p2 = expr_pos e2 in
        assert_ok_type Cast.Rank t1 p1;
        assert_ok_type Cast.Rank t2 p2;
        Cast.RangeExpr(e1, e2, p), Cast.L(Cast.Rank)
    | Ast.SeqExpr(e1, e2, p) ->
        let (e1, t1) = sem_expr ev e1 in
        let (e2, t2) = sem_expr ev e2 in
        let p1 = expr_pos e1 in
        let p2 = expr_pos e2 in
        let t = assert_seq_types t1 t2 p1 p2 in
        if t = Cast.L(Cast.Rank) then
          Cast.RankSeqExpr(e1, e2, p), t
        else
          Cast.SuitSeqExpr(e1, e2, p), t
    | Ast.WildRank(p1, p2, e) ->
        let (e, t) = sem_expr ev e in
        let p = expr_pos e in
        assert_type_opts [Cast.Suit; Cast.L(Cast.Suit)] t p;
        Cast.WildRank(p1, p2, e), Cast.L(Cast.Card)
    | Ast.WildSuit(e, p1, p2) ->
        let (e, t) = sem_expr ev e in
        let p = expr_pos e in
        assert_type_opts [Cast.Rank; Cast.L(Cast.Rank)] t p;
        Cast.WildSuit(e, p1, p2), Cast.L(Cast.Card)
    | Ast.WildCard(p1, p2, p3) ->
        Cast.WildCard(p1, p2, p3), Cast.L(Cast.Card)
    | Ast.List(es, p) ->
        let f (es, elt) e =
          let (e', elt') = sem_expr ev e in
          let elt' = assert_listlit_type elt elt' (expr_pos e') in
          (e' :: es, elt')
        in
        let (es, elt) = List.fold_left f ([], Cast.EmptyType) es in
        Cast.List(List.rev es, elt, p), Cast.L(elt)
    | Ast.Ref(r, p) ->
        let (r, t) = sem_ref ev r in
        Cast.Ref(r, p), t
    | Ast.BoolLit(b, p) ->
        Cast.Literal(Cast.BoolVal(b), p), Cast.Boolean
    | Ast.NumLit(i, p) ->
        Cast.Literal(Cast.NumVal(i), p), Cast.Number
    | Ast.StringLit(sl, p) ->
        let sl = sem_str_lit ev sl in
        Cast.StrLiteral(sl, p), Cast.String
    | Ast.RankLit(rk, p) ->
        Cast.Literal(Cast.RankVal(rk), p), Cast.Rank
    | Ast.SuitLit(st, p) ->
        Cast.Literal(Cast.SuitVal(st), p), Cast.Suit
    | Ast.EmptyExpr ->
        Cast.EmptyExpr, Cast.EmptyType

(*
 * Analyze the Ast.str_lit, resolving any nested
 * references against the current env and ensuring
 * they can be converted to strings.
 *
 * This results in a Cast.str_lit.
 *)
and sem_str_lit : env -> Ast.str_lit -> Cast.str_lit =
  fun ev sl ->
    let sem_part ev = function
        Ast.StrRef(r) -> let (r, t) = sem_ref ev r in
                         assert_str_conv_ok t (ref_pos r);
                         Cast.StrRef(r)
      | Ast.StrLit(s) -> Cast.StrLit(s)
    in List.map (sem_part ev) sl
```

```
(*
 * Semantic analysis of statements
 *)

(*
 * This statement information is threaded through each statement
 * as it is analyzed.
 *
 * The si_locals member is a count of how many locals have
 * been declared so far in the entire action. It is set to 0
 * when an action is entered and it is incremented every
 * time a "let" or "for" introduces a new variable that
 * needs space.
 *
 * The si_skips member maps a skip name to the position at
 * which the skip was encountered. If more than one skip for
 * the same name is active, any one of those positions is
 * used, since only one is needed when reporting errors.
 * It is set to empty every time a new scope is entered,
 * and each time a scope is exited, the inner skips are
 * merged with the outer skips. Skips are removed from
 * this map when their corresponding label is found.
 * Any skips remaining at the end of an action were
 * unresolved.
 *
 * The si_labels member maps a label name to the position
 * at which it was encountered. This starts empty at
 * the beginning of an action and accumulates labels
 * for every statement in the action. It is used to detect
 * duplicate labels.
 *)
type sinfo = { si_locals : int;
               si_skips  : Cast.pos StringMap.t;
               si_labels : Cast.pos StringMap.t; }

(*
 * Increment the si_locals count and return a new sinfo.
 *)
let si_inc_locals : sinfo -> sinfo =
  fun si -> { si with si_locals = si.si_locals + 1 }

(*
 * Helper. This takes a triplet: ('a list, 'b, 'c) and
 * a function: (('a, 'b, 'c) -> ('d, 'b, 'c)) and acts
 * a bit like fold_left.
 *
 * The function will be applied to each item in the 'a list.
 * The 'b and 'c arguments for each application of the
 * function come from the return value of the previous
 * invocation of the function (they start with the given
 * 'b and 'c for thread).
 *
 * Each 'd value the function returns is accumulated
 * in to a list, and the final 'd list is returned
 * with the final 'b and 'c from the final invocation
 * of the function.
 *)
let thread2 : ('a * 'b * 'c -> 'd * 'b * 'c)
          -> 'a list * 'b * 'c
          -> 'd list * 'b * 'c =
  fun f (alist, b, c) ->
    let walk (dlist, b, c) a =
      (let (d, b, c) = f (a, b, c) in (d :: dlist, b, c)) in
    let (dlist, b, c) = List.fold_left walk ([], b, c) alist in
    (List.rev dlist, b, c)

(* Like thread2 but only threads one item instead of two *)
let thread1 : ('a * 'b -> 'c * 'b)
          -> 'a list * 'b
          -> 'c list * 'b =
  fun f (alist, b) ->
    let walk (clist, b) a =
      (let (c, b) = f (a, b) in (c :: clist, b)) in
    let (clist, b) = List.fold_left walk ([], b) alist in
```

```
    (List.rev clist, b)

(* Ensure there are no pending skips that could pass the Let *)
let assert_skip_let : 'a StringMap.t -> Cast.pos -> unit =
  fun map p ->
    if StringMap.is_empty map
      then ()
      else Utils.pos_error p "Can't skip past Let"

(*
 * Analyze the Ast.stmt, creating a Cast.stmt.
 * Thread the sinfo and env through it.
 *)
let rec sem_stmt : Ast.stmt * sinfo * env
                -> Cast.stmt * sinfo * env =
  fun (stmt, si, ev) ->
    match stmt with
      Ast.Ask(r, qs, p) ->
        let (r, t) = sem_ref ev r in
        assert_ok_type Cast.Player t (ref_pos r);
        let (qs, si) = sem_questions ev (qs, si) in
        Cast.Ask(r, qs, p), si, ev
    | Ast.Assign(r, e, p) ->
        let (r, t1) = sem_ref ev r in
        let (e, t2) = sem_expr ev e in
        assert_ok_type t1 t2 (ref_pos r);
        Cast.Assign(r, e, p), si, ev
    | Ast.Compound(r, op, e, p) ->
        let (r, t1) = sem_ref ev r in
        assert_ok_type Cast.Number t1 (ref_pos r);
        let (e, t2) = sem_expr ev e in
        assert_ok_type Cast.Number t2 (expr_pos e);
        Cast.Compound(r, op, e, p), si, ev
    | Ast.Deal(e, r1, r2, p) ->
        let (e, t) = sem_expr ev e in
        assert_type_opts [Cast.Number; Cast.Card] t (expr_pos e);
        let (r1, t1) = sem_ref ev r1 in
        assert_deal_from_type t1 (ref_pos r1);
        let (r2, t2) = sem_ref ev r2 in
        assert_deal_to_type t2 (ref_pos r2);
        (* Decide on which Deal it is *)
        if t = Cast.Card then
          (if t2 = Cast.L(Cast.Player) then
             Utils.pos_error p "Can't deal a single Card to a PlayerList";
           Cast.DealCard(e, r1, r2, p), si, ev)
        else
          Cast.DealCards(e, r1, r2, p), si, ev
    | Ast.Forever(b, p) ->
        let (b, si) = sem_block ev (b, si) in
        Cast.Forever(b, p), si, ev
    | Ast.For(id, e1, e2, b, p) ->
        let (e1, t1) = sem_expr ev e1 in
        let elt = assert_list_type t1 (expr_pos e1) in
        let (e2, t2) = sem_expr ev e2 in
        assert_ok_type elt t2 (expr_pos e2);
        (* Create a temporary local var *)
        let var = Cast.Local(si.si_locals, elt) in
        let si = si_inc_locals si in
        (* Create a new scope for the duration of the For *)
        let evtmp = env_push_scope ev in
        let evtmp = env_add_var evtmp id var p in
        let (b, si) = sem_block evtmp (b, si) in
        Cast.For(var, e1, e2, b, p), si, ev
    | Ast.If(ess, p) ->
        let sem_ess ev ((e, b), si) =
          let (e, t) = sem_expr ev e in
          assert_ok_type Cast.Boolean t (expr_pos e);
          let (b, si) = sem_block ev (b, si) in
          (e, b), si
        in
        let (ess, si) = thread1 (sem_ess ev) (ess, si) in
        Cast.If(ess, p), si, ev
    | Ast.Invoke(id, p) ->
        let act = res_action ev id p in
        Cast.Invoke(act, p), si, ev
    | Ast.Label(id, p) ->
```

```
        (* Ensure no duplicate labels *)
        assert_no_dup si.si_labels id p "Label";
        (* Remove any skips from the sinfo.*)
        let skips = if StringMap.mem id si.si_skips
                    then StringMap.remove id si.si_skips
                    else Utils.pos_error p "Unused label" in
        let labs = StringMap.add id p si.si_labels in
        let si = { si with si_skips = skips;
                           si_labels = labs; } in
        Cast.Label(id, p), si, ev
    | Ast.Let(id, e, p) ->
        (* Ensure there are no active skips *)
        assert_skip_let si.si_skips p;
        let (e, t) = sem_expr ev e in
        let var = Cast.Local(si.si_locals, t) in
        let decl = Cast.VarDecl(var, e, p) in
        let si = si_inc_locals si in
        let ev = env_add_var ev id var p in
        Cast.Let(decl, p), si, ev
    | Ast.Message(r, e, p) ->
        let (r, t1) = sem_ref ev r in
        assert_msg_type t1 p;
        let (e, t2) = sem_expr ev e in
        assert_ok_type Cast.String t2 (expr_pos e);
        Cast.Message(r, e, p), si, ev
    | Ast.Order(r, id, p) ->
        let (r, t) = sem_ref ev r in
        assert_ok_type (Cast.L(Cast.Card)) t (ref_pos r);
        let ord = res_ordering ev id p in
        Cast.Order(r, ord, p), si, ev
    | Ast.Play(id, r1, r2, p) ->
        let rule = res_rule ev id p in
        let (r1, t1) = sem_ref ev r1 in
        assert_ok_type Cast.Player t1 (ref_pos r1);
        let (r2, t2) = sem_ref ev r2 in
        assert_play_type t2 (ref_pos r2);
        Cast.Play(rule, r1, r2, p), si, ev
    | Ast.Rotate(r, p) ->
        let (r, t) = sem_ref ev r in
        ignore (assert_list_type t (ref_pos r));
        Cast.Rotate(r, p), si, ev
    | Ast.Shuffle(r, p) ->
        let (r, t) = sem_ref ev r in
        ignore (assert_list_type t (ref_pos r));
        Cast.Shuffle(r, p), si, ev
    | Ast.Skip(id, p) ->
        (* Add this skip to the sinfo *)
        let skips = StringMap.add id p si.si_skips in
        let si = { si with si_skips = skips } in
        Cast.Skip(id, p), si, ev
    | Ast.Winner(r, p) ->
        let (r, t) = sem_ref ev r in
        assert_winner_type t (ref_pos r);
        Cast.Winner(r, p), si, ev

(*
 * Analyze the Ast.question list, creating a Cast.question list.
 * Thread the sinfo through it.
 *)
and sem_questions : env -> (Ast.question list * sinfo)
                        -> (Cast.question list * sinfo) =
  fun ev qsi -> thread1 (sem_question ev) qsi

(*
 * Analyze a single Ast.question, creating a Cast.question.
 * Thread the sinfo through it.
 *)
and sem_question : env -> (Ast.question * sinfo)
                       -> (Cast.question * sinfo) =
  fun ev (q, si) ->
    match q with
      Ast.Uncond(sl, b, p) ->
        let sl = sem_str_lit ev sl in
        let (b, si) = sem_block ev (b, si) in
        Cast.Uncond(sl, b, p), si
    | Ast.Cond(sl, e, b, p) ->
```

```
        let sl = sem_str_lit ev sl in
        let (e, t) = sem_expr ev e in
        assert_ok_type Cast.Boolean t (expr_pos e);
        let (b, si) = sem_block ev (b, si) in
        Cast.Cond(sl, e, b, p), si

(*
 * Analyze the Ast.block, creating a Cast.block.
 * Thread the sinfo through it.
 *
 * A new local scope is introduced for the duration of the block.
 *
 * The pending skips on the sinfo is cleared before entering the
 * block, and any pending skips that leave the block are merged
 * into the currently pending skips as the sinfo is returned.
 *)
and sem_block : env -> (Ast.block * sinfo) -> (Cast.block * sinfo) =
  fun ev (b, si) ->
    let ev = env_push_scope ev in
    let si' = { si with si_skips = StringMap.empty } in
    let (b, si', _) = thread2 sem_stmt (b, si', ev) in
    let skips = map_merge si.si_skips si'.si_skips in
    let si' = { si' with si_skips = skips } in
    (b, si')


(*
 * Semantic analysis of declarations
 *)

(*
 * Ensure there are no unresolved skips left in the map.
 *)
let assert_no_skips : Cast.pos StringMap.t -> unit =
  fun map ->
    let err lbl pos =
      Utils.pos_error pos ("Can't find label '" ^ lbl ^ "'")
    in StringMap.iter err map

(*
 * Convert a list of Area options into a bool for facedown
 * and a bool for squaredup.
 *
 * Ensure there are no contradictions.
 *)
let rec sem_areaopts : Ast.areaopts -> bool * bool =
  fun opts ->
    let rec ensure_no item list p =
      match list with
        []     -> ()
      | h :: t ->
        if item = fst h
          then Utils.pos_error p
                 "This option contradicts earlier options"
          else ensure_no item t p
    in
    match opts with
      [] -> (true, true)
    | (Ast.Faceup, p) :: rest ->
        let (_, b) = sem_areaopts rest in
        ensure_no Ast.Facedown rest p;
        (true, b)
    | (Ast.Facedown, p) :: rest ->
        let (_, b) = sem_areaopts rest in
        ensure_no Ast.Faceup rest p;
        (false, b)
    | (Ast.Squaredup, p) :: rest ->
        let (b, _) = sem_areaopts rest in
        ensure_no Ast.Spreadout rest p;
        (b, true)
    | (Ast.Spreadout, p) :: rest ->
        let (b, _) = sem_areaopts rest in
        ensure_no Ast.Squaredup rest p;
        (b, false)

