A Quick Introduction to Soccer-Fun

Peter Achten
Radboud University Nijmegen, Netherlands
P.Achten@cs.ru.nl

March 10, 2017

Summary

This note is a small manual for programming football brains in Soccer-Fun. It is assumed that the reader has programming experience with the functional programming language Clean.

1 The brain

The main subject in the Soccer-Fun framework is the footballer's brain. In an interview with *De Tijd* (7^{th} of may 1982) Johan Cruijff stated the following [translated by author]:

> "If I am in possession of the ball and I want to play it to another team member, I need to take my direct opponent into account, as well as the wind, the grass, and the speed with which other players run. We compute the strength with which to kick and the direction in which to do so in a tenth of a second. The computer needs two minutes for this!"

One way of looking at this statement, is that Johan Cruijff says that a footballer makes a decision at every moment, and that this decision depends on a large number of parameters. This is exactly what a function does: given a number of parameters, yield a result that is dependent on these parameters. The Soccer-Fun framework is based on this idea. We model the "brain" of a footballer with a function. For convenience, the parameters are 'bundled' into one single argument (a *struct* or *record* with the name `BrainInput`). The outcome of the brain is also bundled, with name `BrainOutput memory`. A footballer has a "memory" (`memory`). It is also a parameter of the brain function and can be altered by the brain function. The entire brain function has type `FootballerAI memory` and is defined as follows:

```
:: FootballerAI memory :== (BrainInput, memory) -> (BrainOutput, memory)
:: BrainInput          =   { referee   :: [RefereeAction]                    1.1
                           , football  :: FootballState                      1.2
                           , others    :: [Footballer]                       1.3
                           , me        :: Footballer                         1.4
                           }
:: BrainOutput         :== FootballerAction
```

In this chapter each of these components of `BrainInput` are discussed. The role of the memory is explained in section 1.5. Very simple football brains have no need for a memory. In chapter 2 we discuss the possible actions that a footballer can decide to do. These are values of type `BrainOutput` (which is a synonym type of `FootballerAction`).

1.1 RefereeAction

During a match players are supposed to listen to the decisions of the referee. These decisions are received as a list of `RefereeAction` values. This is a rather extensive algebraic data type. We do not show its definition (you find it in `Footballer.dcl`), but point out the possible decisions:

Five `RefereeAction` values concern the detection of rule violations: `Hands`, `OwnBallIllegally`, `DangerousPlay`, `Offside`, and `TackleDetected`. A player *id* who violates a rule receives a reprimand *r* as (`ReprimandPlayer` *id r*). The reprimand *r* can be a warning (`Warning`), a yellow card (`YellowCard`) or even a red card (`RedCard`). A player who receives two yellow cards, also receives a red card. Players who receive a red card are normally dismissed from the game (but this must be enforced by the referee).

Six `RefereeAction` values concern the duration of a match or training: (`Pause`/`Continue`)-`Game`, `EndHalf`, `GameOver`, (`AddTime` *t*), where *t* is the extra number of minutes, and finally (`GameCancelled` *mh*), where *mh* identifies the team that wins the match, if any. A reason for not pointing out a match winner is for instance when too many players of a team are dismissed from a game.

During a match players need to keep track at what half they are playing. This can be detected and remembered as follows:

```
brain ( {referee}, memory=:{second_half} )
    = (..., { memory & second_half = second_half || end_1st_half })
where
    end_1st_half = any isEndHalf referee
    ...
```

using `False` as initial value for `second_half`.

Whenever the referee decides that a team has scored a goal, this is notified by (`Goal` *h*) where *h* is a value of type `:: Home = West | East`. This value points to the playing half of the team that has scored the goal. After a goal has been scored by team *h*, a (`CenterKick` (`other` *h*)) is given. The function `other` is defined for domains that have two values only, and yields the 'other' value in comparison to its argument.

