The Pip Language Reference Manual

PLT (W4115)
Fall 2008

Frank Wallingford (frw2106@columbia.edu)

Contents

1 Introduction

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.

2 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. The game state is initialized. All card objects take on their normal default values.
3. 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`.
4. 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 player or team with the highest score wins.

3 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.

3.1 ASCII

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

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

3.2 Whitespace

Whitespace separates tokens. Whitespace is defined as follows:

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

3.3 Comments

A comment is a sequence of characters that is ignored by the Pip interpreter. Comments begin with a `#` and contine 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"
```

3.4 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 progam, 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.

3.4.1 Punctuation and Expressions

The following tokens are used for punctuation and in expressions:

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

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

3.4.3 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
```

3.4.4 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 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 references 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.

4.1 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.

4.2 Types

4.2.1 Number

`Number` is a `basic type`.

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

4.2.2 Boolean

`Boolean` is a `basic type`.

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

4.2.3 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`.

4.2.4 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`.

4.2.5 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.

4.2.6 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`.

4.2.7 Deck

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

4.2.8 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`.

4.2.9 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
players	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.

4.2.10 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

An area's `cards` starts out empty.

4.2.11 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.

4.2.12 Ordering

`Ordering` is a `complex type`.

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

4.2.13 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.

4.2.14 List

There is no generic `List` type; however, all specific list types below share the following `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

4.2.15 CardList

`CardList` is a `list type`.

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

4.2.16 RankList

`RankList` is a `list type`.

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

4.2.17 SuitList

`SuitList` is a `list type`.

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

4.2.18 PlayerList

`PlayerList` is a `list type`.

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

4.2.19 TeamList

`TeamList` is a `list type`.

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

5 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` declaraion contains statements and expressions which manipulate the environment, interact with players, and control the flow of execution.

5.1 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.

5.2 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 identifier that names the new object and ends at the end of the file. Declarations can only appear at the top level.

There is a single namespace for all declarations and types, and a second namespace for labels.

Declarations of `object type` and `list type` that don't specify an initial value gain the value `undefined`.

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

5.2.1 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 face down and squared up.

5.2.2 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.

5.2.3 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, a) = 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 area, 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`.

5.2.4 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`.

5.2.5 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 basic type defines a new object that can hold a copy of any value of that basic type. Assignments to the object copy the value being assigned into the new object.

If a `basic type` is defined with no initializer, then it is initialized according to its type:

Type	Initial Value
Number	0
Boolean	False
String	""
Rank	A
Suit	C

If an `object type` or `list type` is defined with no initializer, it's initial value is `undefined`.

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

5.3 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.

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 auto restock

The auto restock statement sets up an automatic deal from one cardlist to another.

```
autostmt: "auto" "restock" ref "from" ref "."
        | "auto" "restock" ref "from" ref "leaving" expr "."
```

Example:

```
auto restock pilea from pileb.
auto restock pilea from pileb leaving 2.
```

Semantics:

After this statement is executed, the first object referenced will never run out of cards. If the last card is ever drawn, the interpreter will automatically take cards from the second object, shuffle them, and deal them back into the first object.

If the `leaving x` form is used, then the top `x` cards will be left on the second object; the rest will be taken, shuffled, and dealt back into the first object.

The references must refer to objects of type `CardList`, or an error will occur.

5.3.5 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.

5.3.6 deal

The deal statement moves cards from one place to another.

```
dealstmt: "deal" some choice "from" ref "to" ref "."
    some: "all" | expr
  choice: ("of" ref "choice")?
```

Example:

```
deal 1 of p choice from p->hand to discardpile.
deal all from deck to drawpile.
```

Semantics:

Some number of cards is moved from the first referenced object to the second referenced object. If an expression is given, it denotes the number of cards that will move; otherwise, if `all` is given, all cards are moved.

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

Cards are moved one at a time from the top of the source to the bottom of the destination.

If the `of choice` version is used, the reference given here must be of type `Player` or an error will occur. That player will be shown all of the cards available for moving and he or she will be able to interactively decide on which ones will be moved. The order of the unmoved cards will remain unchanged. The order in which the moved cards are moved will be the order in which they are selected by the player.

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

If the referenced object that is the source of the deal has an `auto restock` on it, the restock happens after the deal completes.

5.3.7 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.

5.3.8 for

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

```
forstmt: "for" id "in" ref block
       | "for" id "in" ref "starting" "at" ref 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 at the end of the block.

If the `starting at` form is used, then the given reference must exist 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 other form is used, traversal starts at the beginning of the list and ends at the end of the list.

5.3.9 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 evalues to `True`. At that point, the corresponding block is executed and the rest of the statement is ignored.

If no expression evalues 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.

5.3.10 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.

5.3.11 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 appears in the same action.

5.3.12 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 identifier that names the temporary object 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.

5.3.13 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 or team. The message is displayed so only the referenced player(s) see it.

5.3.14 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.

5.3.15 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.

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` or an error will occur.

5.3.16 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.

5.3.17 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.

5.3.18 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.

5.3.19 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`.

5.4 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.

5.4.1 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.

5.4.2 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.

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

5.4.3 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 `object types` and `list types`, references are followed and structures are compared.

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`.

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

5.4.4 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.

5.4.5 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.

5.4.6 defined

The defined expression tests whether a reference object 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 `True`. Otherwise, it evaluates to `False`.

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

5.4.7 Card Expressions

Card expressions provide a concise way to refer to cards.

```
cardexpr: rank "~" suit
    rank: "%" | expr | rank ".." rank | rank ("," rank)*
    suit: "%" | expr | 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.

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`.

5.4.8 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 type.

5.4.9 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", or "Diamonds"
Card	"Jack of Clubs", etc
Player	"Player (name)"
Team	"Team (player names)"

5.4.10 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 preceeding player in the list.

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

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