(*
```

```
 * Analyze a single global Ast.decl when given the
 * current env and current list of global variables
 * in reverse declaration order.
 *
 * This results in the new env and new list of global
 * variables in reverse declaration order.
 *)
let sem_decl : (env * Cast.var_decl list) -> Ast.decl
           -> (env * Cast.var_decl list) =
  fun (ev, vs) -> function
    Ast.Area(id, sl, opts, p) ->
      let sl = sem_str_lit ev sl in
      let (facedown, squaredup) = sem_areaopts opts in
      let var = Cast.Global(ref Cast.UndefVal, Cast.Area) in
      let decl = Cast.AreaDecl(var, sl, facedown, squaredup, p) in
      env_add_var ev id var p, decl :: vs
  | Ast.Action(id, b, p) ->
      let si = { si_locals = 0;
                 si_skips  = StringMap.empty;
                 si_labels = StringMap.empty; } in
      let (b, si) = sem_block ev (b, si) in
      assert_no_skips si.si_skips;
      let act = (si.si_locals, b, p) in
      env_add_action ev id act p, vs
  | Ast.Rule(id, args, e, p) ->
      (* Create a temporary local environment with the args in it. *)
      let name0 = List.nth args 0 in
      let name1 = List.nth args 1 in
      let name2 = List.nth args 2 in
      let arg0  = Cast.Local(0, Cast.Player) in
      let arg1  = Cast.Local(1, Cast.Card) in
      let arg2  = Cast.Local(2, Cast.L(Cast.Card)) in
      let map   = StringMap.empty in
      let map   = StringMap.add name0 arg0 map in
      let map   = StringMap.add name1 arg1 map in
      let map   = StringMap.add name2 arg2 map in
      let evtmp = { ev with e_vars = map :: ev.e_vars } in
      (* Use that temporary env for the expression only. *)
      let (e, t) = sem_expr evtmp e in
      let rule = (e, p) in
      assert_ok_type Cast.Boolean t p;
      env_add_rule ev id rule p, vs
  | Ast.Ordering(id, arg, e, p) ->
      (* Create a temporary local environment with the arg in it. *)
      let arg0  = Cast.Local(0, Cast.L(Cast.Card)) in
      let map   = StringMap.empty in
      let map   = StringMap.add arg arg0 map in
      let evtmp = { ev with e_vars = map :: ev.e_vars } in
      (* Use that temporary env for the expression only. *)
      let (e, t) = sem_expr evtmp e in
      let ord = (e, p) in
      assert_ok_type (Cast.L(Cast.Card)) t p;
      env_add_ordering ev id ord p, vs
  | Ast.Var(t, id, e, p) ->
      let t = sem_type t in
      let (e, t') = sem_expr ev e in
      let var = Cast.Global(ref Cast.UndefVal, t) in
      let decl = Cast.VarDecl(var, e, p) in
      assert_ok_type t t' p;
      env_add_var ev id var p, decl :: vs

(*
 * Analyze the list of global Ast.decls in the given env.
 *
 * The result is a list of variable declarations
 * and the main action.
 *)
let sem_decls : env -> Ast.decl list
                    -> (Cast.action * Cast.var_decl list) =
  fun ev ds ->
    let (ev, vs) = List.fold_left sem_decl (ev, []) ds in
    let main = try StringMap.find "main" ev.e_actions
               with Not_found ->
                 raise(Failure("Error: No 'main' action was declared."))
    in (main, List.rev vs)
```

```
(*
 * Semantic analysis of the game
 *)

(* Analyze the Ast.game and convert to Cast.game *)
let sem_game : Ast.game -> Cast.game =
  fun (sl, c, ds, p) ->
    (* The initial env *)
    let ev = { e_vars      = [StringMap.empty];
               e_actions   = StringMap.empty;
               e_rules     = StringMap.empty;
               e_orderings = StringMap.empty } in
    let sl = sem_str_lit ev sl in
    let (main, vs) = sem_decls ev ds in
    (sl, c, vs, main, p)
```

9.1.7 Interpreter

Listing 9.7: interp.ml

```
(*
 * This module defines the interpreter.
 *
 * This interprets the checked AST structure directly
 * by walking it. The full game is run.
 *
 *)

(* A way to communicate the winner and short circut evaluation *)
exception Winner of string

(* A skip label *)
type skipstr = string

(*
 * This represents the state of the game. It contains
 * the value of special global objects like "players"
 * and "teams" and the deck. It also contains the
 * array of local values, which is changed each
 * time an Action is entered. This is similar to an
 * activation record.
 *)
type env = { e_players : Cast.value;
             e_teams   : Cast.value;
             e_deck    : Cast.value list;
             e_locals  : Cast.value array; }


(*
 * Property Accessors
 *)

(* Get a generic property. *)
let prop_get : Cast.t -> int -> Cast.value -> Cast.value =
  fun t n v ->
    match v with
      Cast.ObjVal(o, t') ->
        if t = t' then o.(n)
                  else Utils.ie "Wrong object for property"
    | _ -> Utils.ie "Wrong object for property"

(* Get the name from a player. *)
let player_name : Cast.value -> Cast.value =
  prop_get Cast.Player 0

(* Get the hand from a player. *)
let player_hand : Cast.value -> Cast.value =
  prop_get Cast.Player 1

(* Set a new team for a player. *)
let player_set_team : Cast.value -> Cast.value -> unit =
  fun player team ->
```

```
    match player with
      Cast.ObjVal(a, Cast.Player) -> a.(4) <- team
    | _ -> Utils.ie "Setting team on non-player"

(* Get the rank of a card. *)
let card_rank : Cast.value -> Cast.value =
  prop_get Cast.Card 0

(* Get the suit of a card. *)
let card_suit : Cast.value -> Cast.value =
  prop_get Cast.Card 1

(* Get the original rank of a card. *)
let card_orig_rank : Cast.value -> Cast.value =
  prop_get Cast.Card 3

(* Get the original suit of a card. *)
let card_orig_suit : Cast.value -> Cast.value =
  prop_get Cast.Card 4

(* Set the last_played_by field of a card. *)
let card_set_played_by : Cast.value -> Cast.value -> unit =
  fun card player ->
    match card with
      Cast.ObjVal(a, Cast.Card) -> a.(2) <- player
    | _ -> Utils.ie "Setting player on non-card"

(* Get the player list from a team. *)
let team_members : Cast.value -> Cast.value =
  prop_get Cast.Team 0

(* Get the cards from an area. *)
let area_cards : Cast.value -> Cast.value =
  prop_get Cast.Area 1


(*
 * Values
 *)

(* Create a new object from the list of properties. *)
let new_obj : Cast.t -> Cast.value list -> Cast.value =
  fun t props ->
    Cast.ObjVal(Array.of_list props, t)

(* Create a new Player with the given name and defaults for the rest. *)
let new_player : string -> Cast.value =
  fun name ->
    new_obj Cast.Player [Cast.StrVal(name);
                         Cast.ListVal(ref [], Cast.Card);
                         Cast.ListVal(ref [], Cast.Card);
                         Cast.NumVal(0);
                         Cast.UndefVal]

(* Create a new Team with the given Players and defaults for the rest. *)
let new_team : Cast.value list -> Cast.value =
  fun members ->
    new_obj Cast.Team [Cast.ListVal(ref members, Cast.Player);
                       Cast.ListVal(ref [], Cast.Card);
                       Cast.NumVal(0)]

(* Create a new Card with the given Rank and Suit. *)
let new_card : Cast.rank -> Cast.suit -> Cast.value =
  fun rank suit ->
    new_obj Cast.Card [Cast.RankVal(rank);
                       Cast.SuitVal(suit);
                       Cast.UndefVal;
      (* Orig rank *)  Cast.RankVal(rank);
      (* Orig suit *)  Cast.SuitVal(suit)]

(* Return the numeric equivalent of a rank. *)
let rank_num : Cast.rank -> int =
  function
    Ast.Ace      -> 1
  | Ast.King     -> 13
  | Ast.Queen    -> 12
```

```
  | Ast.Jack     -> 11
  | Ast.Minor(n) -> n

(* Return the rank equivalent of a number. *)
let num_rank : Cast.pos -> int -> Cast.rank =
  fun p -> function
    1  -> Ast.Ace
  | 11 -> Ast.Jack
  | 12 -> Ast.Queen
  | 13 -> Ast.King
  | n  -> if n >= 2 && n <= 10 then Ast.Minor(n)
            else Utils.pos_error p
                  ("The value " ^ (string_of_int n)
                                 ^ " can't be a Rank")

(* Return a numeric value for a suit. *)
let suit_num : Cast.suit -> int =
  function
    Ast.Clubs -> 1
  | Ast.Hearts -> 2
  | Ast.Spades -> 3
  | Ast.Diamonds -> 4

(* Get a native string from a Cast.StrVal. *)
let as_str : Cast.value -> string =
  function
    Cast.StrVal(s) -> s
  | _ -> Utils.ie "Not a string value"

(* Get a native bool from a Cast.BoolVal. *)
let as_bool : Cast.value -> bool =
  function
    Cast.BoolVal(b) -> b
  | _ -> Utils.ie "Not a boolean value"

(* Get a native int from a Cast.NumVal. *)
let as_num : Cast.value -> int =
  function
    Cast.NumVal(n) -> n
  | Cast.RankVal(r) -> rank_num r
  | _ -> Utils.ie "Not a numeric value"

(* Get a native Ast.rank from a Cast.SuitVal. *)
let as_rank : Cast.pos -> Cast.value -> Ast.rank =
  fun p -> function
    Cast.NumVal(n) -> num_rank p n
  | Cast.RankVal(r) -> r
  | _ -> Utils.ie "Not a rank value"

(* Get a native Ast.suit from a Cast.SuitVal. *)
let as_suit : Cast.value -> Ast.suit =
  function
    Cast.SuitVal(s) -> s
  | _ -> Utils.ie "Not a suit value"

(* Get a list of values from a Cast.ListVal. *)
let as_list : Cast.value -> Cast.value list =
  function
    Cast.ListVal(vs, _) -> !vs
  | _ -> Utils.ie "Not a list value"

(* Convert the rank to a string *)
let string_of_rank : Cast.rank -> string =
  function
    Ast.Ace       -> "Ace"
  | Ast.King      -> "King"
  | Ast.Queen     -> "Queen"
  | Ast.Jack      -> "Jack"
  | Ast.Minor(10) -> "Ten"
  | Ast.Minor(9)  -> "Nine"
  | Ast.Minor(8)  -> "Eight"
  | Ast.Minor(7)  -> "Seven"
  | Ast.Minor(6)  -> "Six"
  | Ast.Minor(5)  -> "Five"
  | Ast.Minor(4)  -> "Four"
  | Ast.Minor(3)  -> "Three"
```

```
  | Ast.Minor(2)  -> "Two"
  | _ -> Utils.ie "Invalid rank for string conversion"

(* Convert the suit to a string *)
let string_of_suit : Cast.suit -> string =
  function
    Ast.Clubs     -> "Clubs"
  | Ast.Hearts    -> "Hearts"
  | Ast.Spades    -> "Spades"
  | Ast.Diamonds -> "Diamonds"

(* Convert the value to a string. *)
let rec to_str : Cast.value -> string =
  fun value ->
    match value with
      Cast.BoolVal(b) -> if b then "True" else "False"
    | Cast.NumVal(n)  -> string_of_int n
    | Cast.RankVal(r) -> string_of_rank r
    | Cast.StrVal(s)  -> s
    | Cast.SuitVal(s) -> string_of_suit s
    | Cast.ObjVal(_, Cast.Card) ->
        let r = to_str (card_orig_rank value) in
        let s = to_str (card_orig_suit value) in
        r ^ " of " ^ s
    | Cast.ObjVal(_, Cast.Player) ->
        as_str (player_name value)
    | Cast.ObjVal(_, Cast.Team) ->
        let ps = team_members value in
        let ps = as_list ps in
        let ns = List.map player_name ps in
        "Team (" ^ (Utils.catmap ", " to_str ns) ^ ")"
    | _ -> Utils.ie "String conversion of invalid type"

(* Get a CardList from a Player, Area, or CardList *)
let to_cardlist : Cast.value -> Cast.value =
  fun v ->
    match v with
      Cast.ObjVal(_, Cast.Player) -> player_hand v
    | Cast.ObjVal(_, Cast.Area) -> area_cards v
    | Cast.ListVal(_, Cast.Card) -> v
    | Cast.ListVal(_, Cast.EmptyType) -> v
    | _ -> Utils.ie "Not a cardlist value"

(* Get a CardList List from a Player, Area, CardList, or PlayerList *)
let to_cardlistlist : Cast.value -> Cast.value list =
  fun v ->
    match v with
      Cast.ObjVal(_, Cast.Player)  -> [player_hand v]
    | Cast.ObjVal(_, Cast.Area)    -> [area_cards v]
    | Cast.ListVal(_, Cast.Card)   -> [v]
    | Cast.ListVal(_, Cast.Player) ->
        let players = as_list v in
        List.map player_hand players
    | Cast.ListVal(_, Cast.EmptyType) -> [v]
    | _ -> Utils.ie "Not a cardlistlist value"

(*
 * Get a Cast.value list from either a scalar Cast.value or
 * a list Cast.value.
 *)
let elt_or_list : Cast.value -> Cast.value list =
  function
    Cast.ListVal(l, _) -> !l
  | v -> [v]

(* Get the length of the list. *)
let list_length : 'a list -> Cast.value =
  fun l -> Cast.NumVal(List.length l)

(* Get the first item from the non-empty list. *)
let list_first : 'a list -> Cast.pos -> 'a =
  fun l p ->
    match l with
      [] -> Utils.pos_error p "Can't get item from empty list"
    | h :: _ -> h
```

```
(* Get the last item from the non-empty list. *)
let rec list_last : 'a list -> Cast.pos -> 'a =
  fun l p ->
    match l with
      [] -> Utils.pos_error p "Can't get item from empty list"
    | h :: [] -> h
    | _ :: t -> list_last t p

(* Compare two values for structural equality. *)
let rec val_eq : Cast.value -> Cast.value -> bool =
  fun v1 v2 ->
    match (v1, v2) with
      Cast.BoolVal(b1),    Cast.BoolVal(b2)   -> b1 = b2
    | Cast.NumVal(n1),     Cast.NumVal(n2)    -> n1 = n2
    | Cast.NumVal(n),      Cast.RankVal(r)    -> n = (rank_num r)
    | Cast.RankVal(r),     Cast.NumVal(n)     -> (rank_num r) = n
    | Cast.RankVal(r1),    Cast.RankVal(r2)   -> r1 = r2
    | Cast.StrVal(s1),     Cast.StrVal(s2)    -> s1 = s2
    | Cast.SuitVal(s1),    Cast.SuitVal(s2)   -> s1 = s2
    | Cast.ObjVal(o1, _),  Cast.ObjVal(o2, _) -> o1 == o2
    | Cast.ListVal(v1, _), Cast.ListVal(v2, _) ->
        let l1 = List.length !v1 in
        let l2 = List.length !v2 in
        (l1 = l2) && (List.for_all2 val_eq !v1 !v2)
    | Cast.UndefVal, Cast.UndefVal -> true
    | _ -> false

(*
 * Compare two values for 'a in b' equality. This comparison is done
 * between 'a' and an element of 'b'. This is like 'val_eq' but also
 * supports 'rank in card' and 'suit in card' comparisons.
 *)
let rec val_eq_in : Cast.pos -> Cast.value -> Cast.value -> bool =
  fun p v1 v2 ->
    match (v1, v2) with
      Cast.RankVal(r), Cast.ObjVal(o, Cast.Card) ->
        r = (as_rank p (card_rank v2))
    | Cast.NumVal(n),  Cast.ObjVal(o, Cast.Card) ->
        n = (as_num (card_rank v2))
    | Cast.SuitVal(s), Cast.ObjVal(o, Cast.Card) ->
        s = (as_suit (card_suit v2))
    | _ -> val_eq v1 v2

(*
 * Variables and Properties
 *)

