replay

Final Report

Eric Bolton - edb2129

Contents

1 Introduction

`replay` is an imperative programming language designed to make strategies in repeated games easier to represent and analyze. It draws inspiration from three papers in game theory, Abreu and Rubinstein (1988), Miller (1987), and Rubinstein (1986), which first formalized the use of automata theory for the analysis of games, an idea that was first suggested by Aumann (1981). As such, `replay` provides a framework for defining strategies as finite automata (Moore machines). In addition, it enables game payoffs to be specified functionally, simplifying the process of defining complex games. Finally, it provides tools central to the genetic algorithm introduced by Holland (1975), which has proven instrumental in the analysis of strategies in repeated games.

1.1 Motivation

Game theory is treacherous for humans. Looks can be deceptive: even the simplest games are tricky to analyze, and it's easy to make silly mistakes. Repeated versions of games compound these problems, adding layers of complexity and confusion. A natural answer is to turn to a computer program for help. However, programming languages embraced by game theorists, such as `MATLAB`, `Python`, or `R`, do little to simplify the task at hand, as they weren't natively designed for these analyses.

`replay` is an imperative programming language that makes finitely and infinitely repeated games easier to represent and analyze. The syntax provides a clear and straightforward way to specify the various parameters of simple games: basic strategies, payoffs, as well as more complicated variants of strategies. Then, the language enables simple, yet powerful analyses on these games. Finally, `replay` makes it possible to have players play simple games repeatedly, enabling analyses of more complicated games - including finitely and infinitely repeated games.

1.2 Background

A repeated game is a version of a simple game that is played more than once. Oftentimes, the optimal strategy in the repeated version of a game is radically different from the one in the base game. For example, in the Prisoner's Dilemma, optimal play dictates that the players do not cooperate. But in the repeated version of the game, depending on the number of repetitions and the value players attribute to their future payoffs, the threat of punishment could encourage collaboration.

Another feature of games is information. Players may know differing amounts about what others in the game have played. An example is the distinction between simultaneous games and sequential games: in a simultaneous game such as Rock-Paper-Scissors, Player 2 does not know what Player 1 has played when she makes her move. This is not the case in a sequential game like Chess.

Finally, while games are typically described by the payoffs yielded by pure strategies, players may choose not to play a pure strategy. That is, they may play different moves with different odds. This is important in games of imperfect information such as simultaneous games. For example, in Rock-Paper-Scissors, it's a bad idea to always play Rock; it is better to attribute a probability of $\frac{1}{3}$ to each move.

2 Tutorial

This tutorial will focus mainly on the sample program `test-genetic.rpl`, which illustrates many of the language's key features. It is included in the `replay/tests` folder. This program implements a version of the genetic algorithm to determine effective strategies for the Repeated Prisoner's Dilemma.

2.1 Functions and scope

`replay` programs are a collection of statements and functions, in any order. This requires some nuance when it comes to scope: functions cannot make use of variables that haven't yet been defined, and their scope only reaches the following statements. This avoids potential confusion about what might happen if a function calls a global variable that has been defined using a call of that function. However, function scope does reach the body of all other functions, enabling mutual recursion.

`test-genetic.rpl` declares makes use of two functions; `sort` and `randindex`. `sort` makes use of the Quicksort algorithm to rank a list of players based on their accumulated payoffs. Since its type is void, it cannot have a return statement.

```
void sort(Player[] players, float[] payoffs) {
  quickSort(players, payoffs, 0, players.len-1);
}
```

`quickSort` is recursive, and uses while and if statements to sort the two lists. It accesses the arrays in memory, and performs exchanges when necessary. Array accesses whose index exceeds the size of the array will print an ”Index out of bounds” error message.

```
void quickSort(Player[] players, float[] payoffs, int lo, int hi) {
  int i = lo;
  int j = hi;

  float pivot = payoffs[(lo+hi)/2];

  while (i <= j) {
    while (payoffs[i] < pivot) {
      i = i + 1;
    }
    while (payoffs[j] > pivot) {
      j = j - 1;
    }
    if (i <= j) {
      Player temp1 = players[i];
      players[i] = players[j];
      players[j] = temp1;
      float temp2 = payoffs[i];
      payoffs[i] = payoffs[j];
      payoffs[j] = temp2;
      i = i + 1;
```

```
      j = j - 1;
    }
  }
  if (lo < j)
    quickSort(players, payoffs, lo, j);
  if (i < hi)
    quickSort(players, payoffs, i, hi);
}
```

2.2 Random numbers

The function `randindex` returns a weighted index from a list of sorted floating point numbers. It does so by computing the sum of floating point numbers on the list, multiplying it by a random number between 0 and 1; the `rand` keyword. `rand` is a pseudo-random number computed using the current time to seed the `srand` random number generator in `C`'s library.

```
/* Return weighted random index */
int randindex(float[] list) {
  float r;
  float sum = 0.0;
  for p in list { sum = sum + p;}
  r = rand * sum;
  for i in [0:list.len-1] {
    /* Accumulate list values, returning the index when r changes sign */
    if (r > 0) {
      r = r - list[i];
      if (r < 0) { return i; }
    }
    else {
      r = r - list[i];
      if (r > 0) { return i; }
    }
  }
  println(list.len-1);
  return list.len-1;
}
```

These functions will prove useful for implementing the genetic algorithm.

2.3 Games

Next, `test-genetic.rpl` uses one of the central features of `replay`: the Game constructor. First, it creates a version of the Prisoner's Dilemma.

```
strat {C, D};
/* Stored as a matrix with a header */
Game pd = Game[2 | {
  (C,C) -> (-1,-1)
| (C,D) -> (-6,0)
```

```
| (D,C) -> (0,-6)
| (D,D) -> (-5,-5)
}];
```

In this game, two prisoners face one year of prison if they cooperate by not denouncing each other. However, if one stays silent while the other confesses, she faces six years of prison, while the other faces none. If both denounce, they get five years.

The strat declaration formalizes the prisoner's two choices; C, for cooperate, is assigned a value of 0, while D, for defect, is assigned a value of 1. Within the game declaration, the number of players playing the game is indicated by the number of values specified in the comma-separated lists. If any of the lists differ in length, an illegal outcome error is raised.

While the strat declaration defines only two moves, the game isn't aware of that. Therefore, it is required that the number of moves be included in the game declaration. As such, the Game constructor has two fields; one for the number of moves, and the second for the list of game payoffs. The constructor automatically sets any payoffs that haven't been explicitly defined to 0.

It's also worth noting that had the game payoffs been omitted, and only one field been defined, this constructor would have been interpreted as an array of type game; `Game[2]` defines an array of two games.

2.4 Strategies

`replay` also enables easy definition of strategies using the Strategy constructor. This constructor requires three fields; the number of moves, the number of strategy states, and the list of states. Strategies are represented as finite automata, where each state dictates what move the strategy outputs. The moves that are played in a given round of the game determine what state the strategy will go to for the next round.

```
/* Stored as an array of transitions with a header */
Strategy grim1 = Strategy[2 | 5 | {
  cooperate: C, (_,D) -> defect;
  defect:    D, (_,_) -> defect;
}];

Strategy grim2 = Strategy[2 | 5 | {
  cooperate: C, (D,_) -> defect;
  defect:    D, (_,_) -> defect;
}];
```

Here, two versions of the "grim trigger" strategy are defined, one for a theoretical player one, and one for a theoretical player two. In the cooperate state, both players will choose to cooperate. However, the moment either player defects, the other will choose to never cooperate again; a rather unforgiving punishment. The constructor interprets wildcard `'_'` operators as being interchangeable with any move.

In addition, since each constructor is called with 5 states, and only two are formally defined, `replay` automatically fills in the remaining states. The default is for each state to map back to itself, while outputting the move 0, in this case C.

2.5 Players

Players store a strategy, which is then used to determine information such as a player's chosen move, her next state, and the payoff she receives.

```
/* 10 structures, indexed by i */
Player[] players1 = Player[10];
Player[] players2 = Player[10];
float[] payoffs1 = float[10];
float[] payoffs2 = float[10];

for i in [0:players1.len - 1] {
  players1[i] = Player[grim1];
  players2[i] = Player[grim2];
  payoffs1[i] = 0.0;
  payoffs2[i] = 0.0;
}
```

This illustrates how constructors can be interpreted differently depending on the input: when an int is provided, an array of players is created, but when a strategy is provided, an individual player is created. Player constructors can also have one additional float field defined: their discount factor, also referred to as delta. Any specification that does not match these formats will output an Illegal object error.

2.6 For loops

Next, `replay` defines for loops using a variable and either a range or array. The variable iterates through the elements of the array as the for loop runs. This for loop indicates that the genetic algorithm is to run 50 times:

```
/* 50 generations */
for t in [1:50] {
...
}
```

2.7 Attributes

Next, the program uses another feature of `replay`: attributes. The major types - Game, Player, and Strategy - all have several useful attributes that can be used to obtain useful information about each type.

```
  /* Pit player ones against player twos */
  for i in [0:9] {
    for j in [0:9] {
      /* Reset both players */
      players1[i] = players1[i].reset;
      players2[j] = players2[j].reset;
      /* Play 20 rounds */
      for r in [1:20] {
        players1[i], players2[j] % pd; /* Updates payoffs and state */
      }
      payoffs1[i] = payoffs1[i] + players1[i].payoff;
      payoffs2[j] = payoffs2[j] + players2[j].payoff;
```

```
13     }
14   }
```

In this listing, the reset and payoff attributes are referenced. The reset attribute simply sets a player's current state, payoff, and number of rounds played back to 0. The payoff attribute indicates the accumulated payoffs of the player. The aformentioned discount factor is a formalization of diminishing rates of return. If a player has a delta that is not equal to one, then each time the player receives a payoff, it will be multiplied by this delta raised to the power of how many rounds that player has played.

2.8 Play (%)

In line 9 of the previous listing, the play operator is used:

`players1[i], players2[j] % pd; /* Updates payoffs and state */`.

This operator involves a comma-separated list of players, and the game they will be playing. The position of each player in the list matters; it is used to determine which player gets what payoff. All players see the same list of moves, and use it to decide what their next state will be.

2.9 Crossover (#) and Mutate (~)

Next, our program introduces the crossover and mutate operators, which are used for the genetic algorithm.

```
/* Form a new population of 10 structures */
/* Top 6 from old pop, 4 new ones generated as children. */
/* Parents selected randomly, but with more weight towards high payoffs */
sort(players1, payoffs1);
sort(players2, payoffs2);
for i in [0:1] {
  int m1 = randindex(payoffs1);
  int f1 = randindex(payoffs1);
  /* Ensure parents are different */
  if (m1 == f1) {
    if (f1 == 9) { m1 = m1-1; }
    else        { f1 = f1+1; }
  }
  int m2 = randindex(payoffs2);
  int f2 = randindex(payoffs2);
  if (m2 == f2) {
    if (f2 == 9) { m2 = m2-1; }
    else        { f2 = f2+1; }
  }
  players1[6+2*i] = Player[players1[m1].strategy];
  players1[6+2*i+1] = Player[players1[f1].strategy];
  players2[6+2*i] = Player[players2[m2].strategy];
  players2[6+2*i+1] = Player[players2[f2].strategy];

  /* Form new children with crossover */
  players1[6+2*i] # rand # players1[6+2*i+1];
  players2[6+2*i] # rand # players2[6+2*i+1];
```

```
28
29      /* Mutate children */
30      players1[6+2*i] ~ 0.2;
31      players2[6+2*i] ~ 0.2;
32      players1[6+2*i+1] ~ 0.2;
33      players2[6+2*i+1] ~ 0.2;
34    }
35
36    /* Reset payoffs before moving on to next generation */
37    for i in [0:9] {
38      payoffs1[i] = 0.0;
39      payoffs2[i] = 0.0;
40    }
```

randindex is used to determine pairs of ”parents” that will be used to create new ”child” strategies. The crossover (lines 25 and 26) is an operation in which sections of each player’s strategy are swapped, one for the other. It takes three arguments; two players and one float. The float determines what fraction of each player’s information will be swapped.

2.10 String concatenation (ˆ)

Finally, we can print the strategies of the number one ranked players of each array of players. The concatenation operator enables several strings to be glued together.

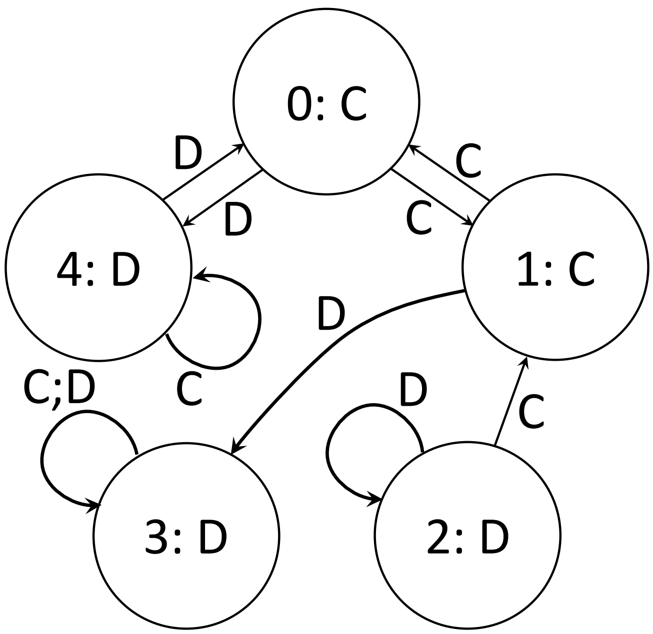
```
/* Print winning strategies */
println(players1[0].strategy ^ players2[0].strategy);
```

One iteration of this program outputted the following final strategy for player 1:

```
State 0: play 0
( 0 0 ) -> state 1
( 0 1 ) -> state 4
( 1 0 ) -> state 3
( 1 1 ) -> state 0
State 1: play 0
( 0 0 ) -> state 0
( 0 1 ) -> state 3
( 1 0 ) -> state 3
( 1 1 ) -> state 3
State 2: play 1
( 0 0 ) -> state 0
( 0 1 ) -> state 3
( 1 0 ) -> state 1
( 1 1 ) -> state 2
State 3: play 1
( 0 0 ) -> state 0
( 0 1 ) -> state 2
( 1 0 ) -> state 3
( 1 1 ) -> state 3
State 4: play 1
```

```
( 0 0 ) -> state 4
( 0 1 ) -> state 2
( 1 0 ) -> state 4
( 1 1 ) -> state 1
```

A representation of this as a finite automaton would look like this:



All transitions are labeled with the moves played by player 2. Transitions where player 1's move would have to differ from the state's output move were omitted for simplicity; these routes would never be used. State 2 is inaccessible under this schematic.

Some recognizable behaviors are exhibited: as long as the opponent cooperates, player 1 cooperates. But once the opponent defects, there is a chance player 1 will move to a state in which he defects forever; state 3. Since there's also a chance he'll go to state 4, this strategy is slightly more forgiving than the grim trigger punishment. The more times the genetic algorithm is repeated, the more forgiving the winning strategies tend to become.

3 Language Reference Manual

3.1 Lexical conventions

3.1.1 Tokens

There are five different kinds of tokens in `replay` : identifiers, keywords, literals, operators, and punctuation.

3.1.1.1 Variables

Variables begin with a letter, followed by any number of letters, digits, or underscores '_'.
The underscore '_' by itself denotes a wildcard in the context of moves (see 3. Expressions).

3.1.1.2 Keywords

The following identifiers are reserved for use as keywords:

```
int       if
bool      else
float     for
string    in
void      return
Game      true
Strategy  false
Player    strat
          rand
```

3.1.1.3 Literals

There are literals for each type, as follows:

- `int`: A sequence of digits.
- `bool`: A ’`true`’ or a ’`false`’
- `float`: Two possibilities:
  - An integer part, a decimal point ’`.`’, a fraction part, and an exponent. The integer and fraction parts consist of a sequence of digits. The exponent consists of an `e`, followed by an optional sign `-` or `+`, and a sequence of digits. The integer and fraction parts are both optional, but at least one of the two must be present. Likewise, the decimal point and the exponent are both optional, but at least one must be present.
  - A ’`rand`’ keyword, which gets interpreted as a random floating point number between 0 and 1 at run time.
- `string`: Any number of characters, delimited by quotes ’`" "`’.

3.1.1.4 Operators

There are 18 operators, as follows:

```
+     ==    &&
-     !=    ||
*     >     !
/     <
=     >=
->    <=
%
#
~
^
```

3.1.1.5 Punctuation

The following characters are used as punctuation:

```
(   )
[   ]
{   }
:   ;
.   ,
|
```

3.1.2 Separators

Comments and whitespace are ignored by the scanner, except to serve as separators.

- Comments are delimited by `/*` and `*/`.
- ' ' (space), '\t' (tab), '\r' (carriage return), and '\n' (newline) are treated as whitespace.

3.2 Expressions

3.2.1 Operations

An expression can be an operation, which consists of parentheses, expressions, unary operators, and binary operators.

3.2.1.1 Unary operators

Unary operators group right-to-left and have higher precedence than binary operators. They behave as follows:

- `-`*expression*: Unary negation. The result is the negative of the expression, without changing its type. It is applicable only to expressions of type `int` or `float`.
- `!`*expression*: Logical negation. The result is `true` if the expression is `false`. Conversely, it is `false` if the expression is `true`. It is applicable only to expressions of type `bool`.

3.2.1.2 Binary operators

Binary operators group left-to-right. Here they are, sorted from highest to lowest by precedence:

```
*    /
+    -
>    >=   <   <=
==   !=
&&
||
^
```

They behave as follows:

Arithmetic operators

- *expression*`*`*expression*: Multiplication. The result is `float` if one of the two operands is `float`. The result is `int` if both operands are `int`. If one operand is `float` and the other is `int`, the `int` gets converted to `float`. Multiplication is applicable only to expressions of type `int` or `float`.
- *expression*`/`*expression*: Division. Result and operand types behave the same way as with multiplication.

- *expression*`+`*expression*: Addition. Types behave the same way.
- *expression*`-`*expression*: Subtraction. Types behave the same way.

Relational operators

- *expression*`>`*expression*: Greater than.
- *expression*`<`*expression*: Less than.
- *expression*`>=`*expression*: Greater than or equal to.
- *expression*`<=`*expression*: Less than or equal to.

The result of these operators is `true` if the relation is true, and `false` if it is false. If one operand is `float` and the other is `int`, the `int` gets converted to `float`. Relational operators are applicable only to expressions of type `int` or `float`.

Equality operators

- *expression*`==`*expression*: Equal to.
- *expression*`!=`*expression*: Not equal to.

These behave the same as the relational operators, except that they are applicable to more types: any pair of operands that are of the same type. Additionally, if one operand is `float` and the other is `int`, the `int` gets converted to `float`.

Boolean operators

- *expression*`||`*expression*: Or. The result is `true` if either operand is `true`, and `false` otherwise. It is applicable only to operands of type `bool`.
- *expression*`&&`*expression*: And. The result is `true` if both operands are `true`, and `false` otherwise. Like `||`, it is applicable only to operands of type `bool`.

Concatenation operator

expression`^`*expression*. The result is the two operands concatenated to each other. It is applicable to operands of type `String`, `int`, `bool`, `float`, and `Strategy`. Any operands that aren't of type `String` are automatically converted to a string using their string attribute.

3.2.1.3 Parentheses

`(`*expression*`)`: The result is simply that of the expression enclosed in parentheses.

Note: special operators

`% # ~`: Play, cross, and mutate: Because these operators behave differently from the operators cited above, they are treated as statements. (See Section 5.2).

3.2.2 Literals and identifiers

3.2.2.1 Literals

An expression can be any of the literals specified in Section 2.1.3. The result is the type of the literal.