A match can be resumed by a team *h* on position *p* with a (`DirectFreeKick` *h p*) and (`ThrowIn` *h p*) or (`GoalKick` *h*) and (`Penalty` *h*) (actually, the latter is only registered, but is not yet fully implemented). Finally, a match can be resumed via a (`Corner` *h e*) where *e* is a value of type `:: Edge = North | South`. In general, when resuming a match the referee displaces players s_i to a new position p_i to make sure that they have sufficient distance to the ball. This is notified via (`DisplacePlayers` `[`(s_0, p_0),... (s_n, p_n) `]`). If pausing the match after detecting a rule violation would give the offended party *h* an additional disadvantage, the referee can decide to let the game continue, and notifies this via (`Advantage` *h*).

For training sessions the referee can also emit text messages *t* to the user via (`TellMessage` *t*).

1.2 FootballState

Obviously, in football we have a ball. The ball can be 'free' on or above the football field (`Free` *ball*) or is 'possessed' by a player with identification *id* (`GainedBy` *id*). This is expressed with the type `FootballState`:

```
::  FootballState = Free Football | GainedBy FootballerID
```

The ball itself is represented with a record type with name `Football`:

```
:: Football = { ballPos :: Position3D, ballSpeed :: Speed3D }
```

At any moment the ball is on some position (`ballPos`) and has some speed (`ballSpeed`). Let's have a look at the position first. A position is split in a *ground position* `pxy` that consists of a x-coordinate and y-coordinate (`px` and `py` of the record type `Position` and a height above the football field (`pz`).

```
:: Position3D = { pxy :: Position, pz :: Metre }
:: Position   = { px  :: Metre,    py :: Metre }

m :: Real -> Metre
```

The advantage of this split is that it is simple to obtain the ground position of the ball: if `ball` is a `Football`, then `ball.pxy` is its ground position. This is useful to compute the bearing of the ball and the distance between yourself and the ground position of the ball.

Distances are expressed in *metres*. You can convert a *decimal value* of type `Real` to a distance of type `Metre` with the function `m`: so (`m 10.0`) is the distance $10.0m$.

Soccer-Fun uses a coordinate system that is similar that you have learned during math at high school: the origin (with coordinates $(0, 0)$) is exactly in the middle of the football field, the x-coordinate increases from left to right, and the y-coordinate increases from below to up. The differences are that the x-coordinate is identified as `px` and the y-coordinate as `py`, and that instead of using decimal numbers values of type `Metre`.

The size of a football field is given by a record of type `FootballField`:

```
::  FootballField = { fwidth :: FieldWidth, flength :: FieldLength }
::  FieldWidth  :== Metre
::  FieldLength :== Metre
```

The width of a football field (`fwidth`) can vary between (`m 64.0`) and (`m 75.0`), and its length (`flength`) can vary between (`m 100.0`) and (`m 110.0`). Hence, the four corners of a football field `v` have the following coordinates:

- the point `{ px = scale -0.5 v.flength, py = scale 0.5 v.fwidth }` is at the left-top corner (`Home` value `West` and `Edge` value `North`);
- the point `{ px = scale -0.5 v.flength, py = scale -0.5 v.fwidth }` is at the left-bottom corner (`Home` value `West` and `Edge` value `South`);
- the point `{ px = scale 0.5 v.flength, py = scale 0.5 v.fwidth }` is at the right-top corner (`Home` value `East` and `Edge` value `North`);
- the point `{ px = scale 0.5 v.flength, py = scale -0.5 v.fwidth }` is at the right-bottom corner (`Home` value `East` and `Edge` value `South`).

These definitions also show that you cannot simply multiply metres (because that gives you values of type square metres). Instead, a function is provided, `scale`, that gets a numeric value as first argument, and as second argument the dimensional value that needs to be multiplied (other options are multiplying seconds (`Seconds`), minutes (`Minutes`), velocity (`Velocity`), angles (`Angle`) and vectors (`RVector` and `RVector3D`)). The computation (`scale -0.5 v.flength`) returns the western-most x-coordinate on a football field `v`.