(* Get the value from a variable. *)
let run_var : env -> Cast.variable -> Cast.value =
  fun env -> function
    Cast.Global(r, _) -> !r
  | Cast.Local(h, _)  -> env.e_locals.(h)
  | Cast.PlayersVar   -> env.e_players
  | Cast.TeamsVar     -> env.e_teams
  | Cast.StandardVar  -> Cast.ListVal(ref env.e_deck, Cast.Card)

(* Get the value from an object's property. *)
let run_prop : Cast.value -> Cast.prop -> Cast.pos -> Cast.value =
  fun v prop p ->
    match (v, prop) with
      Cast.ObjVal(v, _), Cast.ObjProp(n, _) -> v.(n)
    | Cast.ListVal(v, _), Cast.ListProp(lp, _) ->
        (match lp with
           Cast.Size   -> list_length !v
         | Cast.First  -> list_first !v p
         | Cast.Last   -> list_last !v p
         | Cast.Top    -> list_first !v p
         | Cast.Bottom -> list_last !v p)
    | _ -> Utils.ie "Invalid object property"

(* Store the value in the variable. *)
let set_var : env -> Cast.variable -> Cast.value -> Cast.pos -> unit =
  fun env var value p ->
    match var with
      Cast.Global(r, _) -> r := value
    | Cast.Local(h, _)  -> env.e_locals.(h) <- value
```

```
    | _ -> Utils.pos_error p "Can't assign to a read-only variable"

(* Set the value of an object's property. *)
let set_prop : Cast.value -> Cast.prop -> Cast.value -> Cast.pos -> unit =
  fun obj prop value p ->
    match (obj, prop) with
      Cast.ObjVal(v, _), Cast.ObjProp(n, _) -> v.(n) <- value
    | Cast.ListVal(v, _), Cast.ListProp(lp, _) ->
        Utils.pos_error p "Can't assign read-only list properties"
    | _ -> Utils.ie "Invalid object property"

(* Set the value in the mutable list. *)
let set_list : Cast.value -> Cast.value list -> unit =
  fun list value ->
    match list with
      Cast.ListVal(v, _) -> v := value
    | _ -> Utils.ie "Setting non-list"


(*
 * The Deck
 *)

(* Return the unique index in a 52-card deck for this rank/suit combination *)
let card_idx : Cast.rank -> Cast.suit -> int =
  fun rank suit ->
    let n1 = rank_num rank in
    let n2 = suit_num suit in
    (n2 - 1) * 13 + (n1 - 1)

(* Create a new 52-card deck as a list of Card Cast.values. *)
let new_deck : unit -> Cast.value list =
  fun _ ->
    let cards = Array.make 52 Cast.UndefVal in
    let f rank suit =
      let idx = card_idx rank suit in
      cards.(idx) <- new_card rank suit
    in
    let ranks = List.map (num_rank (-1, -1)) (Utils.int_range 1 13) in
    let suits = [Ast.Clubs; Ast.Hearts; Ast.Spades; Ast.Diamonds] in
    ignore (Utils.xmap f ranks suits);
    Array.to_list cards

(* Given a rank and a suit, return the named card from the environment. *)
let get_card : env -> Cast.pos -> Cast.value -> Cast.value -> Cast.value =
  fun env p rank suit ->
    let idx = card_idx (as_rank p rank) (as_suit suit) in
    List.nth env.e_deck idx


(*
 * Expressions and References
 *)

(* Combine two values using a binary operator. *)
let rec run_binary : Cast.value -> Cast.op -> Cast.value
                  -> Cast.pos -> Cast.value =
  fun v1 op v2 p ->
    match op with
      Ast.Plus   -> Cast.NumVal((as_num v1) + (as_num v2))
    | Ast.Minus  -> Cast.NumVal((as_num v1) - (as_num v2))
    | Ast.Times  -> Cast.NumVal((as_num v1) * (as_num v2))
    | Ast.Divide ->
        let n1 = as_num v1 in
        let n2 = as_num v2 in
        if n2 = 0 then Utils.pos_error p "Division by zero"
                  else Cast.NumVal(n1 / n2)
    | Ast.Eq     -> Cast.BoolVal(val_eq v1 v2)
    | Ast.NotEq  -> Cast.BoolVal(not(val_eq v1 v2))
    | Ast.Lt     -> Cast.BoolVal((as_num v1) <  (as_num v2))
    | Ast.LtEq   -> Cast.BoolVal((as_num v1) <= (as_num v2))
    | Ast.Gt     -> Cast.BoolVal((as_num v1) >  (as_num v2))
    | Ast.GtEq   -> Cast.BoolVal((as_num v1) >= (as_num v2))
    | Ast.In     ->
        let l2 = as_list v2 in
        let b = List.exists (val_eq_in p v1) l2 in
```

```
        Cast.BoolVal(b)

    (* And and Or are done in run_binary_short *)
    | _ -> Utils.ie "Unexpected binary op"

(* Like run_binary but does short-circuit operators first *)
and run_binary_short : env -> Cast.expr -> Cast.op -> Cast.expr
                    -> Cast.pos -> Cast.value =
  fun env e1 op e2 p ->
    match op with
      Ast.And ->
        let v1 = run_expr env e1 in
        let b1 = as_bool v1 in
        if not b1 then
          Cast.BoolVal(false)
        else
          let v2 = run_expr env e2 in
          let b2 = as_bool v2 in
          Cast.BoolVal(b2)
    | Ast.Or ->
        let v1 = run_expr env e1 in
        let b1 = as_bool v1 in
        if b1 then
          Cast.BoolVal(true)
        else
          let v2 = run_expr env e2 in
          let b2 = as_bool v2 in
          Cast.BoolVal(b2)
    | _ ->
      let v1 = run_expr env e1 in
      let v2 = run_expr env e2 in
      run_binary v1 op v2 p

(* Run a string literal, producing a string value. *)
and run_str_lit : env -> Cast.str_lit -> Cast.value =
  fun env sl ->
    let part_to_str = function
        Cast.StrRef(r) -> to_str (run_ref env r)
      | Cast.StrLit(s) -> s
    in
    let str = Utils.catmap "" part_to_str sl in
    Cast.StrVal(str)

(* Run a reference, producing a value. *)
and run_ref : env -> Cast.reference -> Cast.value =
  fun env -> function
    Cast.Var(v, _) ->
      run_var env v
  | Cast.Prop(e, prop, p) ->
      let obj = run_expr env e in
      run_prop obj prop p
  | Cast.EmptyRef ->
      Utils.ie "Executing EmptyRef"

(* Store the value in the reference. *)
and set_ref : env -> Cast.reference -> Cast.value -> unit =
  fun env ref value ->
    match ref with
      Cast.Var(v, p) ->
        set_var env v value p
    | Cast.Prop(e, prop, p) ->
        let obj = run_expr env e in
        set_prop obj prop value p
    | Cast.EmptyRef ->
        Utils.ie "Setting EmptyRef"

(* Run a variable declaration, initializing the variable. *)
and run_var_decl : env -> Cast.var_decl -> unit =
  fun env -> function
    Cast.VarDecl(v, e, p) ->
      let value = if e = Cast.EmptyExpr
                  then Cast.UndefVal
                  else run_expr env e
      in set_var env v value p
  | Cast.AreaDecl(v, sl, facedown, squaredup, p) ->
      let name = run_str_lit env sl in
```

```
      let cards = Cast.ListVal(ref [], Cast.Card) in
      let facedown = Cast.BoolVal(facedown) in
      let squaredup = Cast.BoolVal(squaredup) in
      let value = new_obj Cast.Area [name; cards; facedown; squaredup] in
      set_var env v value p

(* Run an expression, producing a value. *)
and run_expr : env -> Cast.expr -> Cast.value =
  fun env -> function
    Cast.Not(e, _) ->
      let b = as_bool (run_expr env e) in
      Cast.BoolVal(not b)
  | Cast.Binary(e1, op, e2, p) ->
      run_binary_short env e1 op e2 p
  | Cast.Defined(r, _) ->
      let v = run_ref env r in
      let b = if v = Cast.UndefVal then false else true in
      Cast.BoolVal(b)
  | Cast.CanPlay(rule, r1, r2, _) ->
      let player = run_ref env r1 in
      let dest = run_ref env r2 in
      let cl = run_rule env rule player dest in
      let b = (List.length cl) <> 0 in
      Cast.BoolVal(b)
  | Cast.CardExpr(e1, p, e2) ->
      let v1 = run_expr env e1 in
      let v2 = run_expr env e2 in
      let ranks = elt_or_list v1 in
      let suits = elt_or_list v2 in
      let cards = Utils.xmap (get_card env p) ranks suits in
      (match cards with
         h :: [] -> h
       | _ -> Cast.ListVal(ref cards, Cast.Card))
  | Cast.RangeExpr(e1, e2, p) ->
      let v1 = run_expr env e1 in
      let v2 = run_expr env e2 in
      let n1 = as_num v1 in
      let n2 = as_num v2 in
      let range = Utils.int_range n1 n2 in
      let f n = Cast.RankVal(num_rank p n) in
      let vs = List.map f range in
      Cast.ListVal(ref vs, Cast.Rank)
  | Cast.RankSeqExpr(e1, e2, _) ->
      let v1 = run_expr env e1 in
      let v2 = run_expr env e2 in
      let l1 = elt_or_list v1 in
      let l2 = elt_or_list v2 in
      let all = l1 @ l2 in
      Cast.ListVal(ref all, Cast.Rank)
  | Cast.SuitSeqExpr(e1, e2, _) ->
      let v1 = run_expr env e1 in
      let v2 = run_expr env e2 in
      let l1 = elt_or_list v1 in
      let l2 = elt_or_list v2 in
      let all = l1 @ l2 in
      Cast.ListVal(ref all, Cast.Suit)
  | Cast.WildRank(_, _, e) ->
      let v = run_expr env e in
      let suits = elt_or_list v in
      let suits = List.map as_suit suits in
      let f card =
        let suit = as_suit (card_suit card) in
        List.mem suit suits
      in
      let matches = List.filter f env.e_deck in
      Cast.ListVal(ref matches, Cast.Card)
  | Cast.WildSuit(e, p, _) ->
      let v = run_expr env e in
      let ranks = elt_or_list v in
      let ranks = List.map (as_rank p) ranks in
      let f card =
        let rank = as_rank p (card_rank card) in
        List.mem rank ranks
      in
      let matches = List.filter f env.e_deck in
      Cast.ListVal(ref matches, Cast.Card)
```

```
  | Cast.WildCard(_, _, _) ->
      Cast.ListVal(ref env.e_deck, Cast.Card)
  | Cast.List(es, t, _) ->
      let vals = List.map (run_expr env) es in
      (* Some of vals are elements and some are lists *)
      let vals = List.map elt_or_list vals in
      let vals = List.concat vals in
      Cast.ListVal(ref vals, t)
  | Cast.Ref(r, p) ->
      let v = run_ref env r in
      if v = Cast.UndefVal then
        Utils.pos_error p "Undefined value";
      v
  | Cast.Literal(v, _) -> v
  | Cast.StrLiteral(sl, _) -> run_str_lit env sl
  | Cast.EmptyExpr -> Utils.ie "Executing EmptyExpr"

(* Return the cards that can be played using a rule. *)
and run_rule : env -> Cast.rule -> Cast.value -> Cast.value
            -> Cast.value list =
  fun env (e, _) player dest ->
    let hand   = player_hand player in
    let cards  = as_list hand in
    let dest   = to_cardlist dest in
    let f card =
      let locals = Array.of_list [player; card; dest] in
      let env = { env with e_locals = locals } in
      let v = run_expr env e in
      as_bool v
    in
    List.filter f cards

(* Return the Cast.value list of an ordering *)
and run_ordering : env -> Cast.ordering -> Cast.value -> Cast.value list =
  fun env (e, _) cards ->
    let locals = Array.of_list [cards] in
    let env = { env with e_locals = locals } in
    let ord = run_expr env e in
    as_list ord


(*
 * Statements
 *)

(*
 * Move a card from one CardList to another CardList.
 *)
let move_card : Cast.value -> Cast.value -> Cast.value -> unit =
  fun card src dst ->
    (* Remove the card from src *)
    let cl1 = as_list src in
    let cl1 = Utils.lremove val_eq card cl1 in
    set_list src cl1;
    (* Store the card in dst *)
    let cl2 = as_list dst in
    let cl2 = card :: cl2 in
    set_list dst cl2

(*
 * Convert a list of questions to a list of string/block pairs.
 * Only include questions where the predicate is true, if there
 * is a predicate.
 *)
let run_questions : env -> Cast.question list -> (string * Cast.block) list =
  fun env qs ->
    let f res = function
        Cast.Uncond(sl, block, _) ->
          let s = as_str (run_str_lit env sl) in
          (s, block) :: res
      | Cast.Cond(sl, e, block, _) ->
          let s = as_str (run_str_lit env sl) in
          let b = as_bool (run_expr env e) in
          if b then (s, block) :: res else res
    in
    let qs = List.fold_left f [] qs in
```

```
    List.rev qs

(*
 * Run a single statement, threading the skip string.
 *
 * If the skip string is the empty string, then the
 * statement is run as normal. A new skip string will
 * be produced if this is a "skip" statement - otherwise,
 * the empty skip string will be the result.
 *
 * If the skip string is not the empty string, a skip
 * has previously been executed, and this statement is
 * not run unless it is the matching label.
 *)
let rec run_stmt : env -> skipstr -> Cast.stmt -> skipstr =
  fun env skip stmt ->
    match (skip, stmt) with
      "", Cast.Ask(r, qs, _) ->
          let player = run_ref env r in
          let name = as_str (player_name player) in
          let choices = run_questions env qs in
          let msg = name ^ ": Choose an action:" in
          let block = Ui.lchoice msg choices in
          run_block env block
    | "", Cast.Assign(r, e, p) ->
          let v = run_expr env e in
          set_ref env r v; ""
    | "", Cast.Compound(r, op, e, p) ->
          let v1 = run_ref env r in
          let v2 = run_expr env e in
          let v = run_binary v1 op v2 p in
          set_ref env r v; ""
    | "", Cast.DealCard(e, r1, r2, _) ->
          let card = run_expr env e in
          let v1 = run_ref env r1 in
          let cl1 = to_cardlist v1 in
          let v2 = run_ref env r2 in
          let cl2 = to_cardlist v2 in
          move_card card cl1 cl2; ""
    | "", Cast.DealCards(e, r1, r2, _) ->
          let v1 = run_ref env r1 in
          let cl1 = to_cardlist v1 in
          let v2 = run_ref env r2 in
          let cl2 = to_cardlistlist v2 in
          let len = List.length cl2 in
          let n = if e = Cast.EmptyExpr
                    then List.length (as_list cl1)
                    else len * (as_num (run_expr env e))
          in
          (* deal cards one at a time to each cardlist in cl2 *)
          for i = 1 to n do
            let cards = as_list cl1 in
            let card = List.hd cards in
            let dest = List.nth cl2 (i mod len) in
            move_card card cl1 dest
          done; ""
    | "", Cast.Forever(block, _) ->
          let new_skip = run_block env block in
          run_stmt env new_skip stmt
    | "", Cast.For(v, e1, e2, block, p) ->
          let v1 = run_expr env e1 in
          let v1 = as_list v1 in
          (* Permute the list so e2 is first *)
          let permute l e =
            if e = Cast.EmptyExpr then l else
              let v = run_expr env e in
              try
                let (front, back) = Utils.split val_eq v l in
                back @ front
              with Not_found ->
                Utils.pos_error p "Starting element is not in list"
          in
          let v1 = permute v1 e2 in
          let f skipstr value =
            if skipstr <> "" then skipstr else
              (set_var env v value p;
```