3.2.2.2 Identifiers

An expression can be an *identifier*, which in turn can be any of the following:

- *variable*: A variable, as specified in section 2.1.1.
- *variable*`(`$actuals_{opt}$`)`: The result of calling the function *variable* with parameters specified by the list of comma-separated values $actuals_{opt}$.
- *variable*`[`*expression*`]`: The value at index *expression* of the array *variable*. *expression* must be an `int`, while *variable* must be an array. (See Section 3.3 Non-primitive types)
- *identifier*`.`*variable*: The attribute *variable* of *identifier*. (See Section 3.4 Attributes)
- *identifier*`.`*variable*: The attribute *variable* of *identifier*. (See Section 3.4 Attributes

3.2.3 Non-primitive types

An expression can also be a `Strategy`, a `Game`, a `Player`, or an array, the non-primitive types of `replay`.

3.2.3.1 Strategies

An expression can be:

- `Strategy[`*params*`]`: A Strategy. *params* is a list of *expression*s separated by `|`'s. In this case, it must consist of the number of states (an `int`), the number of moves the strategy is based on (an `int`), and a list of *states* (see below). The numbers are needed to gauge memory requirements. This expression has type `Strategy`.
- {*states*}: A list of *state*'s, enclosed by braces. This is the last *expression* required by the `Strategy[`*params*`]` constructor.

States

A *state* is specified as follows:

variable`:`*expression*`,`*transitions*`;`

variable is the name of the state, and is set to be an `int`. *expression* corresponds to the move the state outputs, which must be of type `int`. *transitions* is a list of transitions, separated by bars.

Transitions

A *transition* is specified as follows:

`(`*moves*`)->`*variable*

moves is a comma-separated list of either `int`'s or wildcards '_'. *variable* corresponds to the name of the state, and must be an `int`.
To summarize, here is an example Strategy expression:

```
Strategy[10 | 2 | {
  cooperate:  C, (_,D) -> defect
                | (_,C) -> cooperate;
  defect:     D, (_,_) -> defect; }]
```

This specifies a strategy with 10 states, and 2 possible moves. States don't need to have every possible transition specified; a state will by default map back to itself. Furthermore, not all states need to be specified: states will by default output whatever move is denoted by 0, and map back to themselves.

3.2.3.2 Games

An expression can be:

- `Game[`*params*`]`: A Game. In this case, *params* must consist of the number of moves (an `int`) in the game and a list of *outcomes* (see below). This expression has type `Game`.
- {*outcomes*}: A list of *outcome*'s, enclosed by braces and separated by bars `|`.

Outcomes

An *outcome* is specified as follows:

`(`*moves*`)->(`*payoffs*`)`

payoffs is a comma-separated list of *expression*'s, which must be of type `int`.
To summarize, here is an example Game expression:

```
Game[2 | { (C,C) -> (-1,-1)
| (C,D) -> (-5,0)
| (D,C) -> (0,-5)
| (D,D) -> (-3,-3); }]
```

3.2.3.3 Players

An expression can also be a Player, specified as follows:

`Player[`*params*`]`

In this case, the first element of *params* must be a `Strategy`, corresponding to the Player's strategy. Then, the user can optionally specify an additional parameter of type `float`. This corresponds to the Player's delta: how much value the player attributes to payoffs acquired in future rounds.
This expression has type `Player`.
A Player tracks the state it is in, its accumulated payoff, and how many times it has played. Each time it "plays" in a Game, it adds the payoff it received, multiplied by its delta raised to the power of how many times it has played.
To summarize, here is an example Player expression:

```
Player[grim | 0.5 | 0.01 | 0.01]
```

3.2.3.4 Arrays

Finally, an expression can be arrays, specified as follows:

- *type*`[`*params*`]`: A *type* array. When *params* has only one value of type `int`, this construction always specifies an array.
- `[`$expression_1$`:`$expression_2$`]`: An integer array of size $expression_2$ - $expression_1$, whose values range from $expression_1$ to $expression_2$. Both *expression*'s are required to be `int`'s.

3.2.4 Attributes

`replay`'s non-primitive types have built-in attributes:

- `string`:
  - `len`: The length of the string. (`int`)
- Arrays:
  - `len`: The length of the array. (`int`)
- `Strategy`:
  - `size`: The number of states in the Strategy. (`int`)
  - `moves`: The number of moves the Strategy can play. (`int`)
- `Game`:
  - `players`: The number of players in the game. (`int`)
  - `moves`: The number of moves each player can play. (`int`)
- `Player`:
  - `strategy`: The strategy of the Player. (`Strategy`)
  - `state`: The state of the Player's Strategy the Player is in. (`int`)
  - `rounds`: The number of rounds the Player has played. (`int`)
  - `delta`: The delta of the Player. (`float`)
  - `payoff`: The accumulated payoff of the Player. (`float`)
  - `reset`: Resets the accumulated payoff and the number of moves to 0. (`Player`)

`Strategy`, `int`, `bool`, and `float` also have a `string` attribute, which provides a string representation for the type.
Users cannot specify new attributes, nor can they change their value.

3.3 Declarations

There are three types of declarations: function declarations, variable declarations, and the `strat` enumerator.

3.3.1 Functions

A function declaration is specified as follows:

$$\textit{type variable } \texttt{(}\textit{formals}_{opt}\texttt{)}\{\textit{statements}\}$$

type and *variable* specify the type and name of the function. $\textit{formals}_{opt}$ specify the arguments the function takes, just as $\textit{actuals}_{opt}$ specify the arguments passed in a function call. Finally, *statements* is a list of any number of *statement*'s, corresponding to the body of the function. (See Section 5. Statements) A function's last statement must be `return` statement, whose return type must correspond to the type of the function.

3.3.1.1 Built-in functions

Certain function names are reserved. They serve the following two purposes:
Printing

- `print(`$\textit{actuals}_{opt}$`)`: Takes one argument, converts it to a `string` using its type's `string()` attribute, then prints it out. (`void`)
- `println(`$\textit{actuals}_{opt}$`)`: Does the same, then prints a newline character. (`void`)

3.3.2 Variables

A variable declaration can take two forms:

- *type variable* `=` *expression*: This declares a *variable* of type *type*, and initializes it with the value *expression*. The types of the *expression* and the *variable* must match.
- *type variable*: This declares a *variable* of type *type*, and initializes it to a default value.

3.3.2.1 Arrays

In the case of an array declaration, the *type* is followed by `[]`:

- *type*`[]`*variable* `=` *expression*
- *type*`[]`*variable*

3.3.3 `strat` enumerator

Finally, a user can declare a set of moves using the `strat` enumerator:

`strat` {*variables*}

variables is a comma-separated list of n *variable*'s, which get declared with type `int` by this construction. They are initialized with values 0 through n depending on their position in the list.

3.4 Statements

There are many types of statements. A program consists of a list of statements and function declarations, in any order.

3.4.1 Variable or strat declaration

A statement can be either a variable or `strat` declaration, followed by a semicolon '`;`'

3.4.2 Play, Mutate, and Crossover

A statement can be any of the operations on Players.

- *identifiers*`%`*expression*`;`: Play. *identifiers* is a comma-separated list of *identifier*'s, which must be `Player`'s. *expression* must be a `Game`. This operator pits the Players against each other for a round of the Game, updating their payoffs and Strategy states.
- *identifier*`~`*expression*`;`: Mutate. *identifier* must be a `Player`, and *expression* must be a `float`. This operation acts on the bit representation of the `Player`'s strategy, changing each bit with a probability specified by the `float`.
- $identifier_1$`#`*expression*`#`$identifier_2$`;`: Crossover. $identifier_1$ and $identifier_2$ must be `Player`'s, and *expression* must be a `float`. This operation acts on the bit representations of the `Player`'s strategies, crossing the two representations at the position indicated by the `float`.

Now for an illustration of the effect of the following crossover statement's effect: `p1 # 0.75 # p2;`
Let P_1 be the bit representation of `p1`'s strategy, and P_2 be the bit representation of `p2`'s strategy. Say before crossover, we have:
$P_1 = 00000010$ and $P_2 = 00000011$
After crossover, we would have:
$P_1 = 00000011$ and $P_2 = 00000010$. The bits following position 6, or 0.75 of the way into the bit representation, have been swapped.

3.4.3 Assignment

A statement can be an assignment statement.

identifier`=`*expression*`;`

The *identifier* must be an array entry or a *variable*; it cannot be an attribute or a function call. The *expression* must be of the same type as the *identifier*. The value of the *expression* gets assigned to the *identifier*.

3.4.4 Side-effect Call

If a function call has void type, it can't be assigned to any variable. In addition, some function call results may be unwanted. As a result, a statement is allowed to be simply just a function call. *variable*`(`$actuals_{opt}$`);`: The result of calling the function *variable* with parameters specified by the list of comma-separated values $actuals_{opt}$.

3.4.5 Return

A statement can be a return statement.

`return` $expression_{opt}$`;`

If $expression_{opt}$ is left empty, this returns a `void`. Otherwise, this returns an expression of type $expression_{opt}$.

3.4.6 `if`, `else`

A statement can be an if-else statement.

- `if (`*expression*`)` *statement*: *statement* is executed if and only if *expression* evaluates to `true`. *expression* must be a `bool`.

- `if (`*expression*`)` $statement_1$ `else` $statement_2$: If *expression* evaluates to `false`, $statement_2$ is executed.

3.4.7 `for`, `in`

A statement can be a for-loop.

`for` *variable* `in` *expression statement*

expression must be an array. For each iteration of the for-loop, *statement* is executed and *variable* takes on the next value of the array, until the end of the array is reached.

3.4.8 List

Finally, a statement can be a brace-enclosed list of statements:

{*statements*}

3.5 Scope

A *variable*'s scope reaches any line following their declaration. If they are declared within braces {}, they have local scope and cannot be reached outside of the braces.
If a *variable* is declared as part of a for-loop statement, its scope is limited to that for-loop's statement.
If an undeclared *variable* is used in a set of moves to define payoffs functionally within a list of *outcomes*, its scope reaches only the payoffs that directly follow the arrow of the *outcome* it is used in.
Finally, when a *variable* is used to name a state in a Strategy, its scope extends to all states in the strategy, whether they precede or follow it.

4 Project Plan

4.1 Process

4.2 Planning

At the outset of the project, I defined major milestones based on the project due dates. I also drew inspiration from the milestones students had defined in previous years to put together a coherent plan. I used Evernote Web to track daily actions and milestones, and Microsoft Excel to track broader milestones.

4.3 Specification

I intended my language to be useful for analyzing and solving simple Game Theory problems. My initial language proposal was largely informed by what I knew about Game Theory at the time. Then, I turned to literature in evolutionary game theory for additional inspiration. Based on my readings, I decided to build a language focused on defining strategies as finite automata, that would also include an intuitive framework for testing such strategies using simple games and players. This informed the grammar I developed for the Language Reference Manual. As I built the semantic and type checking component, I made small adjustments to the `replay` specification based on what was feasible in OCaml and LLVM.

4.4 Development

I developed components of `replay` roughly according to the order in which they are used by the compiler. I built the scanner first, then the parser and abstract syntax tree concurrently. I also developed the code generator and semantic checker simultaneously, so that Hello World would run before the semantic and type checking was fully complete.

4.5 Testing

I tested gradually more complicated programs as I built the semantic analyzer, ensuring that the correct ones were accepted, while the wrong ones were rejected with the right error message. I automated this testing using a modified version of the `testall.sh` provided by Professor Edwards. Once I made finishing touches to the code generator, I began testing the LLVM code generated by the correct programs, checking that this code ran as expected.

4.6 Style Guide

4.6.1 Comments

Each file begins with a comment indicating the file name, the creation date, the course name, the language name, and the author. All functions are preceded by a comment indicating its purpose and return type. Additional comments are added where needed for clarity.

4.6.2 Indentation and spacing

Tabs were used as indentation, following these rules:

- **`let`, `and`, `in`:** The contents of each `let...in` block begins on the same line as the `let`. Then, each new line is indented relative to the opening `let`. `and` must start a new line, with the same indentation as the opening `let`. For multiple line blocks, the `in` is on a new line. For single line blocks, the `let` and `in` are on the same line.

- **`match` and `function` statements:** The first pattern begins on a new line, indented twice relative to the preceding code. Then, each following option begins on a new line, indented once, so that each pattern that follows a '`|`' character aligns with the first pattern.

- **Functions:** The contents of each function body begins on the same line as the `->`. Then, each new line is indented relative to the opening `->`.

- **Semantic actions and tokens:** For single line semantic actions and tokens, tabs are added before the opening bracket to align with brackets above. For multiple line semantic actions and tokens, the opening bracket begins on a new line.

- **Operators:** Operators are preceded and followed by a space.

For each block, a new line begins at the last white space that precedes character 80.

4.6.3 Naming conventions

- **Functions:** Functions are named according to their purpose, without using abbreviations. Underscores are used to separate multiple words.
- **Variables:** Variable names follow the same rules as function names.
- **Function and pattern matching arguments:** Arguments are abbreviated using the first letter of their type, unless a more descriptive name is warranted.

4.7 Project timeline

Here is the timeline I defined at the outset of the project.

Date	Milestone
September 28	Proposal complete
October 26	Scanner and Parser
October 26	Language Reference Manual complete
November 15	Semantics / typechecking complete
November 20	Hello World runs
November 30	Code generation complete
December 15	Regression testing, debugging complete
December 20	Final report complete

4.8 Software Development Environment

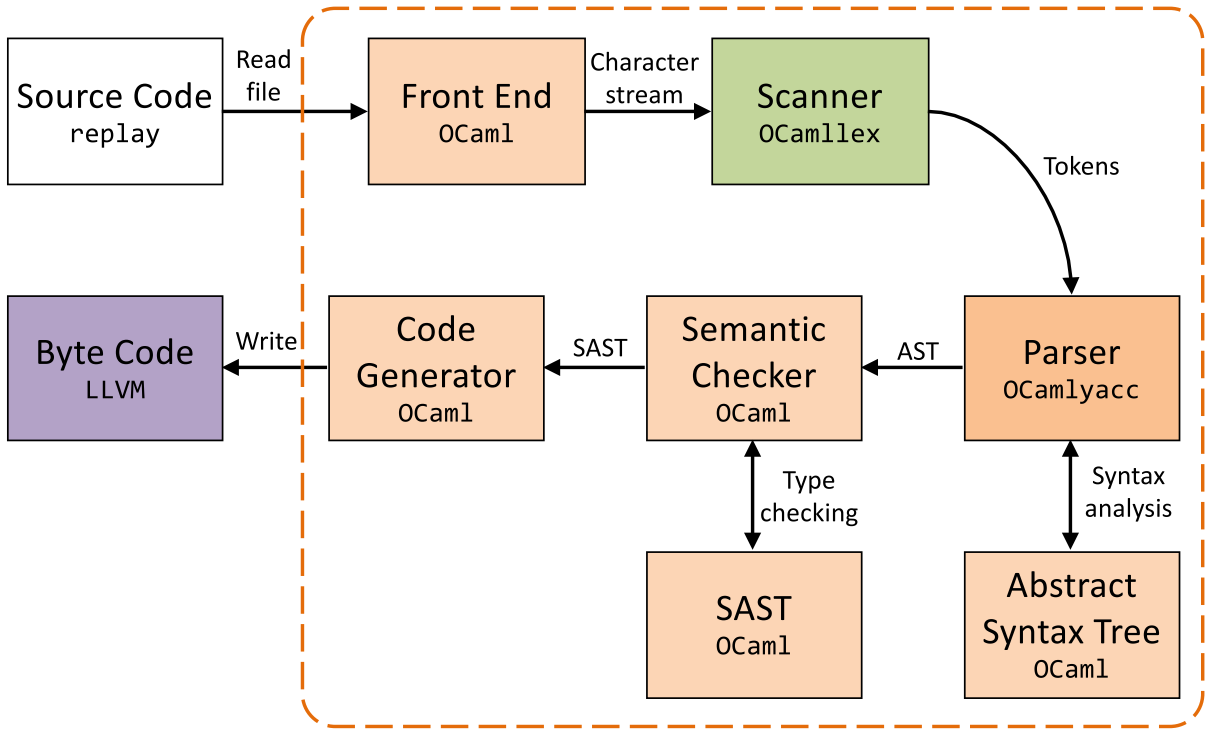
- **Languages:** The compiler was built using OCaml version 4.03.0, as well as the OCamlyacc and OCamllex lexer and parser generators. The target code was generated in LLVM version 3.7.1. The regression testing was coded using a shell script.
- **Tools:** All code was written using Atom 1.12.17, with toroidal-code's language-ocaml package for syntax support.

4.9 Project Log

Here are the dates at which mjaor milestones were accomplished.

Date	Milestone
September 25	Language defined
September 28	Proposal complete
October 20	Main language characteristics finalized
October 24	Scanner complete
October 25	Parser and preliminary AST complete
October 26	Language Reference Manual complete
November 16	Pretty printer complete
November 16	Compiler front end complete
November 19	Hello World runs
December 15	Semantics and typechecking complete
December 20	Code generation complete
December 22	Regression testing, debugging complete

5 Architectural Design



The architecture relies on a front end to read in the source code, then call the various components in order. First, the scanner translates the source code into tokens. Next, the parser to performs syntax analysis on those tokens, and builds an abstract syntax tree. The semantic checker translates the abstract syntax tree into a semantic abstract syntax tree, which it uses to check types. Finally, the code generator translates the semantic abstract syntax tree into LLVM byte code.

6 Test Plan

I designed a suite of 26 failing and 28 succeeding `.rpl` files, and checked each for correct outputs. In particular, I extensively tested the function of the Game, Strategy, and Player objects. For example, the following file; `test-strategy.rpl`, was designed to test that all Strategy fields were probably initialized, in particular that each wild card was correctly interpreted.

```
strat {A, B, C};
Strategy s = Strategy[3|4|{ state0: A, (A,C) -> state1
                                   | (A,B) -> state2;
                           state1: C, (_,_) -> state2;
                           state2: B, (B,_) -> state1
                                   | (B,C) -> state0;}];

```

```
println(s.players);
println(s.moves);
println(s.size);
println(s);
```

The fourth state was intentionally omitted from the declaration, to test that the state was correctly initialized. When run, it produces the following correct output:

```
2
3
4
State 0: play 0
( 0 0 ) -> state 0
( 0 1 ) -> state 2
( 0 2 ) -> state 1
( 1 0 ) -> state 0
( 1 1 ) -> state 0
( 1 2 ) -> state 0
( 2 0 ) -> state 0
( 2 1 ) -> state 0
( 2 2 ) -> state 0
State 1: play 2
( 0 0 ) -> state 2
( 0 1 ) -> state 2
( 0 2 ) -> state 2
( 1 0 ) -> state 2
( 1 1 ) -> state 2
( 1 2 ) -> state 2
( 2 0 ) -> state 2
( 2 1 ) -> state 2
( 2 2 ) -> state 2
State 2: play 1
( 0 0 ) -> state 2
( 0 1 ) -> state 2
( 0 2 ) -> state 2
( 1 0 ) -> state 1
( 1 1 ) -> state 1
( 1 2 ) -> state 0
( 2 0 ) -> state 2
( 2 1 ) -> state 2
( 2 2 ) -> state 2
State 3: play 0
( 0 0 ) -> state 3
( 0 1 ) -> state 3
( 0 2 ) -> state 3
( 1 0 ) -> state 3
( 1 1 ) -> state 3
( 1 2 ) -> state 3
```

```
( 2 0 ) -> state 3
( 2 1 ) -> state 3
( 2 2 ) -> state 3
```

To test the play operator, I wrote the file `test-play.rpl`:

```
strat {A, B};

Game g = Game[2 | {(A,A) -> (1, 1)}];

Strategy s = Strategy[2|2|{state0: A, (A,B) -> state1
                                    | (A,A) -> state1;
                           state1: B, (A,B) -> state0
                                    | (B,B) -> state0;}];

Player p1 = Player[s|0.5];
Player p2 = Player[s|1.0];

println(p1.payoff);
println(p1.state);
println(p1.rounds);

for i in [1:5] {
  p1, p2 % g;
}

println(p1.strategy);
println(p1.payoff);
println(p1.state);
println(p1.rounds);
```

Given how the parameters are set up, the players should alternate between playing (A, A) and (B, B) for the 5 rounds that they play. As such, player 1 earns a payoff stream of (1, 0, 1, 0, 1). Since her delta is 0.5, this translates to a sum of $1 \times 0.5^0 + 0 \times 0.5^1 1 \times 0.5^2 + 0 \times 0.5^3 + 1 \times 0.5^4 = 1.3125$. As expected, the output prints:

```
0
0
0
State 0: play 0
( 0 0 ) -> state 1
( 0 1 ) -> state 1
( 1 0 ) -> state 0
( 1 1 ) -> state 0
State 1: play 1
( 0 0 ) -> state 1
( 0 1 ) -> state 0
( 1 0 ) -> state 1
```

```
( 1 1 ) -> state 0

1.3125
1
5
\end{lslisting}

$~~~$ For automation, I used a slightly modified version of the {\ttfamily
    testall.sh} file provided by Professor Edwards. When run at the issue of the
    project, it produced the following output:

\begin{lstlisting}
Erics-MacBook-Pro:replay eb$ ./testall.sh
test-access... OK
test-arith1... OK
test-arith2... OK
test-arith3... OK
test-array... OK
test-assign1... OK
test-assign2... OK
test-cat1... OK
test-cat2... OK
test-cat3... OK
test-cross... OK
test-for... OK
test-for2... OK
test-func... OK
test-func2... OK
test-game... OK
test-gcd... OK
test-genetic... OK
test-hello... OK
test-if... OK
test-mutate... OK
test-play... OK
test-player... OK
test-rand... OK
test-range... OK
test-strat... OK
test-strategy... OK
test-while... OK
fail-args1... OK
fail-args2... OK
fail-assign... OK
fail-attr1... OK
fail-attr2... OK
fail-cat... OK
fail-char... OK
```

```
fail-construct... OK
fail-dup1... OK
fail-dup2... OK
fail-dup3... OK
fail-dup4... OK
fail-dup5... OK
fail-outcome... OK
fail-parse... OK
fail-player... OK
fail-return1... OK
fail-return2... OK
fail-return3... OK
fail-sdecl1... OK
fail-sdecl2... OK
fail-state... OK
fail-type1... OK
fail-type2... OK
fail-unterm... OK
fail-vdecl... OK
```

Fail cases were selected for each of the errors that `replay` recognizes and outputs during semantic checking. All test cases were designed at a granular level to rigorously test each individual component of `replay`'s grammar.