Similar to the position, the speed of a ball is split in a ground speed (`vxy`) and a height speed (`vz`).

```
:: Speed3D = { vxy       :: Speed, vz       :: Velocity }
:: Speed   = { direction :: Angle, velocity :: Velocity }

rad       :: Real -> Angle
degree    :: Int  -> Angle
ms        :: Real -> Velocity
```

The ground speed has a `direction`. The direction, of type `Angle`, can be written in *radians* or *degrees*. For *radians* the value (`rad 0.0`) points straight east, (`rad (0.5*pi)`) points straight north, (`rad pi`) points straight west, and (`rad (1.5*pi)`) points straight south. For *degrees* the value (`degree 0`) points straight east, (`degree 90`) points straight north, (`degree 180`) points straight west, and (`degree 270`) points straight south. Note that radians use decimal values, and degrees use integer values.

In addition to the direction, a scalar velocity (of type `Velocity`) is used, both for the ground velocity and the z-axis velocity. This `Velocity` is expressed in metres/second. The velocity $10.0\frac{m}{s}$ is denoted with (`ms 10.0`). Note that the ground velocity is always ≥ 0, but that the z-axis velocity can be positive (ball moves up), or negative (ball moves down), or (`ms 0.0`) (ball neither moves up nor down).

Most football brains want to know where the ball is. The function `getBall` computes this for you:

```
getBall :: BrainInput -> Football
```

A typical use-case of this function is:

```
brain (input,memory)
    = ...
where
    ball = getBall input
    ...
```

1.3 All other players

The third field of the `BrainInput` record, `others`, enumerates all players except yourself. Use this information to find out the whereabouts of your team members, whether one of them has gained the ball, if it makes sense to pass the ball to them, and similarly for your opponents. In module `FootballerFunctions` you find two infix operators, `team` and `opponents`, that figure out your team members and your opponents. A typical use-case in a brain function is:

```
brain (input=:{others, me},memory)
    = ...
```

```
where
    my_team      = me team      others
    my_opponents = me opponents others
```

Here, `my_team` contains all players of your team (except yourself), and `my_opponents` are all your opponents. All players are of type `Footballer`. This type is described in the next section.

1.4 You: `Footballer`

The last field of the `BrainInput` record, `me`, represents *you*, which makes it a rather important argument. A footballer is a record of type `Footballer`:

```
:: Footballer = E. memory:
                { playerID :: FootballerID                          1.4.1
                , name     :: String                                1.4.2
                , length   :: Length                                1.4.3
                , pos      :: Position                              1.4.4
                , speed    :: Speed                                 1.4.5
                , nose     :: Angle                                 1.4.6
                , skills   :: MajorSkills                           1.4.7
                , effect   :: Maybe FootballerEffect                1.4.8
                , stamina  :: Stamina                               1.4.9
                , health   :: Health                                1.4.10
                , brain    :: Brain (FootballerAI memory) memory    1.4.11
                }
```

Each of these components are discussed below.

1.4.1 `playerID`

The `playerID` field identifies the players. It is a record type:

```
::  FootballerID = { clubName :: ClubName
                   , playerNr :: PlayersNumber
                   }
```

The only restrictions are that the goal keeper has `playerNr` 1, and that all players from the same team have *different* `playerNr` values, and play for the same `clubName`. The Soccer-Fun framework uses this identification to distinguish players. As a result, it is not permitted that two teams with the same `clubName` play a match. The team definitions that are already provided take this into account by adding the starting playing half to their name (`"_W"` or `"_E"`).

1.4.2 `name`

This is the player name, which can be any `String`. It is displayed on screen next to the player. It is useful to give each player a different (short) name, but it is not required.