```
              run_block env block)
          in
          List.fold_left f "" v1
    | "", Cast.If(ess, _) ->
          let f (expr, _) =
            expr = Cast.EmptyExpr || as_bool (run_expr env expr)
          in
          (try
            let (_, block) = List.find f ess in
            run_block env block
          with Not_found -> "")
    | "", Cast.Invoke(a, _) ->
          run_action env a; ""
    | "", Cast.Label(_, _) -> ""
    | s,  Cast.Label(t, _) ->
          if s = t then "" else s
    | "", Cast.Let(v, _) ->
          run_var_decl env v; ""
    | "", Cast.Message(r1, e, _) ->
          let s = as_str (run_expr env e) in
          if r1 = Cast.EmptyRef then
            Ui.show_all s
          else
            (let p = player_name (run_ref env r1) in
             let p = as_str p in
             Ui.show p s);
          ""
    | "", Cast.Order(r, o, _) ->
          let cards = run_ref env r in
          let ord = run_ordering env o cards in
          let cl = as_list cards in
          let cl = Utils.order val_eq cl ord in
          set_list cards cl; ""
    | "", Cast.Play(rule, r1, r2, p) ->
          let player = run_ref env r1 in
          let dest = to_cardlist (run_ref env r2) in
          let cl = run_rule env rule player dest in
          if cl = [] then
            Utils.pos_error p "No cards match the given rule";
          (* Ask the user which card to play *)
          let name = as_str (player_name player) in
          let pairs = List.map (fun c -> (to_str c, c)) cl in
          let msg = name ^ ": Choose a card:" in
          let card = Ui.lchoice msg pairs in
          (* Move the card from the player's hand to the destination *)
          let hand = player_hand player in
          move_card card hand dest;
          card_set_played_by card player;
          let msg = name ^ " played " ^ (to_str card) ^ "." in
          Ui.show_all msg;
          ""
    | "", Cast.Rotate(r, _) ->
          let list = run_ref env r in
          let v = as_list list in
          let v = match v with
              [] -> v
            | [_] -> v
            | h::t -> t @ [h]
          in
          set_list list v; ""
    | "", Cast.Shuffle(r, _) ->
          let list = run_ref env r in
          let v = as_list list in
          let v = Utils.shuffle v in
          set_list list v; ""
    | "", Cast.Skip(s, _) -> s
    | "", Cast.Winner(r, _) ->
          let w = run_ref env r in
          let s = to_str w in
          raise(Winner(s))
    | _ -> skip

(* Run a block by running all of its statements. *)
and run_block : env -> Cast.block -> skipstr =
  fun env sl -> List.fold_left (run_stmt env) "" sl
```

```
(*
 * Run an action.
 *
 * This sets up a new "activation record" with storage for
 * the local variables in the action, and then runs
 * the block.
 *)
and run_action : env -> Cast.action -> unit =
  fun env (num, block, _) ->
    let env = { env with e_locals = Array.make num Cast.UndefVal; }
    in ignore (run_block env block)


(*
 * The Game
 *)

(*
 * Initialize the players by asking the user for
 * the number of players and the names of each.
 *)
let init_players : Cast.ptcount -> env =
  function
    Ast.PlayerCount(il, _) ->
      let np = Ui.ichoice "How many players are playing?" il in
      let msg = "This game has " ^ (string_of_int np) ^ " players." in
      Ui.show_all msg;
      let players = Array.make np Cast.UndefVal in
      for i = 1 to np do
        let msg = "Enter a name for Player " ^ (string_of_int i) ^ ":" in
        let name = Ui.istring msg in
        players.(i - 1) <- new_player name
      done;
      let players = Array.to_list players in
      { e_players = Cast.ListVal(ref players, Cast.Player);
        e_teams   = Cast.UndefVal;
        e_deck    = new_deck ();
        e_locals  = Array.make 0 Cast.UndefVal; }
  | Ast.TeamCount(il, np, _) ->
      let msg = "This game has " ^ (string_of_int np) ^ " players per team." in
      Ui.show_all msg;
      let nt = Ui.ichoice "How many teams are playing?" il in
      let msg = "This game has " ^ (string_of_int nt) ^ " teams." in
      Ui.show_all msg;
      let all_players = Array.make (np * nt) Cast.UndefVal in
      let teams = Array.make nt Cast.UndefVal in
      for ti = 1 to nt do
        let players = Array.make np Cast.UndefVal in
        for pi = 1 to np do
          let msg = "Enter a name for Player "
                      ^ (string_of_int pi) ^ " on team "
                      ^ (string_of_int ti) ^ ":" in
          let name = Ui.istring msg in
          let idx = (pi - 1) * nt + (ti - 1) in
          let player = new_player name in
          players.(pi - 1) <- player;
          all_players.(idx) <- player
        done;
        let team = new_team (Array.to_list players) in
        (* Update each player->team to point to this team *)
        Array.iter (fun p -> player_set_team p team) players;
        teams.(ti - 1) <- team
      done;
      let all_players = Array.to_list all_players in
      let teams = Array.to_list teams in
      { e_players = Cast.ListVal(ref all_players, Cast.Player);
        e_teams   = Cast.ListVal(ref teams, Cast.Team);
        e_deck    = new_deck ();
        e_locals  = Array.make 0 Cast.UndefVal; }

(*
 * Initialize each global variable by running
 * the initializer expression and storing the
 * value into the global.
 *)
let init_globals : env -> Cast.var_decl list -> unit =
```

```
  fun env vars -> List.iter (run_var_decl env) vars

(*
 * Run the game.
 *
 * This initializes the game, runs the main action, and
 * declares a winner.
 *)
let run_game : Cast.game -> unit =
  fun (name, count, vars, main, p) ->
    let env = { e_players = Cast.UndefVal;
                e_teams   = Cast.UndefVal;
                e_deck    = [];
                e_locals  = Array.make 0 Cast.UndefVal; } in
    let name = run_str_lit env name in
    let name = as_str name in
    Ui.show_all "------------------------------------------------";
    Ui.show_all ("Welcome to " ^ name);
    Ui.show_all "------------------------------------------------";
    let env = init_players count in
    init_globals env vars;
    try run_action env main;
        Ui.show_all "No winner has been declared."
    with Winner(s) -> print_endline ("The game was won by " ^ s)
```

9.1.8 User Interface

Listing 9.8: ui.ml

```
(*
 * This module deals with user input and output during interpretation.
 *)

(*
 * Get a string from the user.
 *)
let istring : string -> string =
  fun msg -> print_string msg; print_string " "; read_line ()

(*
 * Let the given player choose a string from a list of strings.
 * Return the paired item corresponding to the chosen string.
 *)
let rec lchoice : string -> (string * 'a) list -> 'a =
  fun msg -> function
    [] -> Utils.ie "Nothing to choose from"
  | choices ->
      print_endline msg;
      let show (s, _) = print_string "  "; print_endline s
      in List.iter show choices;
      let choice = istring "Your choice?" in
      let f (s, _) = choice = s in
      try snd (List.find f choices)
      with Not_found ->
        (print_endline "I'm sorry, I didn't understand.";
         lchoice msg choices)

(* Offer a choice of integers. *)
let ichoice : string -> int list -> int =
  fun msg il ->
    let f i = (string_of_int i, i) in
    let choices = List.map f il in
    lchoice msg choices

(* Show a message to all users *)
let show_all : string -> unit =
  print_endline

(* Show a message to the named player *)
let show : string -> string -> unit =
  fun player msg ->
    print_endline (player ^ ": " ^ msg)
```

9.1.9 General Utilities

Listing 9.9: utils.ml

```
(*
 * This module defines various generic utilities.
 *)

(**************)
(* List Utils *)

(* Global random initialization *)
Random.self_init ()

(* Randomly shuffle a list using Fisher-Yates. *)
let shuffle : 'a list -> 'a list =
  fun l ->
    let s = List.length l in
    let a = Array.of_list l in
    for n = s downto 2 do
      let r = Random.int n in
      let t = a.(n - 1) in
      a.(n - 1) <- a.(r);
      a.(r) <- t
    done;
    Array.to_list a

(*
 * Split a list at the first occurrence of the given element.
 *
 * The first item in the resulting pair will contain all elements
 * before the given one is found, and the second item in the pair
 * will include the item and everything after it.
 *
 * Raises 'Not_found' if the item is not in the list.
 *
 * Equality is performed by the given function.
 *)
let split : ('a -> 'a -> bool) -> 'a -> 'a list -> ('a list * 'a list) =
  fun f e l ->
    let rec helper l accum =
      match l with
          [] -> raise(Not_found)
        | h :: t -> if f h e then (List.rev accum, l)
                                else helper t (h :: accum)
    in helper l []

(*
 * Remove the first occurrence of the given element from the list.
 * Use the given function to test equality.
 *)
let rec lremove : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list =
  fun f e -> function
    [] -> []
  | h :: t -> if f h e then t else h :: (lremove f e t)

(*
 * Return the 0-based index of the element in the list, using
 * the given function to test equality. If the element is not
 * found, return the default value.
 *)
let lindex : ('a -> 'a -> bool) -> 'a list -> 'a -> int -> int =
  fun fn l e def ->
    let rec helper l ind =
      match l with
        []   -> def
      | h::t ->
          if fn h e then ind else helper t (ind + 1)
    in helper l 0

(*
 * Order the elements of the first list so that they are
 * in the same relative ordering as the same elements in
 * the second list.
 *
 * Use the given function to test equality.
```

```
 *)
let order : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list =
  fun f l1 l2 ->
    let fn e1 e2 =
      (* Return positive if e1 should go after e2 in the result,
         negative if the reverse, and 0 if it doesn't matter *)
      let max = List.length l2 in
      let i1 = lindex f l2 e1 max in
      let i2 = lindex f l2 e2 max in
      i1 - i2
    in
    List.sort fn l1

(*
 * Call the function once for every combination of
 * elements from 'a and 'b. This is the Cartesian Product.
 * Returns a list of the results.
 *)
let xmap : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list =
  fun f al bl ->
    let rec helper al' bl' cl' =
      match (al', bl') with
        ([], _) -> cl'
      | (h::t, []) -> helper t bl cl'
      | (h::_, h'::t') -> let c = f h h' in
                          helper al' t' (c::cl')
    in List.rev (helper al bl [])


(***************)
(* String Utils *)

(* char list to string *)
let str_implode : char list -> string =
  fun clist ->
    let slist = List.map (fun c -> String.make 1 c) clist in
    String.concat "" slist

(* string to char list *)
let str_explode : string -> char list =
  fun s ->
    let rec helper s n l =
      if n == l then []
                else s.[n] :: helper s (n + 1) l
    in helper s 0 (String.length s)

(* Map a function across the chars in a string and concatenate the results *)
let str_map : (char -> string) -> string -> string =
  fun f s ->
    let clist = str_explode s in
    let slist = List.map f clist in
    String.concat "" slist

(* Map a function over a list and concatenate the resulting strings *)
let catmap : string -> ('a -> string) -> 'a list -> string =
  fun s f alist -> String.concat s (List.map f alist)


(************)
(* Int utils *)

(* Return a list range including the end points *)
let rec int_range : int -> int -> int list =
  fun low high ->
    if low = high then [low]
    else if low > high then List.rev (int_range high low)
    else low :: (int_range (low + 1) high)


(**************)
(* Error utils *)

(* Raise a Failure exception with line/column information *)
let pos_error : int * int -> string -> 'a =
  fun pos str ->
    let prefix = "Error: Line " ^ (string_of_int (fst pos)) ^
```

```
                        ", Col " ^ (string_of_int (snd pos)) ^
                        ": "
    in raise(Failure(prefix ^ str))

(* Raise an internal error *)
let ie : string -> 'a =
  fun s -> raise(Failure("Internal Error: " ^ s))


(****************)
(* Line Numbers *)

(* Track line numbers when scanning and parsing. *)
let line_num = ref 1

(* Return the (line, col) of the Lexing position *)
let lexpos : Lexing.position -> int * int =
  fun pos ->
    let line = pos.Lexing.pos_lnum
    and col  = pos.Lexing.pos_cnum - pos.Lexing.pos_bol
    in (line, col + 1)
```

9.2 Unit Tests

Testcases come in pairs - for every file “X” there is a second file “X.log” that contains the expected output.

Note: Some of the logs are empty intentionally - the test case is not meant to produce any output in those cases.

9.2.1 Syntactic Tests

The syntactic tests are run through “pip –print-ast”.

Listing 9.10: decl_others.pip

```
Game "foo" requires 2 players.

Area a labeled "bcd".
Area a2 labeled "def" is faceup, spreadout.

Rule r(a, b, c) = True.

Ordering o(a) = [].
```

Listing 9.11: decl_others.pip.log

```
Game "foo" requires 2 players.
Area a labeled "bcd".
Area a2 labeled "def" is faceup, spreadout.
Rule r(a, b, c) = True.
Ordering o(a) = [].
```

Listing 9.12: decl_var.pip

```
Game "Foo" requires 2 players.

Boolean b.
Card c.
CardList cl.
Deck d.
Number n.
Player p.
PlayerList pl.
Rank r.
RankList rl.
String str.
Suit s.
SuitList sl.
Team t.
TeamList tl.
```

Listing 9.13: decl_var.pip.log

```
Game "Foo" requires 2 players.
Boolean b.
Card c.
CardList cl.
Deck d.
Number n.
Player p.
PlayerList pl.
Rank r.
RankList rl.
String str.
Suit s.
SuitList sl.
Team t.
TeamList tl.
```

Listing 9.14: decl_var_init.pip

```
Game "Foo" requires 2 players.

Boolean b = False.
Card c = J~C.
CardList cl = [c].
Deck d = cl.
Number n = 42.
Player p = players->first.
PlayerList pl = players.
Rank r = A.
RankList rl = [r].
String str = "foo".
Suit s = C.
SuitList sl = [s].
Team t = teams->first.
TeamList tl = teams.
```

Listing 9.15: decl_var_init.pip.log

```
Game "Foo" requires 2 players.
Boolean b = False.
Card c = J~C.
CardList cl = [c].
Deck d = cl.
Number n = 42.
Player p = (players)->first.
PlayerList pl = players.
Rank r = A.
RankList rl = [r].
String str = "foo".
Suit s = C.
SuitList sl = [s].
Team t = (teams)->first.
TeamList tl = teams.
```

Listing 9.16: expressions.pip

```
Game "foo" requires 2 players.