7 Lessons Learned

Designing and writing a compiler has been one of the most difficult and instructive projects I have ever undertaken. Along the way, I learned many valuable lessons. The first was to always expect the worst, especially with a project this size. A lot can go wrong, and Murphy's Law is unforgiving. Second, I learned that deadlines harbor a great deal of power. No matter how much time and effort was put in beforehand, my best and most productive work always came in the final days before a big deadline. Necessity is an unparalleled motivator. Finally, I learned how important it is fully carry out one's thoughts before moving on to something else: such large design projects do not take kindly to simply picking up where you left off!

8 References

Abreu, D. and Rubinstein, A. "The Structure of Nash Equilibrium in Repeated Games with Finite Automata." *Econometrica* 56 (November, 1988):1259-81.

Aumann, R. T. "Survey of Repeated Games." In R. T. Aumann et al., eds., *Essays in Game Theory and Mathematical Economics in Honor of Oscar Morgenstern* (Bibliographisches Institut, Zurich: Mannheim), 1981:11-42.

Holland, T. H. *Adaptation in Natural and Artificial Systems.* Ann Arbor, Michigan: The University of Michigan Press, 1975.

Ritchie, Dennis M. *C Reference Manual* Bell Telephone Laboratories. 1978

Rubinstein, A. "Finite Automata Play the Repeated Prisoner's Dilemma." *Journal of Economic Theory* 39 (June, 1986):83-96.

Miller, J. H. "The Coevolution of Automata in the Repeated Prisoner's Dilemma" SFI Working Paper: 1989–003

9 Appendix

```
(* File: replay.ml
 * Created: 11/16/2016
 *
 * COMSW4115 Fall 2016 (CVN)
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> *)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);  (* Print the AST only *)
                        ("-l", LLVM_IR); (* Generate LLVM, don't check *)
                        ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  try
    let ast = Parser.program Scanner.token lexbuf in
    let sast = Semant.check ast in
    match action with
      Ast -> print_string (Printer.string_of_program ast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
      Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
with e -> Printer.print_error lexbuf e; raise e
```

Listing 1: `replay.ml`

```
(* File: scanner.mll
 * Created: 10/24/2016
 *
 * COMSW4115 Fall 2016 (CVN)
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> *)

{
open Parser
open Lexing
open Printer
}
```

```
let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let num = digit+
let tail = letter | digit | ['_']
let identifier = letter tail*
let exp = ['e']['-' '+']? num
let floatnum = num ['.'] num? exp? | ['.'] num exp? | num exp
let newline = '\r' | '\n' | "\r\n"

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

    (* Keywords *)
    | "if"                 { IF }
    | "else"               { ELSE }
    | "while"              { WHILE }
    | "for"                { FOR }
    | "in"                 { IN }
    | "return"             { RETURN }
    | "true"               { TRUE }
    | "false"              { FALSE }
    | "strat"              { STRAT }
    | "rand"               { RAND }

    (* Types *)
    | "int"                { INT }
    | "bool"               { BOOL }
    | "float"              { FLOAT }
    | "void"               { VOID }
    | "String"             { STRING }
    | "Game"               { GAME }
    | "Strategy"           { STRATEGY }
    | "Player"             { PLAYER }

    (* Variables and literals *)
    | '_'                  { WILD }
    | '"'                  { stringlit (Buffer.create 16) lexbuf }
    | num as lit           { INTLIT(int_of_string lit) }
    | floatnum as lit      { FLOATLIT(float_of_string lit) }
    | identifier as id     { ID(id) }

    (* Operators *)
    | '+'                  { PLUS }
    | '-'                  { MINUS }
```

```
    | ’*’                   { TIMES }
    | ’/’                   { DIVIDE }
    | ’=’                   { ASSIGN }
    | "->"                  { ARROW }
    | "=="                  { EQ }
    | "!="                  { NE }
    | ">"                   { GT }
    | "<"                   { LT }
    | ">="                  { GE }
    | "<="                  { LE }
    | "&&"                  { AND }
    | "||"                  { OR }
    | "!"                   { NOT }
    | "^"                   { CAT }
    | "%"                   { PLAY }
    | "#"                   { CROSS }
    | "~"                   { MUTATE }

    (* Punctuation *)
    | ’(’                   { LPAREN }
    | ’)’                   { RPAREN }
    | ’[’                   { LBRACK }
    | ’]’                   { RBRACK }
    | ’{’                   { LBRACE }
    | ’}’                   { RBRACE }
    | ’:’                   { COLON }
    | ’;’                   { SEMI }
    | ’.’                   { DOT }
    | ’,’                   { COMMA }
    | ’|’                   { BAR }

    | eof                   { EOF }
    | _                     { raise IllegalCharError }

and comment =
  parse "*/" { token lexbuf }
    | _      { comment lexbuf }

(* String parsing borrowed from:
https://github.com/realworldocaml/examples/blob/master/code/parsing/lexer.mll *)

and stringlit buf =
  parse
  | ’"’      { STRINGLIT(Buffer.contents buf) }
  | ’\\’ ’/’ { Buffer.add_char buf ’/’; stringlit buf lexbuf }
  | ’\\’ ’\\’ { Buffer.add_char buf ’\\’; stringlit buf lexbuf }
  | ’\\’ ’b’ { Buffer.add_char buf ’\b’; stringlit buf lexbuf }
  | ’\\’ ’f’ { Buffer.add_char buf ’\012’; stringlit buf lexbuf }
```

```
  | '\\' 'n' { Buffer.add_char buf '\n'; stringlit buf lexbuf }
  | '\\' 'r' { Buffer.add_char buf '\r'; stringlit buf lexbuf }
  | '\\' 't' { Buffer.add_char buf '\t'; stringlit buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      stringlit buf lexbuf }
  | _  { raise IllegalCharError }
  | eof { raise StringUntermError }
```

Listing 2: scanner.mll

```
(* File: ast.ml
 * Created: 10/24/2016
 *
 * COMSW4115 Fall 2016 (CVN)
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> *)

type op = Add | Sub | Mul | Div | Eq | Ne | Gt | Ge | Lt | Le | And | Or | Cat
type uop = Neg | Not
type typ = Int | Bool | Float | String | Void | Game | Strategy | Player
  | Array of typ

type move =
    Move of string
  | Wild

type transition = move list * string

type expr =
    Object of typ * expr list
  | Binop of expr * op * expr
  | Unop of uop * expr
  | IntLit of int
  | BoolLit of bool
  | FloatLit of float
  | StringLit of string
  | Id of string
  | Entry of string * expr
  | Att of expr * string
  | Call of string * expr list
  | Range of expr * expr
  | States of (string * expr * transition list) list
  | Payoffs of (move list * expr list) list
  | Rand
  | Noexpr

```

```
type vdecl = typ * string * expr

type sdecl = string * int

type stmt = Block of stmt list
  | Vdecl of vdecl
  | Sdecl of sdecl list
  | Cross of expr * expr * expr
  | Asn of expr * expr
  | Play of expr list * expr
  | Mut of expr * expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | For of string * expr * stmt
  | SideCall of string * expr list
  | Return of expr

type fdecl = {
  mutable typ    : typ;
  mutable name   : string;
  mutable params : (typ * string) list;
  mutable body   : stmt list;
}

type content = Stmt of stmt
  | Fdecl of fdecl

type program = content list
```

Listing 3: ast.ml

```
/* File: parser.mly
 * Created: 10/24/2016
 *
 * COMSW4115 Fall 2016 (CVN)
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> */

%{ open Ast %}

%token LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE COLON SEMI DOT COMMA BAR
%token PLUS MINUS TIMES DIVIDE ASSIGN ARROW EQ NE GT LT GE LE AND OR NOT CAT
%token PLAY CROSS MUTATE IF ELSE WHILE FOR IN RETURN TRUE FALSE STRAT RAND
%token INT BOOL FLOAT VOID STRING GAME STRATEGY PLAYER WILD EOF

%token <int> INTLIT
%token <float> FLOATLIT
```

```
%token <string> STRINGLIT
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE
%left CAT
%left OR
%left AND
%left EQ NE
%left GT GE LT LE
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG

%start program
%type <Ast.program> program

%%

program: contents EOF { List.rev $1 }

contents:
    /* nothing */  { [] }
  | contents fdecl { Fdecl($2) :: $1 }
  | contents stmt { Stmt($2) :: $1 }

fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE stmts RBRACE
  { { typ = $1; name = $2; params = $4; body = List.rev $7 } }

formals_opt:
    /* nothing */ { [] }
  | formals     { List.rev $1 }

formals:
    typ ID              { [($1, $2)] }
  | formals COMMA typ ID { ($3, $4) :: $1 }

typ:  atom_typ           { $1 }
  | atom_typ LBRACK RBRACK { Array($1) }

atom_typ:
    INT     { Int }
  | BOOL    { Bool }
  | FLOAT   { Float }
  | VOID    { Void }
  | STRING  { String }
  | GAME    { Game }
```

```
  | STRATEGY { Strategy }
  | PLAYER  { Player }

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

stmt:
    vdecl SEMI                               { Vdecl($1) }
  | sdecl SEMI                               { Sdecl($1) }
  | identifiers PLAY expr SEMI               { Play(List.rev $1, $3) }
  | identifier MUTATE expr SEMI              { Mut($1, $3) }
  | identifier CROSS expr CROSS identifier SEMI { Cross($1, $3, $5) }
  | identifier ASSIGN expr SEMI              { Asn($1, $3) }
  | ID LPAREN actuals_opt RPAREN SEMI        { SideCall($1, $3) }
  | RETURN expr_opt SEMI                     { Return($2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE  { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt     { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt            { While($3, $5) }
  | FOR ID IN expr stmt                      { For($2, $4, $5) }
  | LBRACE stmts RBRACE                      { Block(List.rev $2) }

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

actuals:
    expr              { [$1] }
  | actuals COMMA expr { $3 :: $1 }

vdecl:
    typ ID ASSIGN expr { ($1, $2, $4) }
  | typ ID           { ($1, $2, Noexpr) }

sdecl:
    STRAT LBRACE strats RBRACE { List.rev $3 }

strats:
    ID            { [($1,0)] }
  | strats COMMA ID /* Set move constants */
    { match $1 with (x, i)::tl -> ($3, i + 1)::(x, i)::tl | _ -> [($3, 0)] }

identifiers:
    identifier                 { [$1] }
  | identifiers COMMA identifier { $3 :: $1 }

identifier:
    ID                              { Id($1) }
```

```
  | ID LBRACK expr RBRACK             { Entry($1, $3) }
  | identifier DOT ID                 { Att($1, $3) }
  | ID LPAREN actuals_opt RPAREN      { Call($1, $3) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr        { $1 }

expr:
    atom_typ LBRACK params RBRACK { Object($1, List.rev $3) }
  | expr PLUS  expr            { Binop($1, Add, $3) }
  | expr MINUS expr            { Binop($1, Sub, $3) }
  | expr TIMES expr            { Binop($1, Mul, $3) }
  | expr DIVIDE expr           { Binop($1, Div, $3) }
  | expr EQ    expr            { Binop($1, Eq, $3) }
  | expr NE    expr            { Binop($1, Ne, $3) }
  | expr GT    expr            { Binop($1, Gt, $3) }
  | expr GE    expr            { Binop($1, Ge, $3) }
  | expr LT    expr            { Binop($1, Lt, $3) }
  | expr LE    expr            { Binop($1, Le, $3) }
  | expr AND   expr            { Binop($1, And, $3) }
  | expr OR    expr            { Binop($1, Or, $3) }
  | expr CAT   expr            { Binop($1, Cat, $3) }
  | MINUS expr %prec NEG       { Unop(Neg, $2) }
  | NOT  expr                  { Unop(Not, $2) }
  | TRUE                       { BoolLit(true) }
  | FALSE                      { BoolLit(false) }
  | INTLIT                     { IntLit($1) }
  | FLOATLIT                   { FloatLit($1) }
  | STRINGLIT                  { StringLit($1) }
  | identifier                 { $1 }
  | LBRACK expr COLON expr RBRACK { Range($2, $4) }
  | RAND                       { Rand }
  | LPAREN expr RPAREN         { $2 }
  | LBRACE states RBRACE       { States(List.rev $2) }
  | LBRACE outcomes RBRACE     { Payoffs(List.rev $2) }

params:
    expr          { [$1] }
  | params BAR expr { $3::$1 }

states:
    state       { [$1] }
  | states state { $2 :: $1 }

state:
     ID COLON expr COMMA transitions SEMI { ($1, $3, List.rev $5) }

```

```
transitions:
   transition                { [$1] }
 | transitions BAR transition { $3 :: $1 }

transition:
   LPAREN moves RPAREN ARROW ID { ($2, $5) }

moves:
   move             { [$1] }
 | moves COMMA move { $3 :: $1 }

move:
   ID   { Move($1) }
 | WILD { Wild }

outcomes:
   outcome            { [$1] }
 | outcomes BAR outcome { $3 :: $1 }

outcome:
   LPAREN moves RPAREN ARROW LPAREN payoffs RPAREN
   { (List.rev $2, List.rev $6) }

payoffs:
    expr              { [$1] }
  | payoffs COMMA expr { $3 :: $1 }
```

Listing 4: parser.mly

```
(* File: sast.ml
 * Created: 11/23/2016
 *
 * COMSW4115 Fall 2016 (CVN)
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> *)

open Ast

(* Helpful type and functions for tracking scope *)
type symbol_table = {
  mutable parent      : symbol_table option;
  mutable constants   : sdecl list;
  mutable variables   : vdecl list;
  mutable functions   : fdecl list;
  mutable state_labels : (string * int) list;
  mutable return_type : typ;
  mutable has_return  : bool
```

```
}

let rec find_constant scope name =
  try
    List.find (fun (s,_) -> s = name) scope.constants
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_constant parent name
    | _ -> raise Not_found

let rec find_function scope name =
  try
    List.find (fun f -> f.name = name) scope.functions
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_function parent name
    | _ -> raise Not_found

let rec find_variable scope name =
  try
    List.find (fun (_,s,_) -> s = name) scope.variables
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_variable parent name
    | _ -> raise Not_found

(* More informative types *)
type t = Int | Bool | Float | Void | String
  | Game of int option * int
  | Strategy of int option * int option * int
  | Player
  | Array of typ * int option

(* Useful details for attributes *)
type att_info = {
  relevant_types : typ list;
  att_name       : string;
  att_t          : t
}

(* Expression details *)
type expr_detail =
    ArrayLit of typ * expr_detail
  | GameLit of expr_detail * (move list * expr_detail list) list
  | StrategyLit of expr_detail * expr_detail *
      (string * expr_detail * transition list) list
  | PlayerLit of expr_detail * expr_detail
  | Binop of expr_detail * op * expr_detail
```

```
  | Unop of uop * expr_detail
  | IntLit of int
  | BoolLit of bool
  | FloatLit of float
  | StringLit of string
  | Id of typ * string * expr_detail
  | Entry of (typ * string * expr_detail) * expr_detail
  | Att of expr_detail * att_info
  | Call of fdecl * expr_detail list
  | Range of expr_detail * expr_detail
  | Rand
  | Noexpr

(* Expression details and associated type *)
type expr = expr_detail * t

type stmt = Block of symbol_table * stmt list
  | Vdecl of typ * string * expr_detail
  | Sdecl of (string * int) list
  | Cross of expr * expr * expr
  | Asn of expr * expr
  | Play of expr list * expr
  | Mut of expr * expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  | For of string * expr * stmt
  | SideCall of fdecl * expr_detail list
  | Return of expr

(* Updated fdecl *)
type fdecl = {
  typ    : typ;
  name   : string;
  params : (typ * string) list;
  body   : symbol_table * stmt list
}

type program = symbol_table
```