1.4.3 `length`

This is the length of a player, expressed in metres. The length of a player is constant during a match. The length that you give is corrected by the framework to make sure it is a value between the constants `min_length` and `max_length`. Shorter players have a relative advantage in gaining the ball, tire a bit less while sprinting, can dribble the ball slightly better, sustain less damage when falling, and can rotate faster. Longer players have a relative advantage in the gain distance, kicking and heading the ball, tire a bit less while running, and can kick the ball further away.

1.4.4 `pos`

This is the position (`Position`) of a player. As explained in section 1.2 `Position` is a ground coordinate, which is a pair of a x-coordinate (`px`) and a y-coordinate (`py`). Hence, players always find themselves on the ground.

1.4.5 `speed`

This is the speed of a player. Because footballers are always on the ground, their speed is only a ground speed (see 1.2). A player with speed `{direction = a, velocity = v}` moves in direction (`a`) and has velocity (`v`).

1.4.6 `nose`

This is the looking direction of the player. The looking direction influences the effectiveness of most actions: running forward is easier than running backward, kicking the ball forward is more effective than backwards or sideways, and so on. A player with `nose = rad 0.0` looks straight east, with `nose = rad (0.5*pi)` looks straight north, and so on. A player with `nose = rad pi` and `speed = {direction = rad (1.5*pi), velocity = ms 5.0}` looks straight west (because of his `nose`) and runst straight south (because of his `direction`) with a velocity of $5.0\frac{m}{s}$.

1.4.7 `skills`

Every footballer can select *three* major skills, which influence his actions beneficially. There are ten skills to select from:

```
:: MajorSkills :== (Skill,Skill,Skill)
:: Skill = Running  | Dribbling | Rotating   | Gaining
         | Kicking  | Heading   | Feinting   | Jumping
         | Catching | Tackling
```

A footballer with major skill `Running` can move faster when not possessing the ball: this is a useful skill for a defender. An attacker is more interested in `Dribbling`: he can move faster when possessing the ball. `Rotating` gives a player an extra edge while rotating. `Gaining` increases the odds of successfully gaining the ball. `Kicking` increases the kicking distance and accuracy when playing the ball. `Heading` increases the heading accuracy, which is a useful skill for taller players. `Feinting` increases the feint distance, which is useful for attackers to avoid defenders. Although players are always on the football field, they sometimes need to jump in order to gain or head the ball. The `Jumping` skill increases their range. The only

major skill that is of additional importance to a goal keeper is `Catching`. Every player can decide to catch the ball, but this is only allowed for the goal keeper within the penalty area. The final major skill is `Tackling`. A player can attempt to hinder another player in such a way that he falls to the ground, and thus cannot participate in the game for a short while. Note that this can decrease the health of both the tackling player and the target, and that the referee may detect the tackle or think it is a case of dangerous play.

In your team, try to assign the right mixture of major skills that fit your team players best. Choose quick, agile attackers who can dribble well with the ball or kick well, and good runners with a wider gaining reach for defenders. A good goal keeper is probably tall, can jump well, and gain the ball well.

1.4.8 `effect`

This is the information that is fed back to the footballer to tell him how well the previous action has succeeded. This information can be useful for very advanced football brains. It is described in detail when we discuss the possible actions of a footballer in chapter 2.

1.4.9 `stamina`

Every footballer has an amount of stamina (`Stamina`). This is a value between 0.0 and 1.0. Your stamina influences every action: if you are more tired, then it is less likely that your actions are succesful. If your footballer has to keep running on sprinting speed all the time, you will notice a firm decrease of stamina, and that your player kicks the ball less precise or hardly ever gains the ball from an opponent. Stamina can be increased by making your player rest. Resting can be done by standing still, but also by walking.

1.4.10 `health`

Every footballer has an amount of `Health`. Just like `Stamina` this is a value between 0.0 and 1.0, and it influences every action. Your health can decrease because of tackling actions, or because you or another player hits you while running at high speed. An advanced footballer brain will try to avoid collisions. Your health does not increase during a match. Therefor, it is possible that you get eliminated by yourself or your opponents (or even your team members, which is a bit sad).