Action main {
  x = "abc".
  x = 42.
  x = True.
  x = False.
  x = [].
  x = [[]; []].
  x = [D; H; C; S].
  x = [A; K; Q; J].
  x = players.
  x = teams.
  x = standard.
  x = foo->bar.
  x = (J~C)->spaz.
  x = (x).
  x = x + x.
```

```
  x = x - x.
  x = x * x.
  x = x / x.
  x = x == x.
  x = x != x.
  x = x < x.
  x = x <= x.
  x = x > x.
  x = x >= x.
  x = x and x.
  x = x or x.
  x = not x.
  x = x in [].
  x = defined a->b.
  x = canplay x from y to z.
  x = x~x.
  x = x..x~x.
  x = x,x~x.
  x = x~x,x.
  x = %~%.
  x = %~x,x.
  x = x..x~%.
  x = x,x,x~%.
  x = x..x,x..x~%.
}
```

Listing 9.17: expressions.pip.log

```
Game "foo" requires 2 players.
Action main {
  x = "abc".
  x = 42.
  x = True.
  x = False.
  x = [].
  x = [[]; []].
  x = [D; H; C; S].
  x = [A; K; Q; J].
  x = players.
  x = teams.
  x = standard.
  x = (foo)->bar.
  x = (J~C)->spaz.
  x = x.
  x = (x) + (x).
  x = (x) - (x).
  x = (x) * (x).
  x = (x) / (x).
  x = (x) == (x).
  x = (x) != (x).
  x = (x) < (x).
  x = (x) <= (x).
  x = (x) > (x).
  x = (x) >= (x).
  x = (x) and (x).
  x = (x) or (x).
  x = not (x).
  x = (x) in ([]).
  x = defined (a)->b.
  x = canplay x from y to z.
  x = x~x.
  x = x..x~x.
  x = x,x~x.
  x = x~x,x.
  x = %~%.
  x = %~x,x.
  x = x..x~%.
  x = x,x,x~%.
  x = x..x,x..x~%.
}
```

Listing 9.18: game1.pip

```
Game "Name" requires 2 players.
```

Listing 9.19: game1.pip.log

```
Game "Name" requires 2 players.
```

Listing 9.20: game2.pip

```
Game "Name" requires 2 to 4 players.
```

Listing 9.21: game2.pip.log

```
Game "Name" requires 2 or 3 or 4 players.
```

Listing 9.22: game3.pip

```
Game "Name" requires 2 to 1 players.
```

Listing 9.23: game3.pip.log

```
Game "Name" requires 2 or 1 players.
```

Listing 9.24: game4.pip

```
Game "Name" requires 2 to 4 teams of 2.
```

Listing 9.25: game4.pip.log

```
Game "Name" requires 2 or 3 or 4 teams of 2.
```

Listing 9.26: identifiers.pip

```
Game "foo" requires 2 players.

Boolean b.
Boolean a_b_c.
Boolean a9.
Boolean FOOBAR.
```

Listing 9.27: identifiers.pip.log

```
Game "foo" requires 2 players.
Boolean b.
Boolean a_b_c.
Boolean a9.
Boolean FOOBAR.
```

Listing 9.28: illegal_char.pip

```

  !
```

Listing 9.29: illegal_char.pip.log

```
Error: Line 2, Col 3: Illegal character: !
```

Listing 9.30: illegal_char2.pip

```
Game "abc{de=}" requires 2 players.
```

Listing 9.31: illegal_char2.pip.log

```
Error: Line 1, Col 13: Nested string identifier expected; found: =
```

Listing 9.32: literal_strings.pip

```
Game "foo" requires 2 players.

Action main {
  x = "a\nb\tc".
  x = "a
```

```
b
c".
  x = "\a\b\c".
  x = "\"".
  x = "\{\}".
  x = "\\\\".
  x = "\{a{b}cd{e}{fgh}\}".
}
```

Listing 9.33: literal_strings.pip.log

```
Game "foo" requires 2 players.
Action main {
  x = "a\nb\tc".
  x = "a\nb\nc".
  x = "abc".
  x = "\"".
  x = "\{\}".
  x = "\\\\".
  x = "\{a{b}cd{e}{fgh}\}".
}
```

Listing 9.34: order_of_operations.pip

```
Game "foo" requires 2 players.

Action main {
  x = x -> x * x -> x.
  x = x * x + x * x.
  x = x + x == x + y.
  x = not x == x.
  x = not x and not y.
  x = x and x or x and x.
}
```

Listing 9.35: order_of_operations.pip.log

```
Game "foo" requires 2 players.
Action main {
  x = ((x)->x) * ((x)->x).
  x = ((x) * (x)) + ((x) * (x)).
  x = ((x) + (x)) == ((x) + (y)).
  x = not ((x) == (x)).
  x = (not (x)) and (not (y)).
  x = ((x) and (x)) or ((x) and (x)).
}
```

Listing 9.36: parse_error.pip

```


Deck d foo.
```

Listing 9.37: parse_error.pip.log

```
Error: syntax error on line 3
Fatal error: exception Parsing.Parse_error
```

Listing 9.38: shuffle.pip

```
Game "Foo" requires 2 players.
Deck d.

Action main {
  shuffle d.
}
```

Listing 9.39: shuffle.pip.log

```
Game "Foo" requires 2 players.
Deck d.
Action main {
  shuffle d.
}
```

Listing 9.40: statements.pip

```
Game "foo" requires 2 players.

Action main {
  ask player { "Yes" {}
               "No" if False {} }
  x = 2.
  d += q->r.
  d *= q->r.
  deal all from x to y.
  deal some from x->p to y->q.
  deal 2 + 2 from me to you.
  forever {}
  for x in p->q->r {}
  for z in [] starting at q { x(). }
  if p {} elseif q->r {} else {}
  foo().
  label bar.
  let t be x.
  message joe "forty two".
  order x by y.
  play valid from me to you.
  rotate chair.
  shuffle deck.
  skip to out_of_town.
  winner takes_all.
}
```

Listing 9.41: statements.pip.log

```
Game "foo" requires 2 players.
Action main {
  ask player {
    "Yes" {
    }
    "No" if False {
    }
  }
  x = 2.
  d += (q)->r.
  d *= (q)->r.
  deal all from x to y.
  deal some from (x)->p to (y)->q.
  deal (2) + (2) from me to you.
  forever {
  }
  for x in ((p)->q)->r {
  }
  for z in [] starting at q {
    x().
  }
  if p {
  }
  elseif (q)->r {
  }
  else {
  }
  foo().
  label bar.
  let t be x.
  message joe "forty two".
  order x by y.
  play valid from me to you.
  rotate chair.
  shuffle deck.
  skip to out_of_town.
  winner takes_all.
}
```

9.2.2 Semantic Tests

The semantic tests are run through "pip –analyze".

Listing 9.42: action_dup.pip

```
Game "foo" requires 2 players.

Action main {}
Action main {}
```

Listing 9.43: action_dup.pip.log

```
Error: Line 4, Col 1: Duplicate Action: main
```

Listing 9.44: area_dup.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
Area a labeled "a".

Action main {}
```

Listing 9.45: area_dup.pip.log

```
Error: Line 4, Col 1: Duplicate Variable: a
```

Listing 9.46: area_opts.pip

```
Game "foo" requires 2 players.

Area a labeled "a" is faceup, facedown.

Action main {}
```

Listing 9.47: area_opts.pip.log

```
Error: Line 3, Col 23: This option contradicts earlier options
```

Listing 9.48: ask.pip

```
Game "foo" requires 2 players.

Number b.

Action main {
  ask players->first {
    "a" {}
    "b" if b == b {}
    "c" if 42 {}
  }
}
```

Listing 9.49: ask.pip.log

```
Error: Line 9, Col 12: Expected type Boolean but found type Number
```

Listing 9.50: ask1.pip

```
Game "foo" requires 2 players.

Number b.

Action main {
  ask b {
    "" {}
  }
}
```

Listing 9.51: ask1.pip.log

```
Error: Line 6, Col 7: Expected type Player but found type Number
```

Listing 9.52: assign.pip

```
Game "foo" requires 2 players.

Number n.

Action main {
  n = 42.
}
```

Listing 9.53: assign.pip.log

```
```

Listing 9.54: assign1.pip

```
Game "foo" requires 2 players.

Number n.

Action main {
  n = False.
}
```

Listing 9.55: assign1.pip.log

```
Error: Line 6, Col 3: Expected type Number but found type Boolean
```

Listing 9.56: canplay.pip

```
Game "foo" requires 2 players.

Deck d.
Player p.
Boolean b.

Rule r(p, c, a) = True.

Action main {
  b = canplay r from p to d.
}
```

Listing 9.57: canplay.pip.log

```
```

Listing 9.58: canplay2.pip

```
Game "foo" requires 2 players.

Deck d.
Player p.
Boolean b.

Rule r(p, c, a) = True.

Action main {
  b = canplay d from p to d.
}
```

Listing 9.59: canplay2.pip.log

```
Error: Line 10, Col 7: Unknown rule 'd'
```

Listing 9.60: canplay3.pip

```
Game "foo" requires 2 players.

Deck d.
Player p.
Number b.

Rule r(p, c, a) = True.

Action main {
  b = canplay r from d to d.
}
```

Listing 9.61: canplay3.pip.log

```
Error: Line 10, Col 22: Expected type Player but found type CardList
```

Listing 9.62: canplay4.pip

```
Game "foo" requires 2 players.

Card c.
Player p.
Boolean b.

Rule r(p, c, a) = True.

Action main {
  b = canplay r from p to c.
}
```

Listing 9.63: canplay4.pip.log

```
Error: Line 10, Col 27: Expected type to be one of CardList or Player or Area but found type Card
```

Listing 9.64: cardexpr.pip

```
Game "foo" requires 2 players.

Card c.
CardList cl.
Rank r.
Suit s.

Action main {
  c = 2~C.
  c = 4~D.
  c = 6~H.
  c = 8~S.
  cl = J,Q,K,A~C.
  cl = 2..9~H.
  cl = 2,3..4,6..J,Q,K~C.
  cl = 2~C,H.
  cl = 2,3..5~C,D.
  c = r~s.
  cl = 2..r~C,s.
}
```

Listing 9.65: cardexpr.pip.log

```
```

Listing 9.66: cardexpr2.pip

```
Game "foo" requires 2 players.

CardList cl.
Rank r.
Suit s.

Action main {
  cl = 2~%.
  cl = Q~%.
  cl = 2,3~%.
  cl = %~C.
  cl = %~C,H,S.
  cl = %~%.
  cl = %~C,s.
  cl = 2,3..r~%.
}
```

Listing 9.67: cardexpr2.pip.log

```
```

Listing 9.68: compound.pip

```
Game "foo" requires 2 players.

Number n.

Action main {
  n += 42.
  n -= 42.
  n *= 42.
  n /= 42.
}
```

Listing 9.69: compound.pip.log

```
```

Listing 9.70: compound1.pip

```
Game "foo" requires 2 players.

Number n.

Action main {
  n += False.
}
```

Listing 9.71: compound1.pip.log

```
Error: Line 6, Col 8: Expected type Number but found type Boolean
```

Listing 9.72: compound2.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  b -= 42.
}
```

Listing 9.73: compound2.pip.log

```
Error: Line 6, Col 3: Expected type Number but found type Boolean
```

Listing 9.74: deal.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
Deck d.
Player p.
PlayerList pl.

Action main {
  deal all from a to a.
  deal all from d to a.
  deal all from p to a.
  deal all from a to d.
  deal all from d to d.
  deal all from p to d.
  deal all from a to p.
  deal all from d to p.
  deal all from p to p.
  deal all from a to pl.
  deal all from d to pl.
  deal all from p to pl.
}
```

Listing 9.75: deal.pip.log

```
```

Listing 9.76: deal10.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
PlayerList pl.

Action main {
  deal 2 from a to pl.
}
```

Listing 9.77: deal10.pip.log

```
```

Listing 9.78: deal11.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
PlayerList pl.
Card c.

Action main {
  deal c from a to pl.
}
```

Listing 9.79: deal11.pip.log

```
Error: Line 8, Col 3: Can't deal a single Card to a PlayerList
```

Listing 9.80: deal2.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
Card c.

Action main {
  deal all from a to a.
  deal 2 from a to a.
  deal c from a to a.
}
```

Listing 9.81: deal2.pip.log

```
```

Listing 9.82: deal6.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
Card c.

Action main {
  deal 2 from c to a.
}
```

Listing 9.83: deal6.pip.log

```
Error: Line 7, Col 15: Expected type to be one of CardList or Player or Area but found type Card
```

Listing 9.84: deal7.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
Card c.

Action main {
  deal 2 from a to c.
}
```

Listing 9.85: deal7.pip.log

```
Error: Line 7, Col 20: Expected type to be one of CardList or Player or PlayerList or Area but found type Card
```

Listing 9.86: deal8.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
Card c.

Action main {
  deal True from a to a.
}
```

Listing 9.87: deal8.pip.log

```
Error: Line 7, Col 8: Expected type to be one of Number or Card but found type Boolean
```

Listing 9.88: deal9.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
PlayerList pl.

Action main {
  deal all from a to pl.
}
```

Listing 9.89: deal9.pip.log

```
```

Listing 9.90: decl_others.pip

```
Game "foo" requires 2 players.

Area a labeled "bcd".
Area a2 labeled "def" is faceup, spreadout.

Rule r(a, b, c) = True.

Ordering o(a) = [].

Action main {}
```

Listing 9.91: decl_others.pip.log

```
```

Listing 9.92: decl_var.pip

```
Game "foo" requires 2 players.

Boolean b.
Card c.
CardList cl.
Deck d.
Number n.
Player p.
PlayerList pl.
Rank r.
RankList rl.
String str.
Suit s.
SuitList sl.
Team t.
TeamList tl.

Action main {}
```

Listing 9.93: decl_var.pip.log

```
```

Listing 9.94: decl_var_dup.pip

```
Game "foo" requires 2 players.

Boolean b.
Card b.

Action main {}
```

Listing 9.95: decl_var_dup.pip.log

```
Error: Line 4, Col 1: Duplicate Variable: b
```

Listing 9.96: decl_var_init.pip

```
Game "Foo" requires 2 players.

Boolean b = False.
Card c = J~C.
CardList cl = [c].
Deck d = cl.
Number n = 42.
Player p = players->first.
PlayerList pl = players.
Rank r = A.
RankList rl = [r].
String str = "foo".
Suit s = C.
SuitList sl = [s].
Team t = teams->first.
TeamList tl = teams.

Action main {}
```

Listing 9.97: decl_var_init.pip.log

```
```

Listing 9.98: decl_var_type.pip

```
Game "Foo" requires 2 players.

Number n = False.

Action main {}
```

Listing 9.99: decl_var_type.pip.log

```
Error: Line 3, Col 1: Expected type Number but found type Boolean
```

Listing 9.100: defined.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  b = defined b.
}
```

Listing 9.101: defined.pip.log

```
```

Listing 9.102: eq.pip

```
Game "foo" requires 2 players.

Action main {
   if 2 == Q {}
   if Q == 2 {}
}
```

Listing 9.103: eq.pip.log

```
```

Listing 9.104: expr.pip

```
Game "foo" requires 2 players.

Boolean b.
Number n.

Action main {
  b = not True.
  n = 1 + 2.
  n = 1 - 2.
  n = 1 * 2.
  n = 1 / 2.
  b = 1 == 2.
  b = 1 != 2.
  b = 1 < 2.
  b = 1 <= 2.
  b = 1 > 2.
  b = 1 >= 2.
  b = True and False.
  b = True or False.
  b = 2 in [2; 3; 4].
}
```

Listing 9.105: expr.pip.log

```
```

Listing 9.106: expr2.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  b = not 1.
}
```

Listing 9.107: expr2.pip.log

```
Error: Line 6, Col 11: Expected type Boolean but found type Number
```

Listing 9.108: expr3.pip

```
Game "foo" requires 2 players.

Number n.

Action main {
  n = False + True.
}
```

Listing 9.109: expr3.pip.log

```
Error: Line 6, Col 7: Expected type Number but found type Boolean
```

Listing 9.110: expr4.pip

```
Game "foo" requires 2 players.

Boolean b.
Area a labeled "a".