Listing 5: sast.ml

```
(* File: semant.ml
 * Created: 11/14/2016
 *
 * COMSW4115 Fall 2016
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> *)
```

```ocaml

open Printer
open Ast

module S = Sast
module SM = Map.Make(String)

let check contents =

  (* Helpers *)
  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate list =
    let rec helper = function
        n1 :: n2 :: _ when n1 = n2 -> raise (DupError(n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Raise an exception if a variable declaration has a Void type *)
  let check_not_void = function
      (Void, s, _) -> raise (VoidError(s))
    | _ -> ()
  in

  (* Translate a Sast type into an Ast type *)
  let typ_of_t = function
      S.Int -> Int
    | S.Bool -> Bool
    | S.Float -> Float
    | S.Void -> Void
    | S.String -> String
    | S.Game(_) -> Game
    | S.Strategy(_) -> Strategy
    | S.Player -> Player
    | S.Array(typ,_) -> Array(typ)
  in

  (* Translate an Ast type into a Sast type *)
  let t_of_typ = function
      Int -> S.Int
    | Bool -> S.Bool
    | Float -> S.Float
    | Void -> S.Void
    | String -> S.String
    | Game -> S.Game(None, 0)
    | Strategy -> S.Strategy(None, None, 0)
    | Player -> S.Player
```

```
    | Array(typ) -> S.Array(typ, None)
  in

  (* Ensure that the number of players in l is n *)
  let check_nplayers_equal n l =
    if n = List.length l then () else raise (IllegalOutcomeError(n))
  in

  (* Ensure that t is either a float or an int *)
  let check_float_or_int s t =
    if t = S.Int then ()
    else if t = S.Float then ()
      else raise (TypeError(Int, typ_of_t t, s))
  in

  (* Ensure that t1 and t2 have the same type. t1 is the expected type, t2 is
   * the checked type. *)
  let check_same_type s t1 t2 =
    if t1 = t2 then () else raise (TypeError(t1, t2, s))
  in

  (* Return the type of the array *)
  let get_array_type s = function
      Array(typ) -> typ
    | typ -> raise (TypeError(Array(Int), typ, s))
  in

  (* Return an initial expression for type t *)
  let get_init_expr = function
      Int -> IntLit 0
    | Bool -> BoolLit false
    | Float -> FloatLit 0.0
    | String -> StringLit ""
    | Void -> Noexpr
    | Game -> Object(Game, [IntLit 0; Payoffs([])])
    | Strategy -> Object(Strategy, [IntLit 0; IntLit 0; States([])])
    | Player -> Object(Player,
        [Object(Strategy, [IntLit 0; IntLit 0; States([])])])
    | Array(typ) -> Object(typ, [IntLit 0])
  in

  (* Ensure that t has a string attribute *)
  let check_has_string s = function
      S.Void -> raise (TypeError(String, Void, s))
    | S.Array(typ, _) -> raise (TypeError(String, Array(typ), s))
    | S.Game(_) -> raise (TypeError(String, Game, s))
    | S.Player -> raise (TypeError(String, Player, s))
    | _ -> ()
```

```
in

(* Built-in functions, represented as a symbol_table *)
let built_in_env = {
    S.parent = None; S.constants = []; S.variables = []; S.functions =
      [{ typ = Void; name = "print"; params = [(String, "x")]; body = [] };
      { typ = Void; name = "println"; params = [(String, "x")]; body = [] }];
      S.state_labels = []; S.return_type = Void; S.has_return = true
    }
in

(* Built-in attributes, represented as a String Map *)
let built_in_attrs = SM.add "len"
  { S.relevant_types = [String]; S.att_name = "len"; S.att_t = S.Int }
  (SM.add "string"
  { S.relevant_types = [Int; Bool; Float; String; Strategy];
    S.att_name = "string"; S.att_t = S.String }
  (SM.add "size"
  { S.relevant_types = [Strategy]; S.att_name = "size"; S.att_t = S.Int }
  (SM.add "moves"
  { S.relevant_types = [Strategy; Game]; S.att_name = "moves";
    S.att_t = S.Int }
  (SM.add "players"
  { S.relevant_types = [Strategy; Game]; S.att_name = "players";
    S.att_t = S.Int }
  (SM.add "state"
  { S.relevant_types = [Player]; S.att_name = "state"; S.att_t = S.Int }
  (SM.add "rounds"
  { S.relevant_types = [Player]; S.att_name = "rounds"; S.att_t = S.Int }
  (SM.add "strategy"
  { S.relevant_types = [Player]; S.att_name = "strategy";
    S.att_t = S.Strategy(None, None, 0)}
  (SM.add "delta"
  { S.relevant_types = [Player]; S.att_name = "size"; S.att_t = S.Float }
  (SM.add "payoff"
  { S.relevant_types = [Player]; S.att_name = "payoff"; S.att_t = S.Float }
  (SM.singleton "reset"
  { S.relevant_types = [Player]; S.att_name = "reset"; S.att_t = S.Player }
  ))))))))))
in

(* Return the string attribute of e' if its type allows it *)
let string_attribute s e' = function
    S.String -> e'
  | t -> check_has_string s t;
      (S.Att(e', SM.find "string" built_in_attrs))
in

```

```
(* Ensure move list in each transition represents the same number of players.
 * Return the number of players. *)
let rec transitions l = (function
    (ml,_)::tl -> check_nplayers_equal (List.length l) ml; transitions ml tl
  | [] -> List.length l)
in

(* Return true if lists are identical, false otherwise. Code borrowed from
 * http://stackoverflow.com/questions/3997895/comparing-list-of-floats *)
let rec compare_types l1 l2 = match l1, l2 with
    [], []      -> true
  | [], _       -> false
  | _, []       -> false
  | t1::tl1, t2::tl2 -> t1 = t2 && compare_types tl1 tl2
in

(* Type check expr, return S.expr *)
let rec expr env = function
    IntLit(i) -> (S.IntLit(i), S.Int)
  | BoolLit(b) -> (S.BoolLit(b), S.Bool)
  | FloatLit(f) -> (S.FloatLit(f), S.Float)
  | StringLit(str) -> (S.StringLit(str), S.String)

    (* Game constructor *)
  | Object(Game, [e; Payoffs(ol)]) ->
      let (e', t) = expr env e
      and ol' = List.map (fun (ml,el) ->
        (ml, List.map fst (List.map (expr env) el))) ol
      and nplayers = (match ol with
          [] -> 0
        | hd::_ -> List.length (fst hd))
      in

      (* Ensure each number of players corresponds to number of outcomes. *)
      List.iter (check_nplayers_equal nplayers) (List.map fst ol');
      List.iter (check_nplayers_equal nplayers) (List.map snd ol');

      (* For each type in the payoff list, check that it's a float or int *)
      let check_payoffs pl =
        List.iter (check_float_or_int "payoff list")
        (List.map snd (List.map (expr env) pl))
      in
      (List.iter check_payoffs (List.map snd ol));

      (* Ensure specified number of moves is an integer.
       * Record this information if possible. *)
      (match (e', t) with
          (S.IntLit(i), S.Int) -> (S.GameLit(e', ol'),
```

```
          S.Game(Some(i), nplayers))
      | (_, S.Int)       -> (S.GameLit(e', ol'),
          S.Game(None, nplayers))
      | _                -> raise
        (TypeError(Int, typ_of_t t, "Game constructor")))

    (* Strategy constructor *)
  | Object(Strategy, [e1; e2; States(sl)]) ->
      (* Check that each state's output is of type int, and that each
       * transition represents the correct number of players. *)
      let rec states n = (function
          (_,e,trans)::tl -> let (_,t) = expr env e in
            (match t with
                S.Int -> ()
              | _ -> raise
                  (TypeError(Int, typ_of_t t,"Strategy constructor"))
            );
            (match trans with
                [] -> 0
              | hd::_ -> let ml = fst hd in
                let nplayers = transitions ml trans in
                if nplayers = n then states nplayers tl
                else raise (IllegalOutcomeError(n))
            )
        | _ -> n)
      in

      (* First argument of states function: # of moves in first transition
       * of first state. *)
      let n =
        (match sl with
            [] -> 0
          (* Get first state *)
          | shd::_ -> let ts = (fun (_, _, ts) -> ts) shd in
              (match ts with
                  [] -> 0
                (* Get number of moves in first transition *)
                | thd::_ -> List.length (fst thd)
              )
        )
      in

      (* Call states function *)
      let nplayers = states n sl
      and sl' = List.map (fun (s,e,trans) -> (s, fst (expr env e), trans)) sl
      and (e1', t1) = expr env e1
      and (e2', t2) = expr env e2
      in
```

```

    let nstates = match (e1', t1) with
        (S.IntLit(i), S.Int) -> Some(i)
      | (_, S.Int) -> None
      | _ -> raise (TypeError(Int, typ_of_t t1, "Strategy constructor"))
    in

    let nmoves = match (e2', t2) with
        (S.IntLit(i), S.Int) -> Some(i)
      | (_, S.Int) -> None
      | _ -> raise (TypeError(Int, typ_of_t t2, "Strategy constructor"))
    in

  (S.StrategyLit(e1', e2', sl'), S.Strategy(nstates, nmoves, nplayers))

  (* Player constructor *)
| Object(Player, e::el) when typ_of_t (snd (expr env e)) = Strategy ->
    let (e',_) = expr env e
    and s = "Player constructor"
    in

    (match el with
      | [e1] ->
          let (e1', t1) = expr env e1 in check_float_or_int s t1;
          (S.PlayerLit(e', e1'), S.Player)

      | [] ->
          (S.PlayerLit(e', S.FloatLit 1.0),
            S.Player)
      | _ -> raise (IllegalObjectError(Player)))

(* Array constructor *)
| Object(typ,[e]) ->
    (match typ with
        (* replay doesn't allow arrays of arrays *)
        Array(_) -> raise (IllegalObjectError(typ))
      | _ ->
        let (e',t) = expr env e in
        (match (e',t) with
            (S.IntLit(i), S.Int) -> (S.ArrayLit(typ, e'),
              S.Array(typ, Some(i)))
          | (_, S.Int) -> (S.ArrayLit(typ, e'), S.Array(typ, None))
          | _ -> raise (TypeError(Int, typ_of_t t, "array constructor"))
        )
    )

(* Unrecognized constructor format *)
| Object(t,_) -> raise (IllegalObjectError(t))
```

```
  | Binop(e1, op, e2) ->
      let (e1',t1) = expr env e1
      and (e2',t2) = expr env e2
      in

      if op = Div || op = Mul || op = Add || op = Sub then
        (check_float_or_int "left arithmetic operand" t1;
        check_float_or_int "right arithmetic operand" t2;
        if t1 = S.Int && t2 = S.Int then (S.Binop(e1', op, e2'), S.Int)
        else (S.Binop(e1', op, e2'), S.Float))

      else if op = Eq || op = Ne then
        (let typ1 = typ_of_t t1 in
          (match typ1 with
              Int -> check_float_or_int "right equality operand" t2;
            | Float -> check_float_or_int "right equality operand" t2;
            | Bool -> check_same_type "right equality operand" typ1
                (typ_of_t t2);
        (* Void is used as a placeholder in TypeError to indicate that the
         * second argument is unexpected *)
          |  _ -> raise (TypeError(Void, typ1, "left equality operand")));
        (S.Binop(e1', op, e2'), S.Bool))

      else if op = Gt || op = Ge || op = Le || op = Lt then
        (check_float_or_int "left comparison operand" t1;
        check_float_or_int "right comparison operand" t2;
        (S.Binop(e1', op, e2'), S.Bool))

      else if op = And || op = Or then
        (check_same_type "left boolean operand" Bool (typ_of_t t1);
        check_same_type "right boolean operand" Bool (typ_of_t t2);
        (S.Binop(e1', op, e2'), S.Bool))

      else if op = Cat then
        (let s1 = string_attribute "left concatenation operand" e1' t1 in
        let s2 = string_attribute "right concatenation operand" e2' t2 in
        (S.Binop(s1, op, s2), S.String))
      else (* Not reached *) raise (TypeError(Void,Void,"No!"))

  | Unop(uop, e) ->
      let (e', t) = expr env e in
      if uop = Neg then
        (check_float_or_int "unary negation operand" t;
        (S.Unop(uop, e'), t))
      else if uop = Not then
        (check_same_type "unary not operand" Bool (typ_of_t t);
        (S.Unop(uop, e'), S.Bool))
      else (* Not reached *) raise (TypeError(Void,Void,"No!"))
```

```
  | Id(s) ->
      (try let (typ, s, e) = (S.find_variable env s)
        in
        let (e', t') = expr env e
        in (S.Id(typ, s, e'), t')
      with Not_found ->
        let (e, t) =
          (fun (_,i) -> (IntLit i, S.Int)) (S.find_constant env s)
        in (S.Id(Int, s, fst (expr env e)), t)
      )

  | Entry(s, e) ->
      let (e1', t1) = expr env e
      and (e2', t2) = expr env (Id s)
      in

      check_same_type "array index" Int (typ_of_t t1);
      (match t2 with
          S.Array(typ,_) -> (S.Entry(((typ_of_t t2), s, e2'), e1'),
            (t_of_typ typ))
        | _ -> raise (TypeError(Array(Int), typ_of_t t2,"array entry"))
      )

  | Att(e, s) ->
      let (e', t) = expr env e
      and att = SM.find s built_in_attrs
      in

      (* Find attribute s for type t. *)
      (match t with
          S.Array(_) -> if att.S.att_name = "len" then
              (S.Att(e', { S.relevant_types = [Array(Int)];
                S.att_name = "len"; S.att_t = S.Int }), att.S.att_t)
            else raise (WrongAttrError(s, (typ_of_t t)))

        | _ -> try ignore
            (List.find (fun a -> a = (typ_of_t t)) att.S.relevant_types);
            (S.Att(e', att), att.S.att_t)
          with Not_found -> raise (WrongAttrError(s, (typ_of_t t)))
      )

  | Call(s, el) ->
      let fd = S.find_function env s
      and el' = List.map (expr env) el
      in
      if compare_types
        (List.map (fun (typ,_) -> typ) fd.params)
```

```
        (List.map typ_of_t (List.map snd el'))
      then
        (S.Call(fd, List.map fst el'), t_of_typ fd.typ)

      else raise (ArgError(fd.name))

  | Range(e1, e2) ->
      let (e1', t1) = expr env e1
      and (e2', t2) = expr env e2
      in

      check_same_type "left bound of range" Int (typ_of_t t1);
      check_same_type "right bound of range" Int (typ_of_t t2);
      (S.Range(e1', e2'), S.Array(Int, None))
    (* Create new scope for states, which defines state names *)
  | States(_) -> raise (BadPlacementError("States", "Strategy"))
  | Payoffs(_) -> raise (BadPlacementError("Payoffs", "Game"))
  | Rand -> (S.Rand, S.Float)
  | Noexpr -> (S.Noexpr, S.Void)
in

(* Type check statement, return S.stmt *)
let rec stmt env (*statement =
print_endline (string_of_stmt "" statement);
match statement with*) = function
    Block(sl) -> let env' = { S.parent = Some(env); S.constants = [];
        S.variables = []; S.functions = []; S.state_labels = [];
        S.return_type = env.S.return_type; S.has_return = false } in
      let sl = List.map (fun s -> stmt env' s) sl in S.Block(env', sl)

  (* Initiate variable to a default value *)
  | Vdecl(t,s,Noexpr) ->
      let e = get_init_expr t in
      env.S.variables <- (t,s,e)::env.S.variables;
      let e' = expr env e in
      S.Vdecl(t,s,fst e')

  | Vdecl(t,s,e) ->
      let (e', t') = expr env e in
      check_same_type ("variable declaration of " ^ s) t (typ_of_t t');
      env.S.variables <- (t,s,e)::env.S.variables;
      S.Vdecl(t,s,e')

  | Sdecl(sl) -> List.iter (fun (s,i) ->
      env.S.constants <- (s,i)::env.S.constants) sl;
      S.Sdecl(sl)
  | Cross(e1, e2, e3) ->
      let (e1', t1) = expr env e1
```

```
      and (e2', t2) = expr env e2
      and (e3', t3) = expr env e3
      in

      check_same_type "left cross operand" Player (typ_of_t t1);
      check_same_type "middle cross operand" Float (typ_of_t t2);
      check_same_type "right cross operand" Player (typ_of_t t3);
      S.Cross((e1', t1), (e2', t2), (e3', t3))

  | Asn(e1, e2) ->
      let (e1', t1) = expr env e1 in
      let (e2', t2) = expr env e2 in

      (* Ensure that assignment is to a legal expression *)
      (match e1' with
          S.Id(_)   -> ()
        | S.Entry(_) -> ()
        | _ -> raise (IllegalAssignmentError((typ_of_t) t1))
      );
      check_same_type "variable assignment" (typ_of_t t1) (typ_of_t t2);
      S.Asn((e1', t1), (e2', t2))

  | Play(el, e) ->
      let el' = List.map (expr env) el
      and (e', t) = expr env e
      in

      List.iter (check_same_type "play operand" Player)
        (List.map typ_of_t (List.map snd el'));
      check_same_type "play operand" Game (typ_of_t t);
      S.Play(el', (e', t))

  | Mut(e1, e2) ->
      let (e1', t1) = expr env e1
      and (e2', t2) = expr env e2
      in

      check_same_type "left mutation operand" Player (typ_of_t t1);
      check_same_type "right mutation operand" Float (typ_of_t t2);
      S.Mut((e1', t1), (e2', t2))

  | If(e, s1, s2) ->
      let (e', t) = expr env e
      and s1' = stmt env s1
      and s2' = stmt env s2
      in

      check_same_type "if condition" Bool (typ_of_t t);
```

```
      S.If((e', t), s1', s2')

  | While(e, s) ->
      let (e', t) = expr env e
      and s' = stmt env s
      in

      check_same_type "while condition" Bool (typ_of_t t);
      S.While((e', t), s')

  | For(str, e, s) ->
      let (e', t) = expr env e in

      let typ = get_array_type "for loop array" (typ_of_t t) in

      let env' = { S.parent = Some(env); S.constants = []; S.variables = [];
        S.functions = []; S.state_labels = [];
        S.return_type = env.S.return_type; S.has_return = false }
      in env'.S.variables <- (typ, str, get_init_expr typ)::env'.S.variables;

      S.For(str, (e', t), stmt env' s)

  | SideCall("print", [e]) ->
      let (e',t) = (expr env e)
      and s = "call of print"
      in

      (match t with
          S.String -> S.SideCall(S.find_function env "print", [e'])
        | _ -> check_has_string s t;
          S.SideCall(S.find_function env "print", [(string_attribute s e' t)])
      )

  | SideCall("println", [e]) ->
      let (e',t) = (expr env e)
      and s = "call of println"
      in

      (match t with
          S.String -> S.SideCall(S.find_function env "println", [e'])
        | _ -> check_has_string s t;
          S.SideCall(S.find_function env "println",
            [(string_attribute s e' t)])
      )

  | SideCall(s, el) ->
      let fd = S.find_function env s
      and el' = List.map (expr env) el
```

```
      in

      if compare_types
        (List.map (fun (typ,_) -> typ) fd.params)
        (List.map typ_of_t (List.map snd el'))
      then
        (S.SideCall(fd, List.map fst el'))

      else raise (ArgError(fd.name))

  | Return(e) ->
      let (e', t) = expr env e in
      env.S.has_return <- true;
      check_same_type "return statement" env.S.return_type (typ_of_t t);
      S.Return((e', t))
in

(* Type check function, constant, and variable declarations, convert to
 * S.fdecl or S.vdecl.
 * Return updated list of functions and globals.
 *
 * Note: statements will be checked when the main fdecl is checked, no need
 * to check them twice. *)
let content env (fl, gl) = function

    (* Create a new environment containing the return type *)
    Fdecl(f) -> let env' = { S.parent = Some(env); S.constants = [];
      (* Add f's parameters to scope *)
      S.variables = List.map (fun (typ, s) -> (typ, s, get_init_expr typ))
      f.params; S.functions = []; S.state_labels = [];
      S.return_type = f.typ; S.has_return = false } in
        (* Create a function to add to the list of functions *)
        (* Swap use of "main" as name of function for "", and vice versa *)
        ( {S.typ = f.typ; S.name = (match f.name with "" -> "main"
            | "main" -> "" | s -> s);
          (* Check the function body *)
          S.params = f.params; S.body = match (stmt env') (Block f.body) with
              S.Block(env'', sl) -> (env'', sl)
            | _ -> (* Not reachable *) print_endline "No!"; (env, [])
          }::fl, gl)
  | Stmt(Vdecl(v)) -> let v' = (fun (typ,s,_) -> (typ,s, expr env
      (get_init_expr typ))) v in (fl, v'::gl)
  | Stmt(Sdecl(cl)) -> (fl, (List.map (fun (s,i) -> (Int,s,
      (S.IntLit i,S.Int))) cl) @ gl)
  | _ -> (fl, gl)
in

(* Helpers for build_main *)
```

```
(* Raise an exception if the given scope has a duplicate local *)
let check_scope env = report_duplicate ((List.map fst env.S.constants) @
  (List.map (fun (_,s,_) -> s) env.S.variables));
  report_duplicate (List.map (fun fd -> fd.name) env.S.functions);
  report_duplicate (List.map (fun (s,_) -> s) env.S.state_labels)
in

(* Raise exception if environment already contains a return statement *)
let check_no_return env s =
  if env.S.has_return then raise (IllegalReturnError(s)) else ()
in

(* Raise exception if environment doesn't contain a return statement *)
let check_has_return env s =
  if env.S.has_return then () else raise (MissingReturnError(s))
in


(* Raise exception if move doesn't refer to a previously defined constant.
 * Perform any semantic checks that don't require knowledge of types.
 * Return unit. *)
let check_move env = function
    Move(s) -> (try ignore (S.find_constant env s)
      with Not_found -> raise (MissingSdeclError(s)))
  | Wild -> ()
in

(* Raise exception if transition doesn't refer to a state
 * Perform any semantic checks that don't require knowledge of types.
 * Return unit. *)
let check_transition env' (ml, name) =
  (* A move can't refer to a state name, so perform look-up only in parent *)
  (match env'.S.parent with
      Some(env) ->
        (try List.iter (check_move env) ml
        with Not_found -> raise (MissingSdeclError("")))
    | _ -> ());
  try ignore (List.find (fun (s,_) -> s = name) env'.S.state_labels)
  with Not_found -> raise (MissingStateError(name))
in

(* Raise exception if expr refers to an out of scope variable or function.
 * Perform any semantic checks that don't require knowledge of types.
 * Return unit. *)
let rec check_expr env = function
    Object(_,el) -> List.iter (check_expr env) el
  | Binop(e1,_, e2) -> check_expr env e1; check_expr env e2
  | Unop(_, e) -> check_expr env e
```

```
  | Id(s) -> (try ignore (S.find_variable env s)
      with Not_found -> try ignore(S.find_constant env s)
      with Not_found -> raise (MissingVdeclError(s)))
  | Entry(s, e) -> check_expr env e; (try ignore (S.find_variable env s)
      with Not_found -> raise (MissingVdeclError(s)))
  | Att(e, s) -> check_expr env e;
      (try ignore (SM.find s built_in_attrs)
      with Not_found -> raise (MissingAttrError(s)))
  | Call(s, el) -> (try ignore (S.find_function env s)
      with Not_found -> raise (MissingFdeclError(s)));
      List.iter (check_expr env) el
  | Range(e1, e2) -> check_expr env e1; check_expr env e2
    (* Create new scope for states, which defines state labels *)
  | States(states) -> let env' = { S.parent = Some(env); S.constants = [];
      S.variables = []; S.functions = []; S.state_labels = [];
      S.return_type = env.S.return_type; S.has_return = false } in
      let rec helper i = (function
          (s,e,_)::t -> check_expr env e; env'.S.state_labels <-
            (s, i)::env'.S.state_labels; (helper (i + 1) t)
        | [] -> ())
      in (helper 0 states); check_scope env';
      List.iter (fun (_,_,tl) -> List.iter (check_transition env') tl)
        states
  | Payoffs(ol) -> List.iter (fun (ml,el) -> List.iter (check_move env) ml;
      List.iter (check_expr env) el) ol
  | _ -> ()
in


(* Raise exception if stmt refers to an out of scope variable or function,
 * or declares a duplicate variable or function.
 * Perform any semantic checks that don't require knowledge of type.
 * Return unit. *)
let rec check_stmt env = function
    Block(sl) -> let env' = { S.parent = Some(env); S.constants = [];
        S.variables = []; S.functions = []; S.state_labels = [];
        S.return_type = env.S.return_type; S.has_return = false } in
        (* Ensure no statement except last is a return *)
        List.iter (fun s -> check_no_return env' "block"; check_stmt env' s)
          sl; check_scope env'
  | Vdecl(t,s,e) -> check_expr env e;
      env.S.variables <- (t,s,e)::env.S.variables
  | Sdecl(sl) -> List.iter (fun (s,i) ->
      env.S.constants <- (s,i)::env.S.constants) sl
  | Cross(e1, e2, e3) -> check_expr env e1; check_expr env e2;
      check_expr env e3
  | Asn(e1, e2) -> check_expr env e1; check_expr env e2
  | Play(el, e) -> check_expr env e;
```

```
      List.iter (check_expr env) el;
  | Mut(e1, e2) -> check_expr env e1; check_expr env e2
  | If(e, s1, s2) -> check_expr env e; check_stmt env s1;
      check_stmt env s2
  | While(e, s) -> check_expr env e; check_stmt env s
  | For(str, e, stmt) -> check_expr env e; let env' = { S.parent =
        Some(env); S.constants = []; S.variables = []; S.functions = [];
        S.state_labels = []; S.return_type = env.S.return_type;
        S.has_return = env.S.has_return }
      in env'.S.variables <- (Int, str, Noexpr)::env'.S.variables;
      check_stmt env' stmt
  | SideCall(s, el) -> (try ignore (S.find_function env s)
      with Not_found -> raise (MissingFdeclError(s)));
      List.iter (check_expr env) el
  | Return(e) -> env.S.has_return <- true; check_expr env e
in

(* Raise exception if function refers to an out of scope variable or function.
 * Perform any semantic checks that don't require knowledge of types.
 * Return unit *)
let check_fdecl env f =
  let env' = {
      S.parent = Some(env); S.constants = []; S.variables = [];
      S.functions = []; S.state_labels = []; S.return_type = Void;
      S.has_return = false
    } in
  List.iter (fun (t,s) -> env'.S.variables <- (t, s, Noexpr) ::
    env'.S.variables; check_not_void (t, s, Noexpr)) f.params;
  report_duplicate (List.map snd f.params);
  (* Function declarations reach all function bodies; add all function names
   * to scope. *)
  List.iter (function Fdecl(f) -> env'.S.functions <- f::env'.S.functions
    | _ -> ()) contents;
  (* Ensure last statement is a return *)
  List.iter (fun s -> check_no_return env' f.name; check_stmt env' s) f.body;
  if f.typ = Void then (check_no_return env' f.name;) else
    (check_has_return env' f.name);
  (* Finally, check that the scope contains no duplicates *)
  check_scope env'
in

let main = { typ = Int; name = ""; params = []; body =
(* Build main function body and update the built-in environment while
 * applying the following top-level scoping rules:
 * 1. Function declarations only reach following statements.
 * 2. Function declarations reach all function bodies.
 * 3. Variable and strategy declarations reach all following statements and
 *    function declarations.
```

```
   *
   * Perform any semantic checks that don't require knowledge of types.
   *
   * Return stmt list.
   *
   * Note: It's impossible to declare a function named "", so "" is used as a
   * placeholder for the main function's name. *)
      let rec build_main env = function
          Fdecl(f)::tl -> env.S.functions <- f::env.S.functions;
            check_fdecl env f; build_main env tl
        | Stmt(Vdecl(t,s,Noexpr))::tl ->
            let e = get_init_expr t in
            env.S.variables <- (t,s,e)::env.S.variables;
            Asn(Id s, e) :: build_main env tl
        | Stmt(Vdecl(t,s,e))::tl ->
            env.S.variables <- (t,s,e)::env.S.variables;
            check_expr env e; Asn(Id s,e) :: build_main env tl
        | Stmt(Sdecl(cl))::tl -> env.S.constants <- cl @ env.S.constants;
            (List.map (fun (s,i) -> Asn(Id s, IntLit i)) cl) @ build_main env tl
        | Stmt(Return(_))::_ -> raise (IllegalReturnError("main method"))
        | Stmt(s)::tl -> ignore (check_stmt env s); s::(build_main env tl)
        | [] -> []
      in List.rev ((Return (IntLit 0))::
        (List.rev ((build_main built_in_env) contents)))
    }
  in

(* Return semantically checked functions and globals *)
check_scope built_in_env;
List.fold_left (content built_in_env) ([],[]) ((Fdecl main)::contents)
```