1.4.11 `brain`

The final component of a `Footballer` is the most important one, the brain. It is not the case that you can access the brain of other footballers during a match (footballers can not read each other's minds). You program the brain function, and assign it to a footballer, but afterwards it gets locked and you cannot access it. The brain is protected like a *black box*, which is indicated by the keyword `E. memory` at the start of the `Footballer` record. This keyword guarantees that you or any other player can access your brain function, so you can ignore this field of any footballer.

A brain consists of two parts: a memory and a brain function that uses the memory. It is the brain function that we are discussing right now. Its type is explained at the start of this chapter on page 1. The data structure that describes a brain therefor consists of two parts:

```
:: Footballer  = E. memory:
                 { ...
                 , brain :: Brain (FootballerAI memory) memory
                 }
::  Brain ai m = { ai    :: ai, memory :: m }
```

The memory is explained below.

1.5 Your memory

In addition to the `BrainInput` record, your footballer also has a *memory* (`memory`) that is passed as argument to your brain function, and that is returned by your brain function. Hence, the content of the memory can depend on the history of actions of your player. It can be any data structure that you deem suited for your player.

The simplest memory remembers nothing. You can use the trivial type `Void` for this purpose:

```
:: Void = Void
```

This type is predefined, so you can use it out of the box. A brain function that leaves the memory 'untouched' can do so by returning it unchanged:

```
brain (input, memory)
    = (action, memory)
where
    action =  computation that results in action
```

A more interesting use of a player's memory is to store the 'favorite position' of a player on the field. This will help you distribute your players over the football field. When you want to use the memory, it is a good idea to use a record type for this because it easier to extend in the future with extra record fields. Now suppose we wish to store the favorite position of a player in his memory. One way of establishing is as follows:

```
:: Memory = { favorite_position :: Position }
```

You can write a brain function that makes sure that a footballer always returns to his favorite position on the field. The brain's decision is: if the player is within an acceptable distance to the favorite position, he can stand still; if he is not within an acceptable distance to the favorite position, he should move towards it. In that case the main structure of the brain function looks like:

```
brain :: (BrainInput,Memory) -> (BrainOutput,Memory)
brain ({ me }, memory=:{ favorite_position } )
| distance_to_favorite_position < acceptable_distance
    = (stand_still, memory)
| otherwise
    = (run_to_position, memory)
where
    distance_to_favorite_position = dist me favorite_position
    acceptable_distance           = m ...
    stand_still                   = ...
    run_to_position               = ...
```

You will notice that very often you need to compute the bearing between different objects: for instance your bearing to the ball, your bearing to a team member to whom you wish to play the ball, your bearing to the goal posts to attempt to score a goal, and so on. For these computations you can use the function `bearing`:

```
bearing :: Angle base target -> Angle | toPosition base & toPosition target
```

Suppose you wish to compute the angle between an object `base` with an object `target`, then you can dan do this with (`bearing zero base target`). The first argument gives the starting angle from which you wish to compute the bearing. If you are interested in absolute directions (for moving) this first parameter is always zero (straight east). For the angle value zero you can use `zero` or `rad 0.0` or `degree 0`.

A complete example using this function is:

```
brain :: (BrainInput, Memory) -> (BrainOutput, Memory)
brain ({ me }, memory=:{ favorite_position })
| distance_to_favorite_position < acceptable_distance
    = (stand_still, memory)
| otherwise
    = (ren_naar_positie, memory)
where
    distance_to_favorite_position
                        = dist me favorite_position
    acceptable_distance = m 2.0
    stand_still         = Move zero zero
    run_to_position     = Move { direction = bearing zero me favorite_position, velocity = ms 5.0 }
                           (bearing me.nose me favorite_position)
```

2 The actions of a footballer

In the previous chapter all parameters have been discussed that are provided to the brain function of a footballer. With this information the brain function needs to compute two new values:

1. an *action* that has to be performed: this is a value of type `FootballerAction`. It is presented in this chapter.