Action main {
  b = True == False.
  b = "a" == "a".
  b = a == a.
}
```

Listing 9.111: expr4.pip.log

```
Error: Line 9, Col 7: Invalid type Area for comparison
```

Listing 9.112: expr5.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  b = K < A.
  b = 2 < K.
  b = True < False.
}
```

Listing 9.113: expr5.pip.log

```
Error: Line 8, Col 7: Expected type Number but found type Boolean
```

Listing 9.114: expr6.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  b = 1 and 2.
}
```

Listing 9.115: expr6.pip.log

```
Error: Line 6, Col 7: Expected type Boolean but found type Number
```

Listing 9.116: expr7.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  b = 2 in [Q; K; A].
  b = True in [2; 3; 4].
}
```

Listing 9.117: expr7.pip.log

```
Error: Line 7, Col 7: Expected type Number but found type Boolean
```

Listing 9.118: for.pip

```
Game "foo" requires 2 players.

Deck d.
Card c.

Action main {
  for card in d {}
  for card in d starting at c {}
}
```

Listing 9.119: for.pip.log

```
```

Listing 9.120: for2.pip

```
Game "foo" requires 2 players.

Deck d.
Card c.

```

```
Action main {
  for card in d {
    c = card.
  }
}
```

Listing 9.121: for2.pip.log

```
```

Listing 9.122: for3.pip

```
Game "foo" requires 2 players.

Deck d.
Card c.

Action main {
  for card in d {
    let q be card.
    c = q.
  }
}
```

Listing 9.123: for3.pip.log

```
```

Listing 9.124: for4.pip

```
Game "foo" requires 2 players.

Deck d.
Card c.

Action main {
  for card in d {
    let card be card.
    c = card.
  }
}
```

Listing 9.125: for4.pip.log

```
```

Listing 9.126: for5.pip

```
Game "foo" requires 2 players.

Deck d.
Card c.

Action main {
  for card in d {
  }
  c = card.
}
```

Listing 9.127: for5.pip.log

```
Error: Line 9, Col 7: Unknown identifier 'card'
```

Listing 9.128: for6.pip

```
Game "foo" requires 2 players.

Deck d.
Boolean b.

Action main {
  for card in d starting at b {
  }
}
```

Listing 9.129: for6.pip.log

```
Error: Line 7, Col 29: Expected type Card but found type Boolean
```

Listing 9.130: game.pip

```
Game "a{b}c" requires 2 players.
```

Listing 9.131: game.pip.log

```
Error: Line 1, Col 9: Unknown identifier 'b'
```

Listing 9.132: game2.pip

```
Game "foo" requires 2 teams of 4.
```

Listing 9.133: game2.pip.log

```
Error: No 'main' action was declared.
```

Listing 9.134: if.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  if b {}
}
```

Listing 9.135: if.pip.log

```
```

Listing 9.136: if2.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  if b {} else {}
}
```

Listing 9.137: if2.pip.log

```
```

Listing 9.138: if3.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  if b {} elseif b {}
}
```

Listing 9.139: if3.pip.log

```
```

Listing 9.140: if4.pip

```
Game "foo" requires 2 players.

Boolean b.

Action main {
  if b {
    let a be b.
  } else {
    b = a.
  }
}
```

Listing 9.141: if4.pip.log

```
Error: Line 9, Col 9: Unknown identifier 'a'
```

Listing 9.142: if5.pip

```
Game "foo" requires 2 players.

Card c.

Action main {
  if c {}
}
```

Listing 9.143: if5.pip.log

```
Error: Line 6, Col 6: Expected type Boolean but found type Card
```

Listing 9.144: if6.pip

```
Game "foo" requires 2 players.

Boolean b.
Card c.

Action main {
  if b {}
  elseif c {}
}
```

Listing 9.145: if6.pip.log

```
Error: Line 8, Col 10: Expected type Boolean but found type Card
```

Listing 9.146: invoke.pip

```
Game "foo" requires 2 players.

Action main {
  main().
}
```

Listing 9.147: invoke.pip.log

```
Error: Line 4, Col 3: Unknown action 'main'
```

Listing 9.148: invoke2.pip

```
Game "foo" requires 2 players.

Action act {}

Action main {
  act().
}
```

Listing 9.149: invoke2.pip.log

```
```

Listing 9.150: let.pip

```
Game "foo" requires 2 players.

Action main {
  let a be 2.
  let b be 4.
  let c be [1; 2; 3].
}
```

Listing 9.151: let.pip.log

```
```

Listing 9.152: let2.pip

```
Game "foo" requires 2 players.

Action main {
  let a be 2.
  let a be [1; 2; 3].
}
```

Listing 9.153: let2.pip.log

```
Error: Line 5, Col 3: Duplicate Variable: a
```

Listing 9.154: let3.pip

```
Game "foo" requires 2 players.

Action main {
  let a be 2.
  forever {
    let a be 2.
  }
}
```

Listing 9.155: let3.pip.log

```
```

Listing 9.156: let4.pip

```
Game "foo" requires 2 players.

Action main {
  let a be 2.
  forever {
    let a be a.
  }
}
```

Listing 9.157: let4.pip.log

```
```

Listing 9.158: let5.pip

```
Game "foo" requires 2 players.

Action main {
  let a be 2.
  if a == a {}
}
```

Listing 9.159: let5.pip.log

```
```

Listing 9.160: let6.pip

```
Game "foo" requires 2 players.

Action main {
  forever {
    let a be 2.
  }
  if a == a {}
}
```

Listing 9.161: let6.pip.log

```
Error: Line 7, Col 6: Unknown identifier 'a'
```

Listing 9.162: let7.pip

```
Game "foo" requires 2 players.

Action main {
  let a be 2.
  if a == C {}
}
```

Listing 9.163: let7.pip.log

```
Error: Line 5, Col 11: Expected type Number but found type Suit
```

Listing 9.164: let8.pip

```
Game "foo" requires 2 players.

Number a.

Action main {
  let b be 2.
  forever {
    let c be 2.
    forever {
      let d be a + b + c.
    }
  }
}
```

Listing 9.165: let8.pip.log

```
```

Listing 9.166: list.pip

```
Game "foo" requires 2 players.

Rank r.
RankList rl.

Action main {
  rl = [].
  rl = [2; 3].
  rl = [5; Q; A].
  rl = [[]].
  rl = [r; [3; A]; Q].
  rl = [rl].
  rl = [rl; rl; 2; [Q; A]].
}
```

Listing 9.167: list.pip.log

```
```

Listing 9.168: list2.pip

```
Game "foo" requires 2 players.

RankList rl.

Action main {
  rl = [2; True].
}
```

Listing 9.169: list2.pip.log

```
Error: Line 6, Col 12: Expected type Number but found type Boolean
```

Listing 9.170: message.pip

```
Game "foo" requires 2 players.

Action main {
  message "hello world".
}
```

Listing 9.171: message.pip.log

```
```

Listing 9.172: message2.pip

```
Game "foo" requires 2 players.

Action main {
  message players->first "hello world".
}
```

Listing 9.173: message2.pip.log

```
```

Listing 9.174: message3.pip

```
Game "foo" requires 2 players.

Action main {
  message players "hello world".
}
```

Listing 9.175: message3.pip.log

```
```

Listing 9.176: message4.pip

```
Game "foo" requires 2 players.

Action main {
  message teams->first "hello world".
}
```

Listing 9.177: message4.pip.log

```
```

Listing 9.178: message5.pip

```
Game "foo" requires 2 players.

Action main {
  message teams "hello world".
}
```

Listing 9.179: message5.pip.log

```
Error: Line 4, Col 3: Expected type to be one of Player or PlayerList or Team but found type TeamList
```

Listing 9.180: message6.pip

```
Game "foo" requires 2 players.

Action main {
  message True.
}
```

Listing 9.181: message6.pip.log

```
Error: Line 4, Col 11: Expected type String but found type Boolean
```

Listing 9.182: order.pip

```
Game "foo" requires 2 players.

Deck d.

Ordering o(c) = c.

Action main {
  order d by o.
}
```

Listing 9.183: order.pip.log

```
```

Listing 9.184: order2.pip

```
Game "foo" requires 2 players.

Deck d.

Ordering o(c) = c.

Action main {
  order d by d.
}
```

Listing 9.185: order2.pip.log

```
Error: Line 8, Col 3: Unknown ordering 'd'
```

Listing 9.186: order3.pip

```
Game "foo" requires 2 players.

Card c.
Ordering o(c) = c.

Action main {
  order c by o.
}
```

Listing 9.187: order3.pip.log

```
Error: Line 7, Col 9: Expected type CardList but found type Card
```

Listing 9.188: ordering.pip

```
Game "foo" requires 2 players.

Ordering o(c) = c.

Action main {}
```

Listing 9.189: ordering.pip.log

```
```

Listing 9.190: ordering_dup.pip

```
Game "foo" requires 2 players.

Ordering o(c) = [].
Ordering o(c) = [].

Action main {}
```

Listing 9.191: ordering_dup.pip.log

```
Error: Line 4, Col 1: Duplicate Ordering: o
```

Listing 9.192: ordering_type.pip

```
Game "foo" requires 2 players.

Ordering o(c) = 42.

Action main {}
```

Listing 9.193: ordering_type.pip.log

```
Error: Line 3, Col 1: Expected type CardList but found type Number
```

Listing 9.194: play.pip

```
Game "foo" requires 2 players.

Deck d.
Player p.

Rule r(p, c, a) = True.

Action main {
  play r from p to d.
}
```

Listing 9.195: play.pip.log

```
```

Listing 9.196: play2.pip

```
Game "foo" requires 2 players.

Deck d.
Player p.

Rule r(p, c, a) = True.

Action main {
  play d from p to d.
}
```

Listing 9.197: play2.pip.log

```
Error: Line 9, Col 3: Unknown rule 'd'
```

Listing 9.198: play3.pip

```
Game "foo" requires 2 players.

Deck d.
Player p.

Rule r(p, c, a) = True.

Action main {
  play r from d to d.
}
```

Listing 9.199: play3.pip.log

```
Error: Line 9, Col 15: Expected type Player but found type CardList
```

Listing 9.200: play4.pip

```
Game "foo" requires 2 players.

Card c.
Player p.

Rule r(p, c, a) = True.

Action main {
  play r from p to c.
}
```

Listing 9.201: play4.pip.log

```
Error: Line 9, Col 20: Expected type to be one of CardList or Player or Area but found type Card
```

Listing 9.202: property.pip

```
Game "foo" requires 2 players.

Area a labeled "a".
```

```
Boolean b.
Card c.
CardList cl.
Rank r.
Suit s.
Player p.
PlayerList pl.
Team t.
String str.
Number n.

Action main {
  r = c->rank.
  s = c->suit.
  p = c->last_played_by.

  str = p->name.
  cl = p->hand.
  cl = p->stash.
  n = p->score.
  t = p->team.

  pl = t->members.
  cl = t->stash.
  n = t->score.

  str = a->name.
  cl = a->cards.
  b = a->is_facedown.
  b = a->is_squaredup.

  n = cl->size.
  c = cl->first.
  c = cl->last.
  c = cl->top.
  c = cl->bottom.

  n = ([2; 3])->size.
  n = ([])->size.
}
```

Listing 9.203: property.pip.log

```
```

Listing 9.204: property2.pip

```
Game "foo" requires 2 players.

Card c.
Boolean b.

Action main {
  b = c->rank.
}
```

Listing 9.205: property2.pip.log

```
Error: Line 7, Col 3: Expected type Boolean but found type Rank
```

Listing 9.206: property3.pip

```
Game "foo" requires 2 players.

Card c.
Boolean b.

Action main {
  b = c->unknown.
}
```

Listing 9.207: property3.pip.log

```
Error: Line 7, Col 10: Type Card has no property named 'unknown'
```

Listing 9.208: rank_num.pip

```
Game "Foo" requires 2 players.

Rank r = 2.
Number n = Q.

Action main {}
```

Listing 9.209: rank_num.pip.log

```
```

Listing 9.210: rotate.pip

```
Game "foo" requires 2 players.

Action main {
  rotate players.
}
```

Listing 9.211: rotate.pip.log

```
```

Listing 9.212: rotate2.pip

```
Game "foo" requires 2 players.

Deck d.

Action main {
  rotate d.
}
```

Listing 9.213: rotate2.pip.log

```
```

Listing 9.214: rotate3.pip

```
Game "foo" requires 2 players.

Card c.

Action main {
  rotate c.
}
```

Listing 9.215: rotate3.pip.log

```
Error: Line 6, Col 10: Expected a list type but found type Card
```

Listing 9.216: rule.pip

```
Game "foo" requires 2 players.

Rule r(p, c, cl) = (p == p and c == c and cl->top == cl->top).

Action main {}
```

Listing 9.217: rule.pip.log

```
```

Listing 9.218: rule_dup.pip

```
Game "foo" requires 2 players.

Rule r(p, c, a) = True.
Rule r(p, c, a) = True.

Action main {}
```

Listing 9.219: rule_dup.pip.log

```
Error: Line 4, Col 1: Duplicate Rule: r
```

Listing 9.220: rule_type.pip

```
Game "foo" requires 2 players.

Rule r(p, c, a) = 42.

Action main {}
```

Listing 9.221: rule_type.pip.log

```
Error: Line 3, Col 1: Expected type Boolean but found type Number
```

Listing 9.222: shuffle.pip

```
Game "foo" requires 2 players.

Deck d.

Action main {
  shuffle d.
}
```

Listing 9.223: shuffle.pip.log

```
```

Listing 9.224: shuffle2.pip

```
Game "foo" requires 2 players.

Action main {
  shuffle players.
}
```

Listing 9.225: shuffle2.pip.log

```
```

Listing 9.226: shuffle3.pip

```
Game "foo" requires 2 players.

Card c.

Action main {
  shuffle c.
}
```

Listing 9.227: shuffle3.pip.log

```
Error: Line 6, Col 11: Expected a list type but found type Card
```

Listing 9.228: skip.pip

```
Game "foo" requires 2 players.

Action main {
  skip to bar.
  label bar.
}
```

Listing 9.229: skip.pip.log

```
```

Listing 9.230: skip1.pip

```
Game "foo" requires 2 players.

Action main {
  skip to bar.
}
```

Listing 9.231: skip1.pip.log

```
Error: Line 4, Col 3: Can't find label 'bar'
```

Listing 9.232: skip2.pip

```
Game "foo" requires 2 players.

Action main {
  skip to bar.
  forever {
    skip to bar.
    label bar.
  }
}
```

Listing 9.233: skip2.pip.log

```
Error: Line 4, Col 3: Can't find label 'bar'
```

Listing 9.234: skip3.pip

```
Game "foo" requires 2 players.

Action main {
  label bar.
}
```

Listing 9.235: skip3.pip.log

```
Error: Line 4, Col 3: Unused label
```

Listing 9.236: skip4.pip

```
Game "foo" requires 2 players.

Action main {
  skip to bar.
  label bar.
  label bar.
}
```

Listing 9.237: skip4.pip.log

```
Error: Line 6, Col 3: Duplicate Label: bar
```

Listing 9.238: skip5.pip

```
Game "foo" requires 2 players.

Action main {
  label bar.
  skip to bar.
}
```

Listing 9.239: skip5.pip.log

```
Error: Line 4, Col 3: Unused label
```

Listing 9.240: skip6.pip

```
Game "foo" requires 2 players.

Action main {
  forever {
    skip to bar.
  }
  label bar.
}
```

Listing 9.241: skip6.pip.log

```
```

Listing 9.242: skip7.pip

```
Game "foo" requires 2 players.

Action main {
  skip to bar.
  forever {
    skip to bar.
    label bar.
  }
  label bar.
}
```

Listing 9.243: skip7.pip.log

```
Error: Line 9, Col 3: Duplicate Label: bar
```

Listing 9.244: skip8.pip

```
Game "foo" requires 2 players.

Action main {
  forever {
    skip to bar.
  }
  label bar.
}
```

Listing 9.245: skip8.pip.log

```
```

Listing 9.246: skip9.pip

```
Game "foo" requires 2 players.