Listing 6: `semant.ml`

```
(* File: codegen.ml
 * Created: 11/18/2016
 *
 * COMSW4115 Fall 2016 (CVN)
 * replay
 * Eric D. Bolton <edb2129@columbia.edu> *)

open Ast

module L = Llvm
module S = Sast

module SM = Map.Make(String)

```

```
type translation_environment = {
  mutable parent : translation_environment option;
  mutable vars   : (string * L.llvalue) list;
  mutable srand_set : bool
}

let translate (functions, globals) =

  let context = L.global_context () in
  let the_module = L.create_module context "Replay"
  (* Define types *)
  and f64_t = L.double_type context
  and i32_t = L.i32_type context
  and i8_t  = L.i8_type  context
  and i1_t  = L.i1_type  context
  and void_t = L.void_type context
  and ptr_t = L.pointer_type in
  (* Special types *)
  (* Transition type: output move, next states *)
  let trans_t = L.struct_type context [| i32_t; i32_t |] in
  (* Game type: number of moves, number of players, payoff matrix *)
  let game_t = L.struct_type context [| i32_t; i32_t; (ptr_t f64_t) |] in
  (* Strategy type: number of moves, number of players, number of states,
   * number of terminal states, number of inaccessible states, states matrix *)
  let strategy_t = L.struct_type context [| i32_t; i32_t; i32_t;
    (ptr_t trans_t) |] in
  (* Player type: strategy, delta, payoff, current state, rounds played *)
  let player_t = L.struct_type context [| (ptr_t strategy_t); f64_t; f64_t;
    i32_t; i32_t |] in
  (* Function type: stores return type and param types *)
  let func_t = L.var_arg_function_type in
  (* Get array type of ltype *)
  let array_t ltype =
    (* Size; contents *)
    L.struct_type context [| i32_t; (ptr_t ltype) |]
  in

  (* Convert Ast types to LLVM types *)
  let rec ltype_of_typ = function
      Int -> i32_t
    | Bool -> i1_t
    | Float -> f64_t
    | String -> ptr_t i8_t
    | Void -> void_t
    | Game -> ptr_t game_t
    | Strategy -> ptr_t strategy_t
    | Player -> ptr_t player_t
    | Array(typ) -> ptr_t (array_t (ltype_of_typ typ))
```

```
in

(* Convert Ast types to LLVM constant *)
let lconst_of_typ = function
    Int -> L.const_int i32_t 0
  | Bool -> L.const_int i1_t 0
  | Float -> L.const_float f64_t 0.0
  | String -> L.const_null (ptr_t i8_t)
  | Void -> L.const_int (void_t) 0
  | Game -> L.const_null (ptr_t game_t)
  | Strategy -> L.const_null (ptr_t strategy_t)
  | Player-> L.const_null (ptr_t player_t)
  | Array(typ) -> L.const_null (ptr_t (array_t (ltype_of_typ typ)))
in

(* Translate an Ast type into a Sast type *)
let t_of_typ = function
    Int -> S.Int
  | Bool -> S.Bool
  | Float -> S.Float
  | Void -> S.Void
  | String -> S.String
  | Game -> S.Game(None, 0)
  | Strategy -> S.Strategy(None, None, 0)
  | Player -> S.Player
  | Array(typ) -> S.Array(typ, None)
in

(* Declare printf(), which the print, println built-in functions will call *)
let printf_t = func_t i32_t [| ptr_t i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in

(* Declare strcat(), strcpy(), and strlen() which will be used for strings *)
let strcat_t = func_t (ptr_t i8_t) [| ptr_t i8_t; ptr_t i8_t |] in
let strcat_func = L.declare_function "strcat" strcat_t the_module in
let strcpy_t = func_t (ptr_t i8_t) [| ptr_t i8_t; ptr_t i8_t |] in
let strcpy_func = L.declare_function "strcpy" strcpy_t the_module in
let strlen_t = func_t i32_t [| ptr_t i8_t |] in
let strlen_func = L.declare_function "strlen" strlen_t the_module in

(* Declare sprintf(), which will be used to transform types into strings *)
let sprintf_t = func_t (i32_t) [| ptr_t i8_t; ptr_t i8_t |] in
let sprintf_func = L.declare_function "sprintf" sprintf_t the_module in

(* Declare useful math functions *)
let rand_t = func_t (i32_t) [| |] in
let rand_func = L.declare_function "rand" rand_t the_module in
let srand_t = func_t (void_t) [| i32_t |] in
```

```
let srand_func = L.declare_function "srand" srand_t the_module in
let time_t = func_t (i32_t) (Array.make 0 (i32_t)) in
let time_func = L.declare_function "time" time_t the_module in
let pow_t = func_t (f64_t) [| f64_t; f64_t |] in
let pow_func = L.declare_function "pow" pow_t the_module in

(* Define globals and store them in a map *)
let global_vars =
    (* Add global variable to map, with name as key *)
  let global_var map (typ, name, _) =
    let init = lconst_of_typ typ in
    SM.add name (L.define_global name init the_module) map
  in List.fold_left global_var SM.empty globals
in

(* Define functions and store them in a map *)
let function_decls =
  (* Add function declaration fdecl to map, with its name as key *)
  let function_decl map fdecl =
    let name = fdecl.S.name
    (* Types of the function parameters *)
    and param_types =
      Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.S.params) in
    (* Define the function type *)
    let ftype = func_t (ltype_of_typ fdecl.S.typ) param_types in
    (* Add the function to the string map *)
    SM.add name (L.define_function name ftype the_module, fdecl) map in
  (* Add all function types to the string map *)
  List.fold_left function_decl SM.empty functions
in

(* Build the body of fdecl *)
let build_function_body fdecl =

  let (the_function, _) = SM.find fdecl.S.name function_decls in

  (* Garbage collector *)
  L.set_gc (Some("statepoint-example")) the_function;

  (* Instruction builder *)
  let builder = L.builder_at_end context (L.entry_block the_function) in

  (* Generate random number seed *)
  (if fdecl.S.name = "main" then (
    let time = L.build_call time_func (Array.make 0 (L.const_int i32_t 0))
      "time" builder in ignore (L.build_call srand_func [| time |] ""
    builder);) else ());

```

```
(* Print formatting *)
let print_fmt = L.build_global_stringptr "%s" "fmt" builder in
let println_fmt = L.build_global_stringptr "%s\n" "fmtln" builder in
let int_fmt = L.build_global_stringptr "%d" "fmtd" builder in
let float_fmt = L.build_global_stringptr "%g" "fmtf" builder in
let move_fmt = L.build_global_stringptr "%d " "fmtm" builder in
let trans_fmt = L.build_global_stringptr "( %s) -> state %d\n" "fmtt"
  builder in
let state_fmt = L.build_global_stringptr "State %d: play %d\n" "fmts"
  builder in
let clear_fmt = L.build_global_stringptr "" "fmtclr" builder in
let error_fmt = L.build_global_stringptr "%s: %d =/= %d\n" "fmte" builder in

(* Helper functions *)
(* Check whether builder's current block has terminator, then add branch
 * if it does not. *)
let add_terminal builder branch =
  match L.block_terminator (L.insertion_block builder) with
    Some _ -> ()
  | None -> ignore (branch builder)
in

let build_print_error builder str comp i1 i2 =
  let error_string = L.build_global_stringptr str "error" builder in
  (* Create instructions to evaluate condition at end of builder *)
  let bool_val = (L.build_icmp comp) i1 i2 "error" builder in
  (* Create merge block *)
  let merge_bb = L.append_block context "merge" the_function in
  (* Create error block *)
  let error_bb = L.append_block context "error" the_function in
  let error_builder = (L.builder_at_end context error_bb) in
  ignore (L.build_call printf_func [| error_fmt; error_string; i1; i2|]
      "printerror" error_builder);
  add_terminal error_builder (L.build_br merge_bb);
  (* Create branch instruction at end of builder *)
  ignore (L.build_cond_br bool_val error_bb merge_bb builder);
  (* Move builder to end of merge block *)
  L.position_at_end merge_bb builder;
in

(* Get length of array *)
let build_get_arrlen builder a =
  L.build_gep a [| L.const_int i32_t 0; L.const_int i32_t 0 |]
    "getlen" builder
in

(* Get pointer to contents of array *)
let build_get_arrcon builder a =
```

```
      L.build_gep a [| L.const_int i32_t 0; L.const_int i32_t 1 |]
        "getcon" builder
    in

    (* Looks up element at address i of array in structure a *)
    let build_array_access builder a i =
      let addr = L.build_load a "addr" builder in

      let ptrtocon = build_get_arrcon builder addr in

      let conaddr = L.build_load ptrtocon "conaddr" builder in

      L.build_gep conaddr [| i |] "access" builder
    in

    (* Get field i of struct a *)
    let build_get_field builder a i =
      L.build_gep a [| L.const_int i32_t 0; i |] "field" builder
    in

    (* Different types get stored differently. dest must be a pointer to the
     * same type as src. *)
    let build_store_typ t src dest builder =
      (match t with
          S.String ->
            let l = L.build_call strlen_func [| src |] "strlen" builder in
            let result = L.build_array_alloca i8_t l "result" builder in
            ignore (L.build_call strcpy_func [| result ; src |]
              "strcpy" builder);
            L.build_store result dest builder
        | _ -> L.build_store src dest builder
      )
    in

    (* Add function parameter to local environment *)
    let add_param lst (typ, name) param = L.set_value_name name param;
      let local = L.build_alloca (ltype_of_typ typ) name builder
      in
      ignore (build_store_typ (t_of_typ typ) param local builder);
      (name, local)::lst
    in
    let local_vars = List.fold_left2 add_param []
      fdecl.S.params (Array.to_list (L.params the_function))
    in

    (* Create the local environment *)
    let local_env = { parent = None; vars = local_vars; srand_set = false } in
    (* Useful functions for environment *)
```

```
  let child_env env = { parent = Some(env); vars = local_vars;
    srand_set = false } in
  (* Add variable to environment *)
  let add_local builder lst (typ, name) =
    let local = L.build_alloca (ltype_of_typ typ) name builder
    in
    (name, local)::lst
  in

  let add_to_env env builder (typ, name) =
    env.vars <- add_local builder env.vars (typ, name)
  in

  let rec lookup env name =
    try
      snd (List.find (fun (k,_) -> k = name) env.vars)
    with Not_found ->
      (match env.parent with
        Some(parent) -> lookup parent name
      | _ -> SM.find name global_vars)
  in

  let build_float_of builder e =
    let ltype = L.type_of e in

    (* already a float *)
    if ltype = f64_t then e else

    (* change to float *)
    if ltype = i32_t || ltype = i8_t
    then (L.build_sitofp e f64_t "sitofp" builder)

    (* never reached *)
    else e
  in

  (* Get a random number *)
  let rand_number builder =
    let randint = L.build_call rand_func [| |] "rand" builder in
    (* 32767 is the minimum guaranteed number generated by rand *)
    let randint = L.build_urem randint (L.const_int i32_t 32767) "randint"
      builder in
    let e = build_float_of builder randint in
    L.build_fdiv e (L.const_float f64_t 32767.0) "randfloat" builder
  in

  let build_trans_size builder nmoves nplayers =