2. A new value for *memory*. This has been presented extensively in section 1.5.

Before we discuss each of the possible `FootballerAction`s, we first need to address two important principles of the Soccer-Fun framework:

The brain function generates an intention: The action that has been conjured by the brain of a footballer has to be transferred to 'reality'. For instance, the brain might think that the footballer should stand still *right now* (by conjuring the value `(Move zero zero)`), but if the footballer is actually sprinting at full speed, it is impossible to stand still immediately. Hence, the decisions of the football brain are *intentions*. The Soccer-Fun framework transfers these intentions to concrete actions in a realistic way.

Actions can fail or be inaccurate: In real life football you need to take into account that your actions do not always have the desired effect. For instance, you kick the ball to score a goal, but for some reason the ball deviates from its intended course and misses the goal by several metres. The Soccer-Fun framework adds to each action a deviation. This deviation increases if you have less stamina (see 1.4.9) or are less healthy (see 1.4.10). Your major skills (see 1.4.7) decrease the deviation.

When testing your footballers it can be useful not to have these deviations. You can switch it on and off with the command Game:Mode:Predictable. For a realistic match you should use the command Game:Mode:Realistic.

In module `Footballer` you find a number of functions that you can use to determine whether or not it makes sense to perform a certain action. These are the functions `maxGainReach` up to `maxFeintStep`. The function `maxGainReach` yields the distance within which you can still gain the ball, and the function `maxFeintStep` yields the maximal distance that you can move when performing a feint maneuver.

Because brain intentions need to transferred to real actions that may fail or deviate, the football brain is informed about the success of the previous intention as an effect of type `Maybe FootballerEffect` (the field `effect`, see 1.4.9).

The football brain can conjure the following actions for a footballer, represented by means of an algebraic data type `FootballerAction`:

```
::  FootballerAction
    =   Move      Speed Angle                                  2.1
    |   Feint     FeintDirection                               2.2
    |   KickBall  Speed3D                                      2.3
    |   HeadBall  Speed3D                                      2.4
    |   GainBall                                               2.5
    |   CatchBall                                              2.6
    |   Tackle    FootballerID Velocity                        2.7
```

We present each of the possible actions.

2.1 Move

One of the two ways for a footballer to move is the action `(Move` s a `)` (the other way is `(Feint` d `)`, see 2.2), where s is a value of type `Speed`, and a a value of type `Angle`.

A footballer has a *looking direction* which is set in the field `nose` of a footballer (see 1.4.6). Moving with the action (`Move` s a) *first* alters the looking direction of the footballer with angle a. Hence this is a *relative* rotation. The looking direction influences the rate of success of the speed s: it is 100% in the current looking direction `nose` and minimal in the opposite direction (which depends, amongst others, on the `Running` skill, see 1.4.7). A player who has gained the ball also runs slower because he needs to control the ball (this factor is influenced in a positive way by means of the `Dribbling` skill, see 1.4.7). The direction s`.direction` is *absolute*.

The success of this action is fed back as `(Moved` s' a' `)`, where s' and a' represent the real speeds and rotations that your footballer managed to perform.

2.2 Feint

A footballer can decide to perform a *feint maneuver* using the `(Feint` d `)` action, where d is of type `FeintDirection`. The footballer can make a feint maneuver *to the left* (d has value `FeintLeft`) or a feint maneuver *to the right* (d has value `FeintRight`). This action is always successfull, but it is influenced by the skill of the footballer and his speed. A feint maneuver never alters his speed, but only his position.

The success of this action is fed back as `(Feinted` d `)`, where d has exactly the same value as in the conjured corresponding action.