Action main {
  skip to bar.
  let b be 2.
  label bar.
}
```

Listing 9.247: skip9.pip.log

```
Error: Line 5, Col 3: Can't skip past Let
```

Listing 9.248: str_lit.pip

```
Game "foo" requires 2 players.

Number n.
Boolean b.
Deck d.

String s = "a".
String t = "a{s}b".
String u = "a{n}b{b}c{d}".
```

Listing 9.249: str_lit.pip.log

```
Error: Line 9, Col 23: Can't convert CardList to a String
```

Listing 9.250: winner.pip

```
Game "foo" requires 2 players.

Action main {
  winner players->first.
}
```

Listing 9.251: winner.pip.log

```
```

Listing 9.252: winner2.pip

```
Game "foo" requires 2 players.

Action main {
  winner players.
}
```

Listing 9.253: winner2.pip.log

```
Error: Line 4, Col 10: Expected type to be one of Player or Team but found type PlayerList
```

9.3 Full Tests

9.3.1 Crazy Eights

Listing 9.254: crazy_eights.pip

```
Game "Crazy Eights" requires 2 to 4 players.

Deck d = standard.
Area draw_pile labeled "Draw Pile" is facedown.
Area discard_pile labeled "Discard Pile" is faceup.

Action setup {
  shuffle d.
  message "".
  message "Dealing 7 cards to everyone.".
  deal 7 from d to players.
  deal all from d to draw_pile.
  deal 1 from draw_pile to discard_pile.
}

Rule valid(p, c, cl) = c->rank == cl->top->rank or
                        c->suit == cl->top->suit or
                        c->rank == 8.

Action turn {
  let p be players->first.

  message "".
  message "It is {p}'s turn".

  forever {
    message p "You are holding the following cards:".
    for c in p->hand { message "  {c}". }

    let top be discard_pile->cards->top.
    message p "The top card of the discard pile is {top}.".

    if canplay valid from p to discard_pile {
      play valid from p to discard_pile.
      skip to played.
    } else {
      message p "You have nothing you can play.".
      ask p {
        "Draw" {
          deal 1 from draw_pile to p.
          let card be p->hand->top.
          message p "You picked up {card}.".
        }
      }

      if draw_pile->cards->size == 0 {
        let top be discard_pile->cards->top.
        deal all from discard_pile to draw_pile.
        deal top from draw_pile to discard_pile.
```

```
        shuffle draw_pile->cards.
      }
    }
  }

  label played.
}

Action main {
  setup().

  forever {
    turn().
    if players->first->hand->size == 0 { winner players->first. }
    rotate players.
  }
}
```

9.3.2 Euchre

Listing 9.255: euchre.pip

```
Game "Euchre" requires 2 teams of 2.

Area tricks labeled "Trick Area" is faceup, spreadout.
Area kitty labeled "Kitty" is facedown.

Suit trump.       # Trump suit
Suit trump_minor. # Same color as trump suit
Team maker.       # Who chose trump
Player next.      # Next to play.

Card ordered.
Rule discard(p, c, cl) = c != ordered.

Action setup {
  let d be 9..K,A~%.
  shuffle d.

  # Clear the stashes
  for t in teams { t->stash = []. }

  # Set the first player to play.
  next = players->first.

  message "".
  message "Dealing 5 cards to everyone.".

  deal 5 from d to players.
  deal all from d to kitty.

  let dealer be players->last.
  message "Dealer is {dealer}.".

  let top be kitty->cards->top.
  message "Top of kitty is {top}. Choosing trump.".

  for p in players {
    message p "Your hand contains:".
    for c in p->hand {
      message "  {c}".
    }
    ask p {
      "Order up {top} to {dealer}'s hand" {
        trump = top->suit.
        maker = p->team.
        deal top from kitty to dealer.
        message dealer "You acquired {top}. You must discard something.".
        ordered = top.
        play discard from dealer to kitty.
        skip to trump_found.
      }
      "Pass" {}
```

```
    }
  }

  message "No one called up {top}. Pick a suit to be trump.".

  for p in players {
    message p "Your hand contains:".
    for c in p->hand {
      message "  {c}".
    }
    ask p {
      "Spades"   if top->suit != S { trump = S. maker = p->team. skip to trump_found. }
      "Clubs"    if top->suit != C { trump = C. maker = p->team. skip to trump_found. }
      "Hearts"   if top->suit != H { trump = H. maker = p->team. skip to trump_found. }
      "Diamonds" if top->suit != D { trump = D. maker = p->team. skip to trump_found. }
      "Pass"     if p != dealer {}
    }
  }

  label trump_found.

  # Reset old change
  if defined trump_minor {
    (J~trump_minor)->suit = trump_minor.
  }

  # Find new trump_minor
  if trump == C {
    trump_minor = S.
  } elseif trump == S {
    trump_minor = C.
  } elseif trump == H {
    trump_minor = D.
  } else {
    trump_minor = H.
  }

  # Jack of same color is part of trump suit.
  (J~trump_minor)->suit = trump.
}

Ordering jacks_win(c) = [J~trump; J~trump_minor; A,K..9~trump; A,K..9~c->last->suit; A,K..9~%].

Rule valid(p, c, cl) =
  (cl->size == 0) or
  (cl->last->suit in p->hand and c->suit == cl->last->suit) or
  (not cl->last->suit in p->hand).

# A single hand of Euchre is 5 tricks or rounds. Each round is described here.
Action round {
  # Start with who won the last trick. At the beginning
  # of the game, start with the person left of the dealer.

  message "".
  message "Trump is {trump}.".
  message "Maker is {maker}.".

  for t in teams {
    let tricks be t->stash->size / players->size.
    message "{t} has taken {tricks} tricks so far.".
  }

  message "{next} starts the trick.".

  for p in players starting at next {
    play valid from p to tricks.
    message "Cards played so far:".
    for c in tricks->cards {
      let p be c->last_played_by.
      message "  {c}, played by {p}".
    }
  }

  # Decide who took the trick.

  order tricks->cards by jacks_win.
```

```
  let highcard be tricks->cards->first.
  let w be highcard->last_played_by.

  deal all from tricks to w->team->stash.

  message "{w} took the trick with {highcard}.".
  next = w.
}

Action score {
  for t in teams {
    let tricks be t->stash->size / players->size.
    message "{t} took {tricks} tricks.".

    if t == maker {
      if tricks == 5 {
        message "  {t} scores 2 points.".
        t->score += 2.
      } elseif tricks >= 3 {
        message "  {t} scores 1 point.".
        t->score += 1.
      }
    } else {
      if tricks == 5 {
        message "  {t} scores 4 points.".
        t->score += 4.
      } elseif tricks >= 3 {
        message "  {t} scores 2 points.".
        t->score += 2.
      }
    }
  }

  message "The scores:".
  for t in teams {
    let p be t->score.
    message "  {t} has {p} points.".
  }
}

Action main {
  forever {
    setup().

    forever {
      round().
      if players->first->hand->size == 0 { skip to score. }
    }

    label score.
    score().

    for t in teams {
      if t->score >= 10 { winner t. }
    }

    rotate players.
  }
}
```

9.4 Build System

9.4.1 Top-level Makefile

Listing 9.256: Makefile

```
#
# Description:
#
#  This top-level Makefile is responsible for compiling all files in src/
#  and creating the binaries in bin/, documents in doc/, etc.
```

```
#
#  Here are the targets:
#
#    make          # Build the binaries.
#    make clean    # Remove all generated files.
#    make tests    # Run all of the tests.
#    make logs     # Run all tests but generate logs instead of checking them.
#    make pdfs     # Build the pdfs.
#    make <file>   # Re-build the named file (bin/prog, etc).
#

# Must be set first
TOP := .

# The default rule
.PHONY: default
default: build

# Settings
include $(TOP)/mk/settings.mk.inc

# Rules for this directory
.PHONY: clean
clean:
	$(Q)rm -rf $(BIN_DIR) $(TEMP_DIR) $(DOC_DIR)

.PHONY: tests
tests: build
	$(Q)export TOP=$(TOP);                                              \
	    for X in $(TEST_DIR)/syntactic/*.pip; do                        \
	      echo "   T $$X";                                              \
	      $(TESTIT) $$X.log $(BIN_DIR)/pip --print-ast $$X &&           \
	        $(TESTIT) $$X.log $(BIN_DIR)/pip --print-ast $$X.log;       \
	    done;                                                           \
	    for X in $(TEST_DIR)/semantic/*.pip; do                         \
	      echo "   T $$X";                                              \
	      $(TESTIT) $$X.log $(BIN_DIR)/pip --analyze $$X;               \
	    done;                                                           \
	    for X in $(TEST_DIR)/full/*.pip; do                             \
	      echo "   T $$X";                                              \
	      $(TESTIT) $$X.log $(BIN_DIR)/pip --analyze $$X;               \
	    done

.PHONY: logs
logs: build
	$(Q)export TOP=$(TOP);                                              \
	    for X in $(TEST_DIR)/syntactic/*.pip; do                        \
	      echo "   G $$X";                                              \
	      $(BIN_DIR)/pip --print-ast $$X > $$X.log 2>&1;                \
	    done;                                                           \
	    for X in $(TEST_DIR)/semantic/*.pip; do                         \
	      echo "   G $$X";                                              \
	      $(BIN_DIR)/pip --analyze $$X > $$X.log 2>&1;                  \
	    done;                                                           \
	    for X in $(TEST_DIR)/full/*.pip; do                             \
	      echo "   G $$X";                                              \
	      $(BIN_DIR)/pip --analyze $$X > $$X.log 2>&1;                  \
	    done

# Load all TOP/src/**/Make*.inc
MF := $(shell find $(TOP)/src -name 'Make*.inc')
include $(MF)

# The standard rules
include mk/rules.mk.inc
```

9.4.2 Interpreter Makefiles

Listing 9.257: src/pip/Makefile

```
#
# Description:
#
```

```
#  This Makefile is a generic sub-directory Makefile. It includes the
#  specific rules for this sub-directory and sets up a default rule.
#
#  Here are the targets:
#
#    make           # Build the sub-directory
#    make <file>    # Re-build the named file (bin/prog, etc)
#

# Always first. Relative path to TOP.
TOP := ../..

# The default rule
.PHONY: default
default: build

# Settings
include $(TOP)/mk/settings.mk.inc

# This directory's setup
include Makefile.inc

# Rules
include $(TOP)/mk/rules.mk.inc
```

Listing 9.258: src/pip/Makefile.inc

```
#
# Description:
#
#  This Makefile is for the pip binary. It is responsible for
#  setting up the build system so that all the source files in
#  this directory end up in a binary in TOP/bin.
#

HERE := $(TOP)/src/pip

# This directory's policy
include $(TOP)/mk/binary_from_ml.mk.inc

# Hack: The ocamldep program isn't smart enough to see this dependency
$(TEMP_DIR)/src/pip/parser.cmi: $(TEMP_DIR)/src/pip/ast.cmo
$(TEMP_DIR)/src/pip/parser.cmi: $(TEMP_DIR)/src/pip/utils.cmo
```

9.4.3 Support Makefiles

Listing 9.259: mk/binary_from_ml.mk.inc

```
#
# Description:
#
#  This includable Makefile sets up a binary in $(TOP)/bin to be created
#  from all of the *.ml, *.mli, *.mly, and *.mll files in the current directory.
#
#  The current directory is given by the variable $(HERE).
#
#  The name of the binary will be the same as the name of the directory
#  unless $(MY_BIN) is set to an absolute path. This Makefile unsets
#  $(MY_BIN) at the end.
#

# $(HERE) and $(TOP) are set. Find the relative path from TOP to HERE
# and set it as $(REL_HERE).

# Make 3.81:
# ABS_TOP  := $(realpath $(TOP))
# ABS_HERE := $(realpath $(HERE))

# Make 3.80:
ABS_TOP  := $(shell cd "$(TOP)"; /bin/pwd)
ABS_HERE := $(shell cd "$(HERE)"; /bin/pwd)
```

```
REL_HERE := $(patsubst $(ABS_TOP)/%,%,$(ABS_HERE))

# Set up sources and objects local to this directory
MY_ML_SRC := $(wildcard $(HERE)/*.ml)
MY_ML_OBJ := $(patsubst $(HERE)/%.ml,$(TEMP_DIR)/$(REL_HERE)/%.cmo,$(MY_ML_SRC))
MY_ML_DEP := $(patsubst $(HERE)/%.ml,$(TEMP_DIR)/$(REL_HERE)/%.cmo.d,$(MY_ML_SRC))

MY_MLI_SRC := $(wildcard $(HERE)/*.mli)
MY_MLI_OBJ := $(patsubst $(HERE)/%.mli,$(TEMP_DIR)/$(REL_HERE)/%.cmi,$(MY_MLI_SRC))
MY_MLI_DEP := $(patsubst $(HERE)/%.mli,$(TEMP_DIR)/$(REL_HERE)/%.cmi.d,$(MY_MLI_SRC))

MY_MLL_SRC := $(wildcard $(HERE)/*.mll)
MY_MLL_ML  := $(patsubst $(HERE)/%.mll,$(TEMP_DIR)/$(REL_HERE)/%.ml,$(MY_MLL_SRC))
MY_MLL_OBJ := $(patsubst $(TEMP_DIR)/$(REL_HERE)/%.ml,$(TEMP_DIR)/$(REL_HERE)/%.cmo,$(MY_MLL_ML))
MY_MLL_DEP := $(patsubst $(TEMP_DIR)/$(REL_HERE)/%.ml,$(TEMP_DIR)/$(REL_HERE)/%.cmo.d,$(MY_MLL_ML))

MY_MLY_SRC   := $(wildcard $(HERE)/*.mly)
MY_MLY_ML    := $(patsubst $(HERE)/%.mly,$(TEMP_DIR)/$(REL_HERE)/%.ml,$(MY_MLY_SRC))
MY_MLY_MLI   := $(patsubst $(HERE)/%.mly,$(TEMP_DIR)/$(REL_HERE)/%.mli,$(MY_MLY_SRC))
MY_MLY_OBJ   := $(patsubst $(TEMP_DIR)/$(REL_HERE)/%.ml,$(TEMP_DIR)/$(REL_HERE)/%.cmo,$(MY_MLY_ML))
MY_MLY_DEP   := $(patsubst $(TEMP_DIR)/$(REL_HERE)/%.ml,$(TEMP_DIR)/$(REL_HERE)/%.cmo.d,$(MY_MLY_ML))
MY_MLY_OBJI := $(patsubst $(TEMP_DIR)/$(REL_HERE)/%.mli,$(TEMP_DIR)/$(REL_HERE)/%.cmi,$(MY_MLY_MLI))
MY_MLY_DEPI := $(patsubst $(TEMP_DIR)/$(REL_HERE)/%.mli,$(TEMP_DIR)/$(REL_HERE)/%.cmi.d,$(MY_MLY_MLI))

MY_BIN := $(if $(MY_BIN),$(MY_BIN),$(BIN_DIR)/$(notdir $(ABS_HERE)))

# These pass information to the global Make system (like rules.mk.inc)
LEXFILES  += $(MY_MLL_ML)
YACCFILES += $(MY_MLY_ML)
OBJECTS   += $(MY_ML_OBJ) $(MY_MLI_OBJ) $(MY_MLL_OBJ) $(MY_MLY_OBJ) $(MY_MLY_OBJI)
DEPS      += $(MY_ML_DEP) $(MY_MLI_DEP) $(MY_MLL_DEP) $(MY_MLY_DEP) $(MY_MLY_DEPI)
BINARIES  += $(MY_BIN)