```

```
    let fmoves = build_float_of builder nmoves in
    let fplayers = build_float_of builder nplayers in
    let ftrans = L.build_call pow_func [| fmoves; fplayers |] "exp"
      builder in
    L.build_fptoui ftrans i32_t "size" builder
  in

  let build_trans_access builder strategy trans_size state trans =
    let transaddr = L.build_load (build_get_field builder strategy
      (L.const_int i32_t 3)) "transaddr" builder in
    let state_ind = L.build_mul trans_size state "state" builder in
    let trans_ind = L.build_add trans state_ind "state" builder in
    L.build_gep transaddr [| trans_ind |] "access" builder
  in

  let build_sprintf builder args =
    L.build_call sprintf_func args "sprintf" builder
  in

  let build_string_alloca builder size =
    L.build_array_alloca i8_t size "stralloca" builder
  in

  let build_strcat builder dest source =
    L.build_call strcat_func [| dest ; source |] "strcat" builder
  in

  (* Compute a ^ b *)
  let build_pow builder a b =
    let fa = build_float_of builder a in
    let fb = build_float_of builder b in
    L.build_call pow_func [| fa; fb |] "pow" builder
  in

  (* Compute (transi % (nmoves ^ movesi)) / (nmoves ^ (movesi - 1)).
   * This retrieves the move value from the current transition. *)
  let build_get_move builder movei nmoves transi =
    let decmovei = L.build_sub movei (L.const_int i32_t 1) "decmovei"
      builder in
    let nmoves_pow_movei = build_pow builder nmoves movei in
    let nmoves_pow_decmovei = build_pow builder nmoves decmovei in
    let pow1 = L.build_fptoui nmoves_pow_movei i32_t "fptoui" builder in
    let pow2 = L.build_fptoui nmoves_pow_decmovei i32_t "fptoui" builder in
    let transi_rem_powdec = L.build_urem transi pow1 "urem" builder in
    L.build_sdiv transi_rem_powdec pow2 "div" builder
  in

  let build_strategy_string builder e =
```

```

    let nplayers = L.build_load (build_get_field builder e
      (L.const_int i32_t 0)) "nplayers" builder in
    let nmoves = L.build_load (build_get_field builder e
      (L.const_int i32_t 1)) "nmoves" builder in
    let nstates = L.build_load (build_get_field builder e
      (L.const_int i32_t 2)) "nstates" builder in

    (* Based on formats of trans_fmt, move_fmt, and state_fmt, string size
     * should be at least ((nplayers x 2 + 15) x ntrans + 16) x nstates *)
    let move_size = L.build_mul nplayers (L.const_int i32_t 2) "move_size"
      builder in
    let moves_size = L.build_add move_size (L.const_int i32_t 15) "moves_size"
      builder in
    let ntrans = build_trans_size builder nmoves nplayers in

    let temp1 = L.build_mul ntrans moves_size "temp1" builder in
    let state_size = L.build_add temp1 (L.const_int i32_t 16) "state_size"
      builder in
    let strat_str_size = L.build_mul state_size nstates "strat_str_size"
      builder in
    let strofstrat = build_string_alloca builder strat_str_size in
    ignore (build_sprintf builder [| strofstrat; clear_fmt |]);

    (* Initialize values for while loops *)
    let init_env = child_env local_env in
    ignore (add_to_env init_env builder (Int, "statei"));
    ignore (add_to_env init_env builder (Int, "transi"));
    ignore (add_to_env init_env builder (Int, "movei"));
    ignore (L.build_store (L.const_int i32_t 0)
      (lookup init_env "statei") builder);

    (******************************** While 1 ********************************)
    (* Basic block for while condition *)
    let cond1_bb = L.append_block context "whileone" the_function in
      ignore (L.build_br cond1_bb builder);

    (* Basic block for while loop *)
    let loop1_bb = L.append_block context "whileone_loop" the_function in
    let loop1_builder = (L.builder_at_end context loop1_bb) in

    (* Set transi to 0 *)
    ignore (L.build_store (L.const_int i32_t 0)
      (lookup init_env "transi") loop1_builder);
    (* Get location of current transition *)
    let statei = L.build_load (lookup init_env "statei") "load"
      loop1_builder in
    let transi = L.build_load (lookup init_env "transi") "load"
```

```
      loop1_builder in

    let current_trans = build_trans_access loop1_builder e ntrans statei
      transi in

    (* Get current output move, initialize state string by clearing it, print
     * current state and its output move *)
    let current_output = L.build_load (build_get_field loop1_builder
      current_trans (L.const_int i32_t 0)) "current_output" loop1_builder in
    let strofstate = build_string_alloca loop1_builder
      (L.const_int i32_t 16) in
    ignore (build_sprintf loop1_builder [| strofstate; clear_fmt |]);
    ignore (build_sprintf loop1_builder [| strofstate; state_fmt; statei;
        current_output|]);
    ignore (build_strcat loop1_builder strofstrat strofstate);
    (**************************** Nested while 2 ****************************)
    (* Basic block for while condition *)
    let cond2_bb = L.append_block context "whiletwo" the_function in
      ignore (L.build_br cond2_bb loop1_builder);

    (* Basic block for while loop *)
    let loop2_bb = L.append_block context "whiletwo_loop" the_function in
    let loop2_builder = (L.builder_at_end context loop2_bb) in

    (* Initialize move and transition strings *)
    let moves_str_size = L.build_mul nmoves (L.const_int i32_t 2) "mul"
      loop2_builder in
    let trans_str_size = L.build_add moves_str_size (L.const_int i32_t 15)
      "add" loop2_builder in
    let strofmoves = build_string_alloca loop2_builder moves_str_size in
    ignore (build_sprintf loop2_builder [| strofmoves; clear_fmt |]);
    let stroftrans = build_string_alloca loop2_builder trans_str_size in
    ignore (build_sprintf loop2_builder [| stroftrans; clear_fmt |]);

    ignore (L.build_store nplayers (lookup init_env "movei") loop2_builder);
    (**************************** Nested while 3 ****************************)
    (* Basic block for while condition *)
    let cond3_bb = L.append_block context "whilethree" the_function in
      ignore (L.build_br cond3_bb loop2_builder);

    (* Basic block for while loop *)
    let loop3_bb = L.append_block context "whilethree_loop" the_function in
    let loop3_builder = (L.builder_at_end context loop3_bb) in

    (* Initialize move and transition strings *)
    let movei = L.build_load (lookup init_env "movei") "load" loop3_builder in
    let transi = L.build_load (lookup init_env "transi") "load"
      loop3_builder in
```

```
      let current_move = build_get_move loop3_builder movei nmoves transi in
      let strofmove = build_string_alloca loop3_builder (L.const_int i32_t 2) in
      ignore (build_sprintf loop3_builder [| strofmove; move_fmt;
        current_move |]);
      ignore (build_strcat loop3_builder strofmoves strofmove);
      let movei_dec = L.build_sub movei (L.const_int i32_t 1) "sub"
        loop3_builder in
      ignore (L.build_store movei_dec (lookup init_env "movei") loop3_builder);
      add_terminal loop3_builder (L.build_br cond3_bb);
      (* Builder at end of the condition block *)
      let cond3_builder = L.builder_at_end context cond3_bb in
      (* Add instruction at end of condition block
       * to compute the boolean value *)
      let bool_val = L.build_icmp L.Icmp.Sgt (L.build_load (lookup init_env
        "movei") "load" cond3_builder) (L.const_int i32_t 0) "sgt" cond3_builder
      in
      let merge3_bb = L.append_block context "merge" the_function in
      (* Add branch at end of condition block based on bool_val *)
      ignore (L.build_cond_br bool_val loop3_bb merge3_bb cond3_builder);

      let loop2_builder = L.builder_at_end context merge3_bb in
      (************************ End of nested while 3 ************************)
      let statei = L.build_load (lookup init_env "statei") "load"
        loop2_builder in
      let transi = L.build_load (lookup init_env "transi") "load"
        loop2_builder in
      let current_trans = build_trans_access loop2_builder e ntrans statei
        transi in

      let current_nextstate = L.build_load (build_get_field loop2_builder
        current_trans (L.const_int i32_t 1)) "current_nextstate"
        loop2_builder in
      ignore (build_sprintf loop2_builder [| stroftrans; trans_fmt; strofmoves;
        current_nextstate |]);
      ignore (build_strcat loop2_builder strofstrat stroftrans);
      let transi = L.build_load (lookup init_env "transi") "load"
        loop2_builder in
      let transi_inc = L.build_add transi (L.const_int i32_t 1) "add"
        loop2_builder in
      ignore (L.build_store transi_inc (lookup init_env "transi")
        loop2_builder);

      add_terminal loop2_builder (L.build_br cond2_bb);
      (* Builder at end of the condition block *)
      let cond2_builder = L.builder_at_end context cond2_bb in
      (* Add instruction at end of condition block
       * to compute the boolean value *)

```

```
      let bool_val = L.build_icmp L.Icmp.Slt (L.build_load (lookup init_env
        "transi") "load" cond2_builder) ntrans "slt" cond2_builder in
      let merge2_bb = L.append_block context "merge" the_function in
      (* Add branch at end of condition block based on bool_val *)
      ignore (L.build_cond_br bool_val loop2_bb merge2_bb cond2_builder);

      let loop1_builder = L.builder_at_end context merge2_bb in
      (************************ End of nested while 2 ***********************)
      let statei = L.build_load (lookup init_env "statei") "load"
        loop1_builder in
      let statei_inc = L.build_add statei (L.const_int i32_t 1) "add"
        loop1_builder in
      ignore (L.build_store statei_inc (lookup init_env "statei")
        loop1_builder);
      add_terminal loop1_builder (L.build_br cond1_bb);
      (* Builder at end of the condition block *)
      let cond1_builder = L.builder_at_end context cond1_bb in
      (* Add instruction at end of condition block
       * to compute the boolean value *)
      let bool_val = L.build_icmp L.Icmp.Slt (L.build_load (lookup init_env
        "statei") "load" cond1_builder) nstates "slt" cond1_builder in
      let merge1_bb = L.append_block context "merge" the_function in
      (* Add branch at end of condition block based on bool_val *)
      ignore (L.build_cond_br bool_val loop1_bb merge1_bb cond1_builder);

      (* Reposition builder *)
      ignore (L.position_at_end (merge1_bb) builder);
      (*************************** End of while 1 ***************************)
      strofstrat
    in

    let build_string_of builder e =
      let ltype = L.type_of e in

      (* already a string *)
      if ltype = (ptr_t i8_t) then e else

      (* String representation of numbers never exceeds 32 characters *)
      let strofnum = L.build_array_alloca i8_t (L.const_int i32_t 32)
        "strofnum" builder in

      (* Change int to string, copy it into strofnum, return strofnum *)
      if ltype = i32_t || ltype = i1_t
      then (ignore (L.build_call sprintf_func
        [| strofnum ; int_fmt ; e |] "sprintf" builder); strofnum)
      else

      if ltype = i8_t
```

```
      then (ignore (L.build_call sprintf_func
        [| strofnum ; int_fmt ; e |] "sprintf" builder); strofnum)
      else

      (* Change float to string, copy it into strofnum, return strofnum *)
      if ltype = f64_t
      then (ignore (L.build_call sprintf_func
        [| strofnum ; float_fmt ; e |] "sprintf" builder); strofnum)

      (* Build a while loop that appends strategy information *)
      else if ltype = (ptr_t strategy_t) then
        build_strategy_string builder e
      (* never reached *)
      else e
    in

    let build_float_op = function
        Add -> L.build_fadd
      | Sub -> L.build_fsub
      | Mul -> L.build_fmul
      | Div -> L.build_fdiv
      | Eq  -> L.build_fcmp L.Fcmp.Oeq
      | Ne  -> L.build_fcmp L.Fcmp.One
      | Lt  -> L.build_fcmp L.Fcmp.Olt
      | Le  -> L.build_fcmp L.Fcmp.Ole
      | Gt  -> L.build_fcmp L.Fcmp.Ogt
      | Ge  -> L.build_fcmp L.Fcmp.Oge
      | _      -> (* Not reached *) L.build_fadd
    in

    let build_int_op = function
        Add -> L.build_add
      | Sub -> L.build_sub
      | Mul -> L.build_mul
      | Div -> L.build_sdiv
      | And -> L.build_and
      | Or  -> L.build_or
      | Eq  -> L.build_icmp L.Icmp.Eq
      | Ne  -> L.build_icmp L.Icmp.Ne
      | Lt  -> L.build_icmp L.Icmp.Slt
      | Le  -> L.build_icmp L.Icmp.Sle
      | Gt  -> L.build_icmp L.Icmp.Sgt
      | Ge  -> L.build_icmp L.Icmp.Sge
      | _      -> (* Not reached *) L.build_fadd
    in

    (* Build an array of type typ and size size and return a pointer to it *)
    let build_array builder typ size =
```

```
      (* Allocate room for array struct *)
      let newarray = L.build_alloca (array_t typ)
        "newarray" builder in
      (* Allocate room for contents *)
      let contents = L.build_array_alloca typ size "contents" builder in
      (* Get pointer to array size field *)
      let sizedest = L.build_gep newarray [| L.const_int i32_t 0;
        L.const_int i32_t 0 |] "ptrdest" builder in
      (* Get pointer to contents field *)
      let ptrdest = L.build_gep newarray [| L.const_int i32_t 0;
        L.const_int i32_t 1 |] "ptrdest" builder in

      ignore (L.build_store size sizedest builder);
      ignore (L.build_store contents ptrdest builder);

      newarray
    in

    (* Build a game of nmoves and nplayers and return a pointer to it *)
    let build_game builder nmoves nplayers =
      let newgame = L.build_alloca game_t "newgame" builder in
      let fmoves = build_float_of builder nmoves in
      let fplayers = build_float_of builder nplayers in
      let foutcomes = L.build_call pow_func [| fmoves; fplayers |] "exp"
        builder in
      let fsize = L.build_fmul fplayers foutcomes "fsize" builder in
      let size = L.build_fptoui fsize i32_t "size" builder in
      let outcomes = L.build_array_alloca f64_t size "outcomes" builder in

      (* Get pointers to fields *)
      let nplayersdest = L.build_gep newgame [| L.const_int i32_t 0;
        L.const_int i32_t 0 |] "nplayersdest" builder in
      let nmovesdest = L.build_gep newgame [| L.const_int i32_t 0;
        L.const_int i32_t 1 |] "nmovesdest" builder in
      let ptrdest = L.build_gep newgame [| L.const_int i32_t 0;
        L.const_int i32_t 2 |] "ptrdest" builder in

      ignore (L.build_store nplayers nplayersdest builder);
      ignore (L.build_store nmoves nmovesdest builder);
      ignore (L.build_store outcomes ptrdest builder);
      newgame
    in

    (* Calculate an index within an array based on the moves played.
     * Perform calculation:
     * Sum from i = 0 to nplayers-1 of (move_i x nmoves^i) *)
    let rec build_index builder moves nmoves nplayers intplayers i =
      if i = intplayers then (L.const_int i32_t 0) else
```

```
      let fmoves = build_float_of builder nmoves in
      let fplayers = build_float_of builder nplayers in

      let move = L.build_load (build_array_access builder moves
        (L.const_int i32_t i)) "move" builder in
      let fmove = build_float_of builder move in

      let temp = L.build_call pow_func [| fmoves;
        (L.const_uitofp (L.const_int i32_t i) f64_t) |] "temp" builder in
      let temp2 = L.build_fmul temp fmove "temp2" builder in
      let current = L.build_fptoui temp2 i32_t "current" builder in

      L.build_add current (build_index builder moves fmoves fplayers
        intplayers (i + 1)) "result" builder
    in

    let build_get_player_move builder player =
      let state = L.build_load (build_get_field builder player
        (L.const_int i32_t 3)) "state" builder in
      let strategy = L.build_load (build_get_field builder player
        (L.const_int i32_t 0)) "strategy" builder in
      let nplayers = L.build_load (build_get_field builder strategy
        (L.const_int i32_t 0)) "nplayers" builder in
      let nmoves = L.build_load (build_get_field builder strategy
        (L.const_int i32_t 1)) "nmoves" builder in
      let ntrans = build_trans_size builder nmoves nplayers in
      let current_trans = build_trans_access builder strategy ntrans state
        (L.const_int i32_t 0) in
      let output = L.build_load (build_get_field builder current_trans
        (L.const_int i32_t 0)) "output" builder in
        output
    in


    (* Calculate trans access from a list of moves, similar to
      build_moves_payoff_access *)
    let build_moves_trans_access builder strategy intplayers moves statei =
      let nplayers = L.build_load (build_get_field builder strategy
        (L.const_int i32_t 0)) "nplayers" builder in
      let nmoves = L.build_load (build_get_field builder strategy
        (L.const_int i32_t 1)) "nmoves" builder in
      let ntrans = build_trans_size builder nmoves nplayers in
      (* Address of transition array *)
      let transaddr = L.build_load (build_get_field builder strategy
        (L.const_int i32_t 3)) "transaddr" builder in
      let temp_ind1 = L.build_mul ntrans statei "start"
        builder in
      let temp_ind2 = build_index builder moves nmoves nplayers intplayers 0 in
```

```
    let trans_ind = L.build_add temp_ind1 temp_ind2 "index" builder in
    L.build_gep transaddr [| trans_ind |] "access" builder
  in

  (* Return an access to the location of the payoff for playeri *)
  let build_moves_payoff_access builder game playeri moves intplayers =
    let nplayers = L.build_load (build_get_field builder game
      (L.const_int i32_t 0)) "nplayers" builder in
    let nmoves = L.build_load (build_get_field builder game
      (L.const_int i32_t 1)) "nmoves" builder in
    (* Address of payoff matrix *)
    let payaddr = L.build_load (build_get_field builder game
      (L.const_int i32_t 2)) "payaddr" builder in
    let temp3 = build_index builder moves nmoves nplayers intplayers 0 in
    (* Multiply by nplayers; each player gets one payoff *)
    let temp_ind = L.build_mul temp3 nplayers "temp3" builder in
    let player_ind = L.build_add (L.const_int i32_t playeri) temp_ind
      "playerind" builder in
    let access = L.build_gep payaddr [| player_ind |] "access" builder in
    access
  in

  (* Build a strategy of nmoves, nplayers, and nstates.
   * Return a pointer to it. *)
  let build_strategy builder nmoves nplayers nstates =
    let newstrategy = L.build_alloca strategy_t "newstrategy" builder in
    let transsize = build_trans_size builder nmoves nplayers in
    let size = L.build_mul transsize nstates "size" builder in
    let transitions = L.build_array_alloca trans_t size "trans" builder
    in

    (* Get pointers to fields *)
    let nplayersdest = L.build_gep newstrategy [| L.const_int i32_t 0;
      L.const_int i32_t 0 |] "nplayersdest" builder in
    let nmovesdest = L.build_gep newstrategy [| L.const_int i32_t 0;
      L.const_int i32_t 1 |] "nmovesdest" builder in
    let nstatesdest = L.build_gep newstrategy [| L.const_int i32_t 0;
      L.const_int i32_t 2 |] "nstatesdest" builder in
    let ptrdest = L.build_gep newstrategy [| L.const_int i32_t 0;
      L.const_int i32_t 3 |] "ptrdest" builder in

    ignore (L.build_store nplayers nplayersdest builder);
    ignore (L.build_store nmoves nmovesdest builder);
    ignore (L.build_store nstates nstatesdest builder);
    ignore (L.build_store transitions ptrdest builder);
    newstrategy
  in