2.3 KickBall

You can decide to kick the ball using a `(KickBall` v `)` action, where v is a value of type `Speed3D`. This implies that you can play the ball through the air. This is usually a good thing to do because the air friction is much less than the football field friction, so the ball will move further away. Additionally, it will be harder to gain such a ball by your opponents (but also your team members).

You do not have to have gained the ball first to perform this action. If the ball is free, and within kicking range, you can also kick the ball. This action can fail or have a deviation. The success of this action is fed back as `(KickedBall` mv' `)` where mv' is a 'maybe' value of type `(Maybe Speed3D)`. If you did not succeed in kicking the ball, then mv' is `Nothing`. If you did succeed in kicking the ball, then mv' is `(Just` v' `)`, where v' is the final speed that has been given to the ball.

2.4 HeadBall

This action is actually very similar to kicking the ball, except that you use your head to play the ball. The advantage of heading is that you can alter the speed of the ball while it is in the air. This will increase the rate of the game.

The success of this action is fed back in a similar way, using `(HeadedBall` mv' `)`.

2.5 GainBall

With this action the footballer can attempt to gain the ball. This may fail, for instance because the ball is already gained by another player who manages to keep it. Another reason for failure can be that you tried it too early, and the ball is just out of gaining reach.

The success of this action is fed back as `(GainedBall` s `)`, where s is the value `Success` (you actually gained the ball) or `Fail` (you did not gain the ball). Another way of inspecting whether you gained the ball is by inspecting the `FootballState` (1.2) parameter of your brain function next time.

2.6 CatchBall

The `CatchBall` action is only legal for goal keepers withing their penalty area. It is very similar to `GainBall`. The success of this action is fed back with `(CaughtBall` s `)`, where s is the success, similar to `GainBall` in 2.5.

Field players or goal keepers outside of their penalty area who perform a `CatchBall` can be reprimanded by the referee.

2.7 Tackle

A hazardous football action is the *tackle*. A footballer can decide to tackle another player with `playerID` s using a certain velocity v with the action `(Tackle` s v `)`. If the *tackle* is successfull, this will make the tackled player fall and stay on the ground for a short while. Additionally, his health can decrease by such an action. The referee can decide to reprimand the offender, so you run a risk that your footballer gets dismissed from the match.

The success of this action is fed back with `(Tackled` s v *success* `)`, where s and v have the same values as in the action, and *success* is similar as in `GainBall` (see 2.5). The player who has fallen gets to know this via the effect (`OnTheGround` n). In that case, the speler remains n ($n > 0$) *frames* on the ground, and all decisions by his brain function will be neglected.

3 Play football

You can add a team in Soccer-Fun by creating a new implementation module. Suppose your team is called `MyTeam`. Create an implementation module:

```
implementation module MyTeam

import StdEnv, StdIO, Footballer, FootballerFunctions
```

In this module, add a function that expects its first argument of type `Home` and the second of type `FootballField`, and that returns a team of type `Team`.

```
MyTeam :: Home FootballField -> Team
MyTeam home field
```

You need the `home` argument to know whether to place your players for the `West` side or `East` side of the football field (and hence, score goals at the opposite side). The `field` argument tells you how big the football field is. One way to define your team is as follows:

```
MyTeam :: Home FootballField -> Team
MyTeam home field
| home == West     = westTeam
| otherwise        = eastTeam
where
    club           = "My␣Club"
    eastTeam       = mirror field westTeam
    name           = "MyTeam_" +++ if (home == West) "W" "E"
    westTeam       = [  player home field { px = scale (-0.5*dx) field.flength
                                          , py = scale ( 0.5*dy) field.fwidth
                                          } { clubName = club, playerNr = nr }
                     \\ (dx,dy) <- west_positions
                      & nr      <- [1..]
                     ]
    west_positions = [(0.0, 0.50),(0.20,0.30),(0.20,0.70),(0.23,0.50)
                     ,(0.50,0.05),(0.50,0.95),(0.60,0.50),(0.70,0.15),(0.70,0.85)
```

```
                ,(0.90,0.45),(0.90,0.55)]
```