# These link the sources to the objects, and the objects to the binary
$(MY_ML_OBJ):  $(TEMP_DIR)/$(REL_HERE)/%.cmo:   $(HERE)/%.ml
$(MY_ML_DEP):  $(TEMP_DIR)/$(REL_HERE)/%.cmo.d: $(HERE)/%.ml

$(MY_MLI_OBJ): $(TEMP_DIR)/$(REL_HERE)/%.cmi:   $(HERE)/%.mli
$(MY_MLI_DEP): $(TEMP_DIR)/$(REL_HERE)/%.cmi.d: $(HERE)/%.mli

$(MY_MLL_ML):  $(TEMP_DIR)/$(REL_HERE)/%.ml:    $(HERE)/%.mll
$(MY_MLL_OBJ): $(TEMP_DIR)/$(REL_HERE)/%.cmo:   $(TEMP_DIR)/$(REL_HERE)/%.ml
$(MY_MLL_DEP): $(TEMP_DIR)/$(REL_HERE)/%.cmo.d: $(TEMP_DIR)/$(REL_HERE)/%.ml

$(MY_MLY_ML):   $(TEMP_DIR)/$(REL_HERE)/%.ml:    $(HERE)/%.mly
$(MY_MLY_MLI):  $(TEMP_DIR)/$(REL_HERE)/%.mli:   $(HERE)/%.mly
$(MY_MLY_OBJ):  $(TEMP_DIR)/$(REL_HERE)/%.cmo:   $(TEMP_DIR)/$(REL_HERE)/%.ml
$(MY_MLY_OBJI): $(TEMP_DIR)/$(REL_HERE)/%.cmi:   $(TEMP_DIR)/$(REL_HERE)/%.mli
$(MY_MLY_DEP):  $(TEMP_DIR)/$(REL_HERE)/%.cmo.d: $(TEMP_DIR)/$(REL_HERE)/%.ml
$(MY_MLY_DEPI): $(TEMP_DIR)/$(REL_HERE)/%.cmi.d: $(TEMP_DIR)/$(REL_HERE)/%.mli

# Order is important here. The rules.mk.inc file uses scripts/ocamlorder
# to ensure it is ok.

$(MY_BIN): $(MY_MLY_OBJ) $(MY_MLL_OBJ) $(MY_ML_OBJ)

# Unset this so future included Makefiles can set it up
MY_BIN :=
```

Listing 9.260: mk/pdf_from_all.mk.inc

```
#
# Description:
#
#  This includable Makefile sets up a pdf in $(TOP)/tmp/<dir> to be created
#  for each *.tex in the current directory.
#
#  The current directory is given by the variable $(HERE).
#
#  The name of the pdf will be the same as the name of the tex file.
#

# $(HERE) and $(TOP) are set. Find the relative path from TOP to HERE
# and set it as $(REL_HERE).
```

```
# Make 3.81:
# ABS_TOP  := $(realpath $(TOP))
# ABS_HERE := $(realpath $(HERE))

# Make 3.80:
ABS_TOP  := $(shell cd "$(TOP)"; /bin/pwd)
ABS_HERE := $(shell cd "$(HERE)"; /bin/pwd)

REL_HERE := $(patsubst $(ABS_TOP)/%,%,$(ABS_HERE))

# Set up sources
MY_DOT   := $(wildcard $(HERE)/*.dot)
MY_TEX   := $(wildcard $(HERE)/*.tex)
MY_PSPDF := $(patsubst $(HERE)/%.dot,$(TEMP_DIR)/$(REL_HERE)/%.pdf,$(MY_DOT))
MY_DEPS  := $(patsubst $(HERE)/%.tex,$(TEMP_DIR)/$(REL_HERE)/%.pdf.d,$(MY_TEX))
MY_PDF   := $(patsubst $(HERE)/%.tex,$(DOC_DIR)/%.pdf,$(MY_TEX))

# These pass information to the global Make system (like rules.mk.inc)
PSPDFS  += $(MY_PSPDF)
PDFS    += $(MY_PDF)
TEXDEPS += $(MY_DEPS)

# These link the sources to the objects
$(MY_PSPDF): $(TEMP_DIR)/$(REL_HERE)/%.pdf: $(HERE)/%.dot
$(MY_DEPS):  $(TEMP_DIR)/$(REL_HERE)/%.pdf.d: $(HERE)/%.tex
$(MY_PDF):   $(DOC_DIR)/%.pdf: $(HERE)/%.tex
```

Listing 9.261: mk/rules.mk.inc

```
#
# Description:
#
#  This includable Makefile sets up the rules for building generic
#  targets from generic sources. It should be included at the bottom
#  of an invocable Makefile that needs the rules.
#

.PHONY: build
build: $(BINARIES)

.PHONY: pdfs
pdfs: $(PDFS)

$(PDFS):
	@# The pdf dependencies include more than just .tex
	@# files, but those are the only files we can pass
	@# on the command-line.
	$(Q)mkdir -p $(dir $@)
	$(Q)mkdir -p $(TEMP_DIR)/pdf
	@echo " PDF $(filter %.tex,$+) => $@"
	@# Run twice for table-of-contents generation
	$(Q)$(SILENCE) $(PDFLATEX) -halt-on-error -output-directory $(TEMP_DIR)/pdf $(filter %.tex,$+)
	$(Q)$(SILENCE) $(PDFLATEX) -halt-on-error -output-directory $(TEMP_DIR)/pdf $(filter %.tex,$+)
	$(Q)$(CP) $(TEMP_DIR)/pdf/$(notdir $@) $@

$(PSPDFS):
	$(Q)mkdir -p $(dir $@)
	@echo " DOT $<"
	$(Q)$(DOT) -Tpdf $< > $@

$(BINARIES):
	$(Q)mkdir -p $(dir $@)
	@echo "  OC $@"
	$(Q)$(CC) -o $@ $(shell $(OORDER) $+)

$(LEXFILES):
	$(Q)mkdir -p $(dir $@)
	@echo "  LX $<"
	$(Q)$(LEX) -q -o $@ $<

$(YACCFILES):
	$(Q)mkdir -p $(dir $@)
	@echo "  YC $<"
	$(Q)$(YACC) -b $(dir $@)/$(basename $(notdir $@)) $<

```

```
$(OBJECTS):
	@# The ocamldep dependencies add in more than just .ml and .mll
	@# files, but those are the only files we can pass on the command-line.
	$(Q)mkdir -p $(dir $@)
	$(Q)echo "  OC $(filter %.ml,$+) $(filter %.mli,$+)"
	$(Q)$(CC) -I $(dir $@) -c -o $@ $(filter %.ml,$+) $(filter %.mli,$+)

$(DEPS):
	$(Q)mkdir -p $(dir $@)
	@echo " DEP $<"
	$(Q)$(DEP) -I $(dir $<) -I $(dir $@) $< > $@
	$(Q)$(FIXDEP) $(TOP) $@ $(dir $<) $(dir $@)

$(TEXDEPS):
	$(Q)mkdir -p $(dir $@)
	@echo " DEP $<"
	$(Q)$(TEXDEP) $< $(DOC_DIR)/$(basename $(notdir $<)).pdf > $@
	$(Q)$(FIXDEP) $(TOP) $@

# Include any automatically-generated dependency information
ifneq "clean" "$(MAKECMDGOALS)"
-include $(DEPS)
-include $(TEXDEPS)
endif
```

9.5 Support Scripts

9.5.1 Dependency Fixer

Listing 9.262: fixdep

```
#!/usr/bin/env perl
#
# Called with two arguments: The path to $(TOP) and the path to a dep file.
#
# This script fixes up the dep file so that it can be used from anywhere
# in the tree - it makes all file references that are currently relative
# to the value of $(TOP) to instead be relative to the variable $(TOP).
#
# For example, it changes this:
#
#    foo/bar.cmo:   src/baz.cmi
#
# with a TOP of "." to this:
#
#    $(TOP)/foo/bar.cmo:   $(TOP)/src/baz.cmi
#
# To fix limitations in ocamldep, special actions are done if two extra
# paths, A and B, are given.
#
# 1) Any path starting with A is changed to start with B
# 2) Any bare name with no relative path is also changed to start with B

sub trim($)
{
  my($s) = @_;
  $s =~ s|^\s+||;
  $s =~ s|\s+$||;
  return $s;
}

die "Usage: fixdep <TOP> <depfile> [A B]\n" unless ( scalar(@ARGV) == 2 or scalar(@ARGV) == 4 );

my $top = shift @ARGV;
my $dep = shift @ARGV;
my $a   = shift @ARGV;
my $b   = shift @ARGV;

my @newdep;

my $prev = "";
```

```
open( DEP, "<", $dep ) or die "Can't read '$dep': $!\n";
while ( <DEP> ) {
  my $line = "$prev$_";
  chomp($line);
  if ( $line =~ m|\\$| ) {
    $prev = $line;
    $prev =~ s|\\$||;
    next;
  }
  $prev = "";
  my ( $left, $right ) = split( ':', $line, 2 );
  my @left = split( /\s+/, trim($left) );
  my @right = split( /\s+/, trim($right) );
  do { s|^\Q$a\E|$b|g foreach ( @left, @right ) } if ( defined $a and defined $b );
  do { m|/| or s|^|$b|g foreach ( @left, @right ) } if ( defined $a and defined $b );
  do { m|^\.| or m|^/| or s|^|./|g foreach ( @left, @right ) };
  s|^\Q$top\E|\$(TOP)|g foreach ( @left, @right );
  push @newdep, sprintf( "%s: %s\n", join( ' ', @left ), join( ' ', @right ) );
}
close( DEP ) or die "Can't close '$dep': $!\n";

open( DEP, ">", $dep ) or die "Can't write '$dep': $!\n";
print DEP @newdep;
close( DEP ) or die "Can't write '$dep': $!\n";

exit(0);
```

9.5.2 Silencer

Listing 9.263: silence

```
#!/usr/bin/env perl
#
# Simple script that runs arguments with no output unless
# the arguments return non-zero.
#
# Simplifies the ever-common:
#
#   command > /dev/null 2>&1
#
# except that if 'command' errors, the stdout and stderr will
# be shown at that point.
#
##################################################################

my $output, $pid;

$pid = open( CHILD, "-|" );

if( $pid )
{
  # Parent
  local $/ = undef; # No line buffering, get all at next read
  $output = <CHILD>;
  if( ! close( CHILD ) )
  {
    # $! is non-zero if a system call failed during the close process (unlikely)
    die "$0: $ARGV[0]: $!\n" if $!;

    # Otherwise, non-zero exit status
    print $output;
    exit ($? >> 8);
  }
}
elsif( defined $pid )
{
  # Child
  open( STDERR, ">&STDOUT" ) or die "$0: dup: $!\n";

  exec( { $ARGV[0] } @ARGV ); # Shouldn't return

  die "$0: $ARGV[0]: $!\n"; # exec failed
```

```
}
else
{
  # Fork failed
  die "$0: fork: $!\n";
}
```

9.5.3 LaTeX Dependency Scanner

Listing 9.264: texdep

```
#!/usr/bin/env perl
#
# This script takes in a *.tex input file and a *.pdf output
# file and outputs a list of rules that force the output file
# to depend on any files that the input includes via a list of
# recognized TeX directives.

my @directives = qw(
  includegraphics
  lstinputlisting
);

sub usage
{
  die "Usage: texdep <input.tex> <output.pdf>\n";
}

my $tex = shift @ARGV or usage();
my $pdf = shift @ARGV or usage();

open(IN, "<", $tex) or die "Can't read '$tex': $!\n";

while (<IN>) {
  my $line = $_;
  chomp($line);

  foreach my $dir ( @directives ) {
    while ( $line =~ m|\\\Q$dir\E\[.*?\]{(.*?)}|g ) {
      print "$pdf: $1\n";
    }
  }
}

close(IN) or die "Can't read '$tex': $!\n";
```

9.5.4 Testcase Driver

Listing 9.265: testit

```
#!/bin/sh
#
# This takes in an output log and a program to run.
#
# This runs the program and compares the output to the log.

if [ $# -lt 2 ]; then
  echo "Error: Usage: testit <log> cmd arg arg..." 1>&2
  exit 1
fi

GOAL="$1"
shift

# Temp file, with automatic cleanup
LOG="/tmp/testit.$$.log"
trap 'rm -f "$LOG"' EXIT

# Run
"$@" > "$LOG" 2>&1
```

```
RC=$?

# Compare
diff "$GOAL" "$LOG"

# Cleanup happens automatically
exit $RC
```

9.5.5 OCaml Dependency Ordering Tool

Listing 9.266: ocamlorder

```
#!/usr/bin/env perl
#
# This takes in a list of *.cmo files and orders them topologically.
# This errors out if there is a cycle.
# This uses "ocamlobjinfo" and "tsort", both of which must be in the PATH.
# This prints out the *.cmo files in order so that a link will succeed.

sub usage
{
  print STDERR "Usage: ocamlorder <*.cmo>\n";
  exit 1;
}

sub main
{
  usage() unless @ARGV;

  # Read in the dep information
  my %deps;
  my %map;
  my @avail = map { modname($_) } @ARGV;

  foreach my $x ( @ARGV ) {
    my $name = modname($x);
    $map{$name} = $x;
    $deps{$name} = find_deps($x, \@avail);
  }

  # Do a topo-sort

  my $order = topo_sort(\%deps);

  # Map back to input file names

  my @sorted = map { $map{$_} } @$order;

  # Write out the results

  print "@sorted\n";

  exit(0);
}

# foo/bar.cmo => bar
sub modname
{
  my( $fname ) = @_;
  $fname =~ s|\..*?$||;
  $fname =~ s|.*/||;
  return ucfirst($fname);
}

# Use ocamlobjinfo to list deps. Only return a list of
# those deps that exist in the given list of names.
sub find_deps
{
  my( $file, $avail ) = @_;

  my @cmd = ('ocamlobjinfo', $file);

  my $imp = 0;
```

```
  my @found;
  my $self = modname($file);

  open( INFO, "-|", @cmd ) or die "Can't run 'ocamlobjinfo': $!\n";

  while ( <INFO> ) {
    # Interfaces imported:
    #         71f888453b0f26895819460a72f07493          Pervasives
    #         ...
    if ( /Interfaces\s+imported/ ) {
      $imp = 1;
    }

    if ( $imp ) {
      if ( /^\s*[0-9a-f]{32}\s+(\S+)\s*$/ ) {
        my $name = $1;
        if ( grep { $_ eq $name and $_ ne $self } @$avail ) {
          push @found, $name;
        }
      }
    }
  }

  close( INFO ) or die "Running 'ocamlobjinfo' failed: $! $?\n";

  return \@found;
}

# Take a hash that maps strings to dependencies, return
# a topologically sorted list
my $tf;
END { unlink($tf) if $tf; }

sub topo_sort
{
  my( $deps ) = @_;

  $tf = "/tmp/ocamlorder.$$" unless $tf;

  open( TMP, ">$tf" ) or die "Can't write '$tf': $!\n";

  foreach my $d ( keys %$deps ) {
    my @v = @{$deps->{$d}};
    foreach my $v ( @v ) {
      print TMP "$v $d\n";
    }
  }

  close( TMP ) or die "Can't write '$tf': $!\n";

  # Use tsort

  my @sorted;
  my @cmd = ('tsort', $tf);

  open( TS, "-|", @cmd ) or die "Can't run 'tsort': $!\n";

  while ( <TS> ) {
    chomp;
    push @sorted, $_;
  }

  close( TS ) or die "Running 'tsort' failed: $! $?\n";

  return \@sorted;
}

main();
```