```

```
  let build_copy_strategy builder source dest fraction error =
    let nplayers = L.build_load (build_get_field builder source
      (L.const_int i32_t 0)) "nplayers" builder in
    let nmoves = L.build_load (build_get_field builder source
      (L.const_int i32_t 1)) "nmoves" builder in
    let nstates = L.build_load (build_get_field builder source
      (L.const_int i32_t 2)) "nstates" builder in
    let ntrans = build_trans_size builder nmoves nplayers in
    let size = L.build_mul ntrans nstates "size" builder in

    let fsize = build_float_of builder size in
    let fmoves = build_float_of builder nmoves in
    let fstates = build_float_of builder nstates in

    let fcopy_size = L.build_fmul fsize fraction "fcopy_size" builder in
    let copy_size = L.build_fptoui fcopy_size i32_t "copy_size" builder in

    let copy_env = child_env local_env in
    ignore (add_to_env copy_env builder (Int, "transi"));
    ignore (L.build_store (L.const_int i32_t 0)
      (lookup copy_env "transi") builder);
    (****************************** Begin while ******************************)
    (* Basic block for while condition *)
    let cond_bb = L.append_block context "copycond" the_function in
      ignore (L.build_br cond_bb builder);
    (* Basic block for while loop: get the source, dest output move and
     * next state. *)
    let loop_bb = L.append_block context "copyloop" the_function in
    let loop_builder = (L.builder_at_end context loop_bb) in
    let transi = L.build_load (lookup copy_env "transi") "load"
      loop_builder in
    let sourceaddr = L.build_load (build_get_field loop_builder source
      (L.const_int i32_t 3)) "sourceaddr" loop_builder in
    let destaddr = L.build_load (build_get_field loop_builder dest
      (L.const_int i32_t 3)) "destaddr" loop_builder in
    let source_trans = L.build_gep sourceaddr [| transi |] "source"
      loop_builder in
    let dest_trans = L.build_gep destaddr [| transi |] "dest" loop_builder in
    let source_output = L.build_load (build_get_field loop_builder
      source_trans (L.const_int i32_t 0)) "source_output" loop_builder in
    let dest_output = build_get_field loop_builder dest_trans
      (L.const_int i32_t 0) in
    let source_nextstate = L.build_load (build_get_field loop_builder
      source_trans (L.const_int i32_t 1)) "source_nextstate" loop_builder in
    let dest_nextstate = build_get_field loop_builder dest_trans
        (L.const_int i32_t 1) in
    let rand1 = rand_number loop_builder in
    (****************************** Begin if1 ******************************)
```

```
(* Create instructions to evaluate condition at end of builder *)
let bool_val1 = (L.build_fcmp L.Fcmp.Olt) rand1 error "randcomp"
  loop_builder in
(* Create merge block *)
let merge1_bb = L.append_block context "merge" the_function in
(* Create then block, ensure it has a terminator *)
let then1_bb = L.append_block context "then" the_function in
let then1_builder = L.builder_at_end context then1_bb in
let rand = rand_number then1_builder in
let frandmove = L.build_fmul rand fmoves "frandmove" then1_builder in
let randmove = L.build_fptoui frandmove i32_t "randmove" then1_builder in
ignore (L.build_store randmove dest_output then1_builder);
add_terminal then1_builder (L.build_br merge1_bb);

(* Create else block, ensure it has a terminator *)
let else1_bb = L.append_block context "else" the_function in
let else1_builder = (L.builder_at_end context else1_bb) in
ignore (L.build_store source_output dest_output else1_builder);
add_terminal else1_builder (L.build_br merge1_bb);
(* Create branch instruction at end of builder *)
ignore (L.build_cond_br bool_val1 then1_bb else1_bb loop_builder);
(* Move builder to end of merge block *)
let loop_builder = L.builder_at_end context merge1_bb in
(******************************** End if *********************************)
let rand2 = rand_number loop_builder in
(*************************** Begin second if *****************************)
(* Create instructions to evaluate condition at end of builder *)
let bool_val2 = (L.build_fcmp L.Fcmp.Olt) rand2 error "randcomp"
  loop_builder in
(* Create merge block *)
let merge2_bb = L.append_block context "merge" the_function in
(* Create then block, ensure it has a terminator *)
let then2_bb = L.append_block context "then" the_function in
let then2_builder = L.builder_at_end context then2_bb in
let rand = rand_number then2_builder in
let frandstate = L.build_fmul rand fstates "frandstate" then2_builder in
let randstate = L.build_fptoui frandstate i32_t "randstate"
  then2_builder in
ignore (L.build_store randstate dest_nextstate then2_builder);
add_terminal then2_builder (L.build_br merge2_bb);
(* Create else block, ensure it has a terminator *)
let else2_bb = L.append_block context "else" the_function in
let else2_builder = (L.builder_at_end context else2_bb) in
ignore (L.build_store source_nextstate dest_nextstate else2_builder);
add_terminal else2_builder (L.build_br merge2_bb);
(* Create branch instruction at end of builder *)
ignore (L.build_cond_br bool_val2 then2_bb else2_bb loop_builder);
(* Move builder to end of merge block *)
```

```
    let loop_builder = L.builder_at_end context merge2_bb in
    (**************************** End second if *****************************)
    let transi_inc = L.build_add transi (L.const_int i32_t 1)
      "inc" loop_builder in
    ignore (L.build_store transi_inc (lookup copy_env "transi")
      loop_builder);
    add_terminal loop_builder (L.build_br cond_bb);
    (* Builder at end of the condition block *)
    let cond_builder = L.builder_at_end context cond_bb in
    (* Add instruction at end of condition block
     * to compute the boolean value *)
    let bool_val = (L.build_icmp L.Icmp.Slt) (L.build_load
      (lookup copy_env "transi") "transi" cond_builder) copy_size
      "statecomp" cond_builder in
    let merge_bb = L.append_block context "merge" the_function in
    (* Add branch at end of condition block based on bool_val *)
    ignore (L.build_cond_br bool_val loop_bb merge_bb cond_builder);
    ignore (L.position_at_end merge_bb builder);
    (******************************* End while *****************************)
    dest
  in

  (* Build a player using strategy with discount factor delta *)
  let build_player builder strategy delta =
    let newplayer = L.build_alloca player_t "newplayer" builder in

    (* Get pointers to fields *)
    let strategydest = L.build_gep newplayer [| L.const_int i32_t 0;
      L.const_int i32_t 0 |] "strategydest" builder in
    let deltadest = L.build_gep newplayer [| L.const_int i32_t 0;
      L.const_int i32_t 1 |] "deltadest" builder in
    let payoffdest = L.build_gep newplayer [| L.const_int i32_t 0;
      L.const_int i32_t 2 |] "payoffdest" builder in
    let statedest = L.build_gep newplayer [| L.const_int i32_t 0;
      L.const_int i32_t 3 |] "statedest" builder in
    let roundsdest = L.build_gep newplayer [| L.const_int i32_t 0;
      L.const_int i32_t 4 |] "roundsdest" builder in

    (* Get details of player strategy *)
    let nplayers = L.build_load (build_get_field builder strategy
      (L.const_int i32_t 0)) "nplayers" builder in
    let nmoves = L.build_load (build_get_field builder strategy
      (L.const_int i32_t 1)) "nmoves" builder in
    let nstates = L.build_load (build_get_field builder strategy
      (L.const_int i32_t 2)) "nstates" builder in
    let newstrategy = build_strategy builder nmoves nplayers nstates in
    (* Copy strategy into newstrategy. The floats represent the fraction to be
     * copied and the error, respectively. *)
```

```
      ignore (build_copy_strategy builder strategy newstrategy (L.const_float
        f64_t 1.0) (L.const_float f64_t 0.0));
      ignore (L.build_store newstrategy strategydest builder);
      ignore (L.build_store delta deltadest builder);
      ignore (L.build_store (L.const_float f64_t 0.0) payoffdest builder);
      (* Players start in state 0 *)
      ignore (L.build_store (L.const_int i32_t 0) statedest builder);
      ignore (L.build_store (L.const_int i32_t 0) roundsdest builder);
      newplayer
    in

    (* Initialize the array "moves" in env. The array "moves" corresponds to a
     * move set by each player. Each wild card is initially set to 0. *)
    let rec prepare_wild_info env builder player = function
        Wild::tl ->
          (* Store 0 in wild card entry, then store next player's move *)
          let wild = build_array_access builder (lookup env "moves")
            (L.const_int i32_t player)
          in
          ignore (L.build_store (L.const_int i32_t 0) wild builder);
          prepare_wild_info env builder (player + 1) tl;
      | Move(str)::tl ->
          let move =
            build_array_access builder (lookup env "moves")
            (L.const_int i32_t player)
          in
          (* Perform look up for move in parent environment *)
          let parent = (match env.parent with Some(p) -> p | _ -> env) in
          let number = L.build_load (lookup parent str) str builder
          in ignore (L.build_store number move builder);
          prepare_wild_info env builder (player + 1) tl;
      | [] -> () (* Each player's move has been recorded *)
    in

    (* Function that records information for lists containing wild card
     * moves using a for loop. It accepts a store_info function, which
     * computes an array access based on the moves played, and stores info
     * in obj. Env must contain an array entitled "moves" *)
    let rec record_wild_info env builder store_info obj info nmoves player
       ival1 ival2 = function
        Wild::tl ->
          (* Access the value of wild. It is set to 0 by default *)
          let wild = build_array_access builder (lookup env "moves")
            (L.const_int i32_t player)
          in
          ignore (L.build_store (L.const_int i32_t 0) wild builder);
          (* Create condition block *)
          let cond_bb = L.append_block context "wildcond" the_function in
```

```
      ignore (L.build_br cond_bb builder);
     (* Create loop block *)
     let loop_bb = L.append_block context "wildloop" the_function in
     (* Body of loop *)
     let loop_builder = L.builder_at_end context loop_bb in
     (* Recursion: add nested loop *)
     let loop_builder = record_wild_info env loop_builder store_info obj
      info nmoves (player + 1) ival1 ival2 tl
     in
     (* Increment wild *)
     let next = L.build_add (L.build_load wild "loadwild"
      loop_builder) (L.const_int i32_t 1) "next" loop_builder
     in
     ignore (L.build_store next wild loop_builder);
     (* Connect back to condition block *)
     add_terminal loop_builder (L.build_br cond_bb);
     let cond_builder = L.builder_at_end context cond_bb in
     (* Compute the boolean value *)
     let bool_val = (L.build_icmp L.Icmp.Slt) (L.build_load
      wild "player" cond_builder) nmoves "wildcomp"
      cond_builder
     in
     let merge_bb = L.append_block context "merge" the_function in
     (* Add branch at end of condition block based on bool_val *)
     ignore (L.build_cond_br bool_val loop_bb merge_bb cond_builder);
     L.builder_at_end context merge_bb
   | Move(_)::tl ->
     (* This player doesn't have a wild card move, simply move on to
      * next player without incrementing this player's move. *)
     record_wild_info env builder store_info obj info nmoves
      (player + 1) ival1 ival2 tl
   | [] ->
     (* The list is now exhausted, we are building at the core of the
      * nested "wild" loops. Store info in object using an index computed
      * according to "moves" *)
     store_info env builder obj nmoves ival1 ival2 0 info
  in

  let rec expr env builder = function
     S.IntLit(i) -> L.const_int i32_t i
   | S.BoolLit(b) -> L.const_int i1_t (if b then 1 else 0)
   | S.FloatLit(f) -> L.const_float f64_t f
   | S.StringLit(str) -> L.build_global_stringptr str "str" builder
   | S.Noexpr -> L.const_int i32_t 0
   | S.ArrayLit(typ, e) -> let e' = expr env builder e in
     build_array builder (ltype_of_typ typ) e'
   | S.GameLit(nmoves,ol) ->
     let
```

```
          nmoves' = expr env builder nmoves
        in
        let intplayers =
          (match ol with ((ml,_)::_) -> List.length ml | [] -> 0)
        in
        let nplayers' = L.const_int i32_t intplayers in
        let
          game = build_game builder nmoves' nplayers'
        in
        (* Create initialization game outcome, used below to set all payoffs
         * to 0. *)
        let rec initoutcome i =
          if i = intplayers then ([],[]) else
            let (wilds, zeros) = initoutcome (i+1) in
            (Wild::wilds, (S.FloatLit 0.0)::zeros)
        in
        let ol = (initoutcome 0)::ol in

        (* Create environment with a move index representing each player. *)
        let game_env = child_env env in
          ignore (add_to_env game_env builder (Array(Int), "moves"));
          ignore (L.build_store (build_array builder i32_t nplayers')
            (lookup game_env "moves") builder);

        (* The store_info function to be used by record_wild_info *)
        let rec store_payoffs env builder game nmoves intplayers playeri
          ival2 = function
            payoff::tl ->
              (* Compute payoffs based on information in parent environment *)
              let parent = (match env.parent with Some(p) -> p | _ -> env) in
              let payoff' = build_float_of builder
                (expr parent builder payoff) in
              (* Build the payoff access to find where to store the payoffs *)
              let
                access = build_moves_payoff_access builder game playeri
                  (lookup env "moves") intplayers
              in
              (* Now that the access has been computed, store the payoff *)
              ignore(L.build_store payoff' access builder);
              store_payoffs env builder game nmoves intplayers
                (playeri + 1) ival2 tl
          | [] -> builder
        in

        (* Get a new builder by folding a list of outcomes into
         * prepare_wild_info, then record_wild_info *)
        let new_builder = List.fold_left (fun b (ml, pl) ->
          (* Reinitialize the "moves" array *)
```

```
          prepare_wild_info game_env b 0 ml;
          (* Get a new builder from record_wild_info *)
          record_wild_info game_env b store_payoffs game pl nmoves'
            0 intplayers 0 ml) builder ol
        in

        (* Reposition builder *)
        ignore (L.position_at_end (L.insertion_block new_builder) builder);
        game

    | S.StrategyLit(nmoves, nstates, sl) ->
        let nmoves' = expr env builder nmoves in
        let nstates' = expr env builder nstates in
        let intplayers =
          (match sl with
              (_,_,trans)::_ ->
                  (match trans with (ml,_)::_ -> List.length ml | [] -> 0)
            | [] -> 0
          )
        in
        let nplayers' = L.const_int i32_t intplayers in
        let strategy = build_strategy builder nmoves' nplayers' nstates' in

        (* Store each strategy state number, then store each strategy state
         * output move. Find the number of states that have a defined output
         * move. *)
        let strategy_env = child_env env in
          ignore (add_to_env strategy_env builder (Array(Int), "outputs"));
          ignore (L.build_store (build_array builder i32_t nstates')
            (lookup strategy_env "outputs") builder);
        let rec store_state_info statei = function
            (str,e,_)::tl ->
              let output = build_array_access builder (lookup strategy_env
                "outputs") (L.const_int i32_t statei)
              in
              ignore (L.build_store (expr strategy_env builder e) output
                builder);
              ignore (add_to_env strategy_env builder (Int, str));
              ignore (L.build_store (L.const_int i32_t statei)
              (lookup strategy_env str) builder);
              store_state_info (statei + 1) tl;
          | [] -> statei
        in
        (* Useful values for initialization *)
        let defstates = store_state_info 0 sl in
        let ntrans = build_trans_size builder nmoves' nplayers' in

        (* Initialize all states: all undefined state transitions must be
```

```
         * to the state itself, and all undefined output moves must be 0. *)
        let init_env = child_env strategy_env in
        (* Add tracking indices *)
        ignore (add_to_env init_env builder (Int, "statei"));
        ignore (add_to_env init_env builder (Int, "transi"));
        ignore (L.build_store (L.const_int i32_t 0)
          (lookup init_env "statei") builder);

        (*************************** While loop 1 ***************************)
        (* Basic block for while condition *)
        let cond1_bb = L.append_block context "statecond" the_function in
          ignore (L.build_br cond1_bb builder);
        (* Basic block for while loop *)
        let loop1_bb = L.append_block context "state_loop" the_function in
        let loop1_builder = (L.builder_at_end context loop1_bb) in
          ignore (L.build_store (L.const_int i32_t 0)
            (lookup init_env "transi") loop1_builder);

        (************************ Nested while loop ************************)
        let cond2_bb = L.append_block context "transcond" the_function in
          ignore (L.build_br cond2_bb loop1_builder);
        let loop2_bb = L.append_block context "trans_loop" the_function in
        let loop2_builder = (L.builder_at_end context loop2_bb) in
        (* Store current state as transition state *)
        let current_statei = L.build_load (lookup init_env "statei") "statei"
          loop2_builder in
        let current_transi = L.build_load (lookup init_env "transi") "transi"
          loop2_builder in
        let access = build_trans_access loop2_builder strategy ntrans
          current_statei current_transi in
        let current_nextstate = build_get_field loop2_builder access
          (L.const_int i32_t 1) in
        ignore (L.build_store current_statei current_nextstate loop2_builder);

        (***************************** If block *****************************)
        (* Create instructions to evaluate condition at end of builder *)
        let bool_val = (L.build_icmp L.Icmp.Slt) current_statei
          (L.const_int i32_t defstates) "ifcomp" loop2_builder in
        (* Create merge block *)
        let merge3_bb = L.append_block context "merge3" the_function in
        (* Create then block, ensure it has a terminator *)
        let then_bb = L.append_block context "then" the_function in
        let then_builder = L.builder_at_end context then_bb in

        (* Store output move set for current state *)
        let current_statei = L.build_load (lookup init_env "statei") "statei"
          then_builder in
        let current_transi = L.build_load (lookup init_env "transi") "transi"
```

```
      then_builder in
    let access = build_trans_access then_builder strategy ntrans
      current_statei current_transi in
    let current_output = build_get_field then_builder access
      (L.const_int i32_t 0) in
    let output = build_array_access then_builder
      (lookup init_env "outputs") current_statei in
    let outputmove = L.build_load output "outputmove" then_builder in
    ignore (L.build_store outputmove current_output then_builder);
    add_terminal then_builder (L.build_br merge3_bb);
    (* Create else block, ensure it has a terminator *)
    (* Store output move 0, store current state as transition state *)
    let else_bb = L.append_block context "else" the_function in
    let else_builder = (L.builder_at_end context else_bb) in
    let current_statei = L.build_load (lookup init_env "statei") "statei"
      else_builder in
    let current_transi = L.build_load (lookup init_env "transi") "transi"
      else_builder in
    let access = build_trans_access else_builder strategy ntrans
      current_statei current_transi in
    let current_output = build_get_field else_builder access
      (L.const_int i32_t 0) in
    ignore (L.build_store (L.const_int i32_t 0) current_output
      else_builder);
    add_terminal else_builder (L.build_br merge3_bb);

    (* Create branch instruction at end of builder *)
    ignore (L.build_cond_br bool_val then_bb else_bb loop2_builder);

    (* Move builder to end of merge block *)
    let loop2_builder = L.builder_at_end context merge3_bb in
    (************************* End of if block **************************)

    let current_transi = L.build_load (lookup init_env "transi") "transi"
      loop2_builder in
    let next_transi = L.build_add current_transi (L.const_int i32_t 1)
      "next" loop2_builder in
    ignore (L.build_store next_transi (lookup init_env "transi")
      loop2_builder);
    add_terminal loop2_builder (L.build_br cond2_bb);

    (* Builder at end of the condition block *)
    let cond2_builder = L.builder_at_end context cond2_bb in
    (* Add instruction at end of condition block
     * to compute the boolean value *)
    let bool_val = (L.build_icmp L.Icmp.Slt) (L.build_load
      (lookup init_env "transi") "transi" cond2_builder) ntrans
      "transcomp" cond2_builder
```

```
        in
        let merge2_bb = L.append_block context "merge2" the_function in
        (* Add branch at end of condition block based on bool_val *)
        ignore (L.build_cond_br bool_val loop2_bb merge2_bb cond2_builder);
        let loop1_builder = L.builder_at_end context merge2_bb in
        (******************** End of nested while loop ********************)

        let current_statei = L.build_load (lookup init_env "statei") "statei"
          loop1_builder in
        let next_statei = L.build_add current_statei (L.const_int i32_t 1)
          "next" loop1_builder in
        ignore (L.build_store next_statei (lookup init_env "statei")
            loop1_builder);
        add_terminal loop1_builder (L.build_br cond1_bb);

        (* Builder at end of the original condition block *)
        let cond1_builder = L.builder_at_end context cond1_bb in
        (* Add instruction at end of condition block
         * to compute the boolean value *)
        let bool_val = (L.build_icmp L.Icmp.Slt) (L.build_load
          (lookup init_env "statei") "statei" cond1_builder) nstates'
          "statecomp" cond1_builder
        in
        let merge1_bb = L.append_block context "merge1" the_function in
        (* Add branch at end of condition block based on bool_val *)
        ignore (L.build_cond_br bool_val loop1_bb merge1_bb cond1_builder);
        ignore (L.position_at_end merge1_bb builder);

        (* Ensure environment contains "moves" array with
         * move representing each player *)
        ignore (add_to_env init_env builder (Array(Int), "moves"));
        ignore (L.build_store (build_array builder i32_t nplayers')
          (lookup init_env "moves") builder);

        (* Define a recorder function for use by record_wild_info *)
        let store_nextstate env builder strategy nmoves intplayers statei
          player nextstate =
          (* init_env will be passed in, look for nextstate in parent env *)
          let parent = (match env.parent with Some(p) -> p | _ -> env) in
          let nextstate' =
            L.build_load (lookup parent nextstate) nextstate builder in
          let access = build_moves_trans_access builder strategy intplayers
              (lookup env "moves") (L.const_int i32_t statei) in
          let current_nextstate = build_get_field builder
            access (L.const_int i32_t 1) in
          ignore(L.build_store nextstate' current_nextstate builder);

          (* These arguments go unused, but they must fit the format of the
```

```
             * storing function. *)
            ignore(player); ignore(nmoves);

            builder

          in

          (* For each state, perform a list fold left on the transition list *)
          let rec get_new_builder builder statei = function
              (_,_,transl)::tl -> let new_builder =
               List.fold_left (fun b (ml,t) ->
               (* Reinitialize the "moves" array *)
               prepare_wild_info init_env b 0 ml;
               (* Get a new builder from record_wild_info*)
               record_wild_info init_env b store_nextstate strategy t nmoves'
                 0 intplayers statei ml) builder transl in
                 get_new_builder new_builder (statei + 1) tl
            | [] -> builder
          in

          let new_builder = get_new_builder builder 0 sl in

          (* Reposition builder *)
          ignore (L.position_at_end (L.insertion_block new_builder) builder);
          strategy

      | S.Range(e1, e2) ->
          let e1' = expr env builder e1 in
          let e2' = expr env builder e2 in
          let diff = L.build_sub e2' e1' "rangediff" builder in
          let size = L.build_add diff (L.const_int i32_t 1) "rangesize"
            builder in
          let range = build_array builder i32_t size in
          let rangeptr = L.build_alloca (ptr_t (array_t i32_t))
            "rangeptr" builder in
          ignore (L.build_store range rangeptr builder);
          let lastentry = build_array_access builder rangeptr diff in

          (* Create new environment and block to store information *)
          let range_env = child_env env in
          let range_bb = L.append_block context "range" the_function in
            ignore (L.build_br range_bb builder);
          let range_builder = L.builder_at_end context range_bb in
          ignore (add_to_env range_env range_builder (Int, "i"));
          ignore (L.build_store (L.const_int i32_t 0)
            (lookup range_env "i") range_builder);

          (* Create condition block *)
```

```
        let cond_bb = L.append_block context "while" the_function in
          ignore (L.build_br cond_bb range_builder);
        (* Create loop block *)
        let loop_bb = L.append_block context "while_loop" the_function in
        let loop_builder = L.builder_at_end context loop_bb in

        (* Body of loop *)
        let (* Load value of i *)
          curr_index = L.build_load (lookup range_env "i") "loadi"
            loop_builder
        in
        let (* Access current entry *)
          curr_entry = build_array_access loop_builder rangeptr curr_index
        in
        let (* Calculate value to be stored *)
          curr_value = L.build_add e1' curr_index "calcval" loop_builder
        in
        (* Store the value *)
        ignore (L.build_store curr_value curr_entry loop_builder);
        (* Increment i *)
        let iplusone = L.build_add (L.build_load (lookup range_env "i")
          "i" loop_builder) (L.const_int i32_t 1) "iplusone" loop_builder in
        ignore (L.build_store iplusone (lookup range_env "i") loop_builder);
        (* Connect back to condition block *)
        add_terminal loop_builder (L.build_br cond_bb);
        (* Builder at end of the condition block *)
        let cond_builder = L.builder_at_end context cond_bb in
        (* Compute the boolean value *)
        let bool_val = (L.build_icmp L.Icmp.Ne) (L.build_load lastentry
          "lastentry" cond_builder) e2' "rangecomp"
          cond_builder in

        let merge_bb = L.append_block context "merge" the_function in
        (* Add branch at end of condition block based on bool_val *)
        ignore (L.build_cond_br bool_val loop_bb merge_bb cond_builder);
        ignore (L.position_at_end merge_bb builder);
        range

    | S.PlayerLit(strategy, delta) ->
        let strategy' = expr env builder strategy in
        let delta' = expr env builder delta in
        let player = build_player builder strategy' delta' in
        player

    | S.Entry((_, name,_), index) ->
        let i' = expr env builder index
        and a' = lookup env name in
        let arrlen = L.build_load (build_get_arrlen builder (L.build_load a'
```

```
          name builder)) "arrlen" builder in
        let arrmax = L.build_sub arrlen (L.const_int i32_t 1) "sub" builder in
        build_print_error builder "Index out of bounds" L.Icmp.Sgt i' arrmax;
        L.build_load (build_array_access builder a' i') name builder

    | S.Id(_,name,_) -> L.build_load (lookup env name) name builder
    | S.Att(e, a) when a.S.att_name = "string" ->

        let e' = expr env builder e in build_string_of builder e'
    | S.Att(e, a) when a.S.att_name = "len" ->
        let e' = expr env builder e in
        let typ = List.hd a.S.relevant_types in
        (match typ with
            Array(_) -> L.build_load (build_get_arrlen builder e') "arrlen"
              builder
          | String  -> L.build_call strlen_func [| e' |] "strlen" builder
          | _ -> (* Never reached *) build_get_arrlen builder e'
        )

    | S.Att(e, a) when a.S.att_name = "players" || a.S.att_name = "strategy" ->
        let e' = expr env builder e in
        L.build_load (build_get_field builder e' (L.const_int i32_t 0))
          "fieldzero" builder

    | S.Att(e, a) when a.S.att_name = "moves" || a.S.att_name = "delta" ->
        let e' = expr env builder e in
        L.build_load (build_get_field builder e' (L.const_int i32_t 1))
          "fieldone" builder

    | S.Att(e, a) when a.S.att_name = "size" || a.S.att_name = "payoff" ->
        let e' = expr env builder e in
        L.build_load (build_get_field builder e' (L.const_int i32_t 2))
          "fieldtwo" builder

    | S.Att(e, a) when a.S.att_name = "state" ->
        let e' = expr env builder e in
        L.build_load (build_get_field builder e' (L.const_int i32_t 3))
          "state" builder

    | S.Att(e, a) when a.S.att_name = "rounds" ->
        let e' = expr env builder e in
        L.build_load (build_get_field builder e' (L.const_int i32_t 4))
          "rounds" builder

      (* Reset player *)
    | S.Att(e, a) when a.S.att_name = "reset" ->
        let e' = expr env builder e in
        let payoff = build_get_field builder e'(L.const_int i32_t 2)
```

```
        and state = build_get_field builder e' (L.const_int i32_t 3)
        and rounds = build_get_field builder e' (L.const_int i32_t 4)
        in
        ignore (L.build_store (L.const_float f64_t 0.0) payoff builder);
        ignore (L.build_store (L.const_int i32_t 0) state builder);
        ignore (L.build_store (L.const_int i32_t 0) rounds builder);
        e'

    | S.Binop (e1, Cat, e2) ->
        let e1' = expr env builder e1 in
        let l1 = L.build_call strlen_func [| e1' |] "strlen" builder in
        let e2' = expr env builder e2 in
        let l2 = L.build_call strlen_func [| e2' |] "strlen" builder in
        let size = L.build_add l1 l2 "size" builder in
        let result = L.build_array_alloca i8_t size "result" builder in
        ignore (L.build_call strcpy_func [| result ; e1' |]
          "strcpy" builder);
        ignore (L.build_call strcat_func [| result ; e2' |]
          "strcat" builder);
        result

    | S.Binop (e1, op, e2) ->
        let e1' = expr env builder e1
        and e2' = expr env builder e2 in

        if (L.type_of e1') = f64_t || (L.type_of e2') = f64_t then
          (let e1' = build_float_of builder e1'
          and e2' = build_float_of builder e2' in
          ((build_float_op op) e1' e2' "tmp" builder))

        else ((build_int_op op) e1' e2' "tmp" builder)

    | S.Unop (uop, e) ->
        let e' = expr env builder e in
          (match uop with
              Neg -> L.build_neg
            | Not -> L.build_not) e' "tmp" builder

    | S.Call (f, el) ->
        let (fdef, fdecl) = SM.find f.name function_decls in
        let actuals =
          List.rev (List.map (expr env builder) (List.rev el)) in
        let result =
          (match fdecl.S.typ with
              Void -> ""
            | _ -> f.name ^ "_result"
          ) in
        L.build_call fdef (Array.of_list actuals) result builder
```

```
    | S.Rand -> rand_number builder

    | S.Att(_,_) -> L.const_int i32_t 0
  in

  let rec stmt env builder (*statement =
  print_endline (string_of_sast_stmt statement);
  match statement with*) = function
      (* Avoid terminators in middle of basic blocks *)
      S.Block(_, sl) ->
        let block_bb = L.append_block context "block" the_function in
          ignore (L.build_br block_bb builder);
        let block_builder = (List.fold_left (stmt (child_env env))
          (L.builder_at_end context block_bb) sl)
        in
        let merge_bb = L.append_block context "merge" the_function in
        add_terminal block_builder (L.build_br merge_bb);

        L.builder_at_end context merge_bb

    | S.Vdecl(typ,name,e) -> let e' = expr env builder e in
        let t = t_of_typ typ in
        ignore (add_to_env env builder (typ, name));
        ignore (build_store_typ t e' (lookup env name) builder); builder
    | S.Sdecl(sl) -> List.iter (fun (s,i) ->
        let i' = expr env builder (S.IntLit i) in
        ignore (add_to_env env builder (Int, s));
        ignore (L.build_store i' (lookup env s) builder)) sl; builder
    | S.SideCall(f, [e]) when f.name = "print" -> ignore (L.build_call
        printf_func [| print_fmt ; (expr env builder e) |] "printf" builder);
        builder
    | S.SideCall(f, [e]) when f.name = "println" -> ignore (L.build_call
        printf_func [| println_fmt ; (expr env builder e) |]
        "printf" builder); builder
    | S.SideCall(f, el) ->
        let (fdef, _) = SM.find f.name function_decls in
        let actuals =
          List.rev (List.map (expr env builder) (List.rev el)) in
        ignore (L.build_call fdef (Array.of_list actuals) "" builder); builder
    | S.Asn((e1,t), (e2,_)) ->
        let e2' = expr env builder e2 in
        let e1' =
          (match e1 with
              S.Id(_,name,_) -> lookup env name
            | (S.Entry((_,name,_),index)) ->
                let i' = expr env builder index in
                let a' = lookup env name in build_array_access builder a' i'
```

```
            | _ -> (* Never reached *) expr env builder e1
          )
        in
        ignore (build_store_typ t e2' e1' builder); builder
    | S.If ((cond_expr,_), then_stmt, else_stmt) ->
        (* Create instructions to evaluate condition at end of builder *)
        let bool_val = expr env builder cond_expr in
        (* Create merge block *)
        let merge_bb = L.append_block context "merge" the_function in
        (* Create then block, ensure it has a terminator *)
        let then_bb = L.append_block context "then" the_function in
          add_terminal (stmt env (L.builder_at_end context then_bb) then_stmt)
          (L.build_br merge_bb);

        (* Create else block, ensure it has a terminator *)
        let else_bb = L.append_block context "else" the_function in
          add_terminal (stmt env (L.builder_at_end context else_bb) else_stmt)
          (L.build_br merge_bb);

        (* Create branch instruction at end of builder *)
        ignore (L.build_cond_br bool_val then_bb else_bb builder);

        (* Move builder to end of merge block *)
        L.builder_at_end context merge_bb

    | S.While ((cond,_), loop) ->
        (* Basic block for while condition *)
        let cond_bb = L.append_block context "while" the_function in
          ignore (L.build_br cond_bb builder);

        (* Basic block for while loop *)
        let loop_bb = L.append_block context "while_loop" the_function in
          add_terminal (stmt env (L.builder_at_end context loop_bb) loop)
          (L.build_br cond_bb);

        (* Builder at end of the condition block *)
        let cond_builder = L.builder_at_end context cond_bb in
        (* Add instruction at end of condition block
         * to compute the boolean value *)
        let bool_val = expr env cond_builder cond in
        let merge_bb = L.append_block context "merge" the_function in
        (* Add branch at end of condition block based on bool_val *)
        ignore (L.build_cond_br bool_val loop_bb merge_bb cond_builder);

        L.builder_at_end context merge_bb

    | S.For(str, (e, t), s) ->
        let e' = expr env builder e in
```

```
        let typ = (match t with S.Array(t',_) -> t' | _ -> Int) in

        let for_env = child_env env in
        let info_env = child_env for_env in
        let eptr = L.build_alloca (ptr_t (array_t (ltype_of_typ typ)))
          "eptr" builder in
        ignore (L.build_store e' eptr builder);
        let size = L.build_load (build_get_arrlen builder e')
          "size" builder in
        let curr_entry = L.build_load (build_array_access builder
          eptr (L.const_int i32_t 0)) "current" builder
        in

        ignore (add_to_env for_env builder (typ, str));
        ignore (build_store_typ t curr_entry (lookup for_env str) builder);
        ignore (add_to_env info_env builder (Int, "i"));
        ignore (build_store_typ t (L.const_int i32_t 0) (lookup info_env "i")
        builder);

        (* Create condition block *)
        let cond_bb = L.append_block context "for" the_function in
          ignore (L.build_br cond_bb builder);
        (* Create loop block *)
        let loop_bb = L.append_block context "for_loop" the_function in

        (* Body of loop *)
        let loop_builder = stmt for_env (L.builder_at_end context loop_bb) s
        in

        (* Increment i *)
        let iplusone = L.build_add (L.build_load (lookup info_env "i")
          "i" loop_builder) (L.const_int i32_t 1) "iplusone" loop_builder in
        ignore (L.build_store iplusone (lookup info_env "i") loop_builder);
        let (* Load current entry *)
          curr_index = L.build_load (lookup info_env "i") "loadi"
            loop_builder
        in
        let curr_entry = L.build_load (build_array_access loop_builder
          eptr curr_index) "current" loop_builder
        in
        (* Store current entry into str *)
        ignore (build_store_typ t curr_entry (lookup for_env str)
          loop_builder);

        (* Connect back to condition block *)
        add_terminal loop_builder (L.build_br cond_bb);
        (* Builder at end of the condition block *)
        let cond_builder = L.builder_at_end context cond_bb in
```

```
         (* Compute the boolean value *)
         let bool_val = (L.build_icmp L.Icmp.Ne) (L.build_load
           (lookup info_env "i") "i" cond_builder) size "forcomp"
           cond_builder
         in

         let merge_bb = L.append_block context "merge" the_function in
         (* Add branch at end of condition block based on bool_val *)
         ignore (L.build_cond_br bool_val loop_bb merge_bb cond_builder);

         L.builder_at_end context merge_bb

      | S.Cross((p1,_), (frac,_), (p2,_)) ->
         let p1' = expr env builder p1
         and p2' = expr env builder p2
         and frac' = expr env builder frac
         in

         let strategy1 = L.build_load (build_get_field builder p1'
           (L.const_int i32_t 0)) "cross1" builder
         and strategy2 = L.build_load (build_get_field builder p2'
           (L.const_int i32_t 0)) "cross2" builder
         in

         let nplayers1 = L.build_load (build_get_field builder strategy1
           (L.const_int i32_t 0)) "nplayers" builder in
         let nplayers2 = L.build_load (build_get_field builder strategy2
           (L.const_int i32_t 0)) "nplayers" builder in
         let nmoves1 = L.build_load (build_get_field builder strategy1
           (L.const_int i32_t 1)) "nmoves" builder in
         let nmoves2 = L.build_load (build_get_field builder strategy2
           (L.const_int i32_t 1)) "nmoves" builder in
         let nstates1 = L.build_load (build_get_field builder strategy1
           (L.const_int i32_t 2)) "nstates" builder in
         let nstates2 = L.build_load (build_get_field builder strategy2
           (L.const_int i32_t 2)) "nstates" builder in

         build_print_error builder "Number of players doesn't match" L.Icmp.Ne
           nplayers1 nplayers2;
         build_print_error builder "Number of moves doesn't match" L.Icmp.Ne
           nmoves1 nmoves2;
         build_print_error builder "Number of states doesn't match" L.Icmp.Ne
           nstates1 nstates2;

         let temp1 = build_strategy builder nplayers1 nmoves1 nstates1 in
         let temp2 = build_copy_strategy builder strategy1 temp1 frac'
           (L.const_float f64_t 0.0) in
         ignore (build_copy_strategy builder strategy2 strategy1 frac'
```

```
        (L.const_float f64_t 0.0));
      ignore (build_copy_strategy builder temp2 strategy2 frac'
        (L.const_float f64_t 0.0)); builder

  | S.Mut((p1,_),(frac,_)) ->
      let p1' = expr env builder p1
      and frac' = expr env builder frac
      in
      let strategy1 = L.build_load (build_get_field builder p1'
        (L.const_int i32_t 0)) "mutate" builder in
      ignore (build_copy_strategy builder strategy1 strategy1
        (L.const_float f64_t 1.0) frac'); builder

  | S.Play(pl,(g,_)) ->
      let pl' = List.map (expr env builder) (List.map fst pl) in
      let g' = expr env builder g in
      let intplayers = List.length pl in
      let nplayers = (L.const_int i32_t intplayers) in
      let game_nplayers = L.build_load (build_get_field builder g'
        (L.const_int i32_t 0)) "nplayers" builder in
      let game_nmoves = L.build_load (build_get_field builder g'
        (L.const_int i32_t 1)) "nmoves" builder in
      build_print_error builder "Number of players doesn't match" L.Icmp.Ne
        nplayers game_nplayers;

      let play_env = child_env env in
        ignore (add_to_env play_env builder (Array(Int), "moves"));
        ignore (L.build_store (build_array builder i32_t nplayers)
          (lookup play_env "moves") builder);

      (* Store all players' output moves *)
      ignore (List.fold_left (fun i player ->
        let output = build_get_player_move builder player in
        let access = build_array_access builder (lookup play_env "moves")
          (L.const_int i32_t i) in
        ignore (L.build_store output access builder); i+1) 0 pl');

      let moves = (lookup play_env "moves") in
      ignore (List.fold_left (fun i player ->
        (* Load strategy information *)
        let strategy = L.build_load (build_get_field builder player
          (L.const_int i32_t 0)) "strategy" builder in
        let strategy_nplayers = L.build_load (build_get_field builder
          strategy (L.const_int i32_t 0)) "nplayers" builder in
        let strategy_nmoves = L.build_load (build_get_field builder
          strategy (L.const_int i32_t 1)) "nmoves" builder in

        build_print_error builder "Number of players doesn't match"
```

```
            L.Icmp.Ne strategy_nplayers game_nplayers;
          build_print_error builder "Number of moves doesn't match"
            L.Icmp.Ne strategy_nmoves game_nmoves;

          let game_payoff = L.build_load (build_moves_payoff_access builder g'
            i moves intplayers) "game_payoff" builder in

          (* Load player information *)
          let delta = L.build_load (build_get_field builder player
            (L.const_int i32_t 1)) "delta" builder in
          let payoff = L.build_load (build_get_field builder player
            (L.const_int i32_t 2)) "payoff" builder in
          let state = L.build_load (build_get_field builder player
            (L.const_int i32_t 3)) "state" builder in
          let rounds = L.build_load (build_get_field builder player
            (L.const_int i32_t 4)) "rounds" builder in

          (* Update information *)
          let discount = build_pow builder delta rounds in
          let temp_payoff = L.build_fmul game_payoff discount "tmp" builder in
          let new_payoff = L.build_fadd payoff temp_payoff "newpayoff" builder
          in
          let trans = build_moves_trans_access builder strategy intplayers
            moves state in
          let new_state = L.build_load (build_get_field builder trans
            (L.const_int i32_t 1)) "newstate" builder in

          let new_rounds = L.build_add (L.const_int i32_t 1) rounds
            "newrounds" builder in

          let payoff = build_get_field builder player (L.const_int i32_t 2) in
          let state = build_get_field builder player (L.const_int i32_t 3) in
          let rounds = build_get_field builder player (L.const_int i32_t 4) in

          (* Store information *)
          ignore (L.build_store new_payoff payoff builder);
          ignore (L.build_store new_state state builder);
          ignore (L.build_store new_rounds rounds builder); i+1
        ) 0 pl');
        builder

    | S.Return(e,_) ->
    ignore (match fdecl.S.typ with
          Void -> L.build_ret_void builder
        | _ -> let e' = expr env builder e in L.build_ret e' builder); builder
  in

  (* Build statements in function by putting them in a block *)
```

```
    let builder = stmt local_env builder
      (S.Block (fst fdecl.S.body, snd fdecl.S.body)) in

    add_terminal builder (match fdecl.S.typ with
        Void -> L.build_ret_void
      | typ -> L.build_ret (lconst_of_typ typ))
  in

  List.iter build_function_body functions;
  the_module
```

Listing 7: `codegen.ml`