In this definition, the initial team positions are defined for the `West` side. To avoid working with positions, they are expressed as percentages of half the field length (for the x-coordinates) half the field width (for the y-coordinates). The overloaded function `mirror` is used to mirror the positions and looking direction of all players with respect to the y axis. Because in a match teams can not have the same name, it is wise to give teams that start on west a different name than teams that start on east. Finally, the footballers are created by the function `player`, who receives also the `home` and `field`, and in addition, the initial position and player identification. This function defines the actual `Footballer`:

```
player :: Home FootballField Position FootballerID -> Footballer                1
player home field position playerID=:{ playerNr }                                2
    = { playerID    = playerID                                                   3
      , name        = "MyName." <+++ playerNr                                    4
      , length      = min_length                                                 5
      , pos         = position                                                   6
      , speed       = zero                                                       7
      , nose        = zero                                                       8
      , skills      = (Running, Kicking, Rotating)                               9
      , effect      = Nothing                                                    10
      , stamina     = max_stamina                                                11
      , health      = max_health                                                 12
      , brain       = { memory = my_memory, ai = my_brain }                      13
      }                                                                          14
```

Here, the player number is used to create a unique name for each player (line 4) (which is not strictly necessary for Soccer-Fun to work, but it is useful for yourself to see what each individual player is doing). We create short players (line 5). The initial position is passed on to the player (line 6). For players who start at the west-side of a football field, it makes sense that their initial looking direction is straight east, so their `speed.direction` and `nose` are both `zero` (line 7 and 8). In this example we have selected the major skills running, kicking, and rotating (line 9). At start, nothing has happened yet (line 10) and everybody has full stamina and health (line 11 and 12). Finally, we equip the player with a brain that has some initial value `my_memory` for its memory and `my_brain` for its intelligence (line 13).

3.1 Integration in *Soccer-Fun*

When you have built your team, you need to do two final actions to see it in action.

3.1.1 Export your team

First of all you need to make your team known by exporting the function `MyTeam`:

```
definition module MyTeam

import Footballer

MyTeam :: Home FootballField -> Team
```

This makes the team `MyTeam` available for the framework. This requires an *import* of the new module `MyTeam`:

3.1.2 Import your team

In Soccer-Fun all teams are collected in module `Team.icl`. This is done by the very first function in this module, `allAvailableTeams`. It enumerates all known functions of the same type as `MyTeam`. In order to do this, you first need to import these functions. This is how it is done in `Team.icl`:

```
implementation module Team                                              1
                                                                        2
import StdEnvExt                                                        3
import Footballer                                                       4
/* Import all standard teams: */                                       5
import TeamMiniEffie                                                    6
import Team_Opponent_Slalom_Assignment                                  7
import Team_Opponent_Passing_Assignment                                 8
import Team_Opponent_DeepPass_Assignment                                9
import Team_Opponent_Keeper_Assignment                                  10
import Team_Student_Slalom_Assignment                                   11
import Team_Student_Passing_Assignment                                  12
import Team_Student_DeepPass_Assignment                                 13
import Team_Student_Keeper_Assignment                                   14
// Marc Schoolderman's team:                                            15
import Team_Harmless                                                    16
// import your team:                                                    17
import MyTeam                                                           18
                                                                        19
allAvailableTeams :: [Home FootballField -> Team]                       20
allAvailableTeams = [ Team_MiniEffies, Harmless                         21
                    , Team_Student_Slalom                               22
                    , Team_Student_Passing                              23
                    , Team_Student_DeepPass                             24
                    , Team_Student_Keeper                               25
                    , Team_Opponent_Slalom                              26
                    , Team_Opponent_Passing                             27
                    , Team_Opponent_DeepPass                            28
                    , Team_Opponent_Keeper                              29
                    , MyTeam                                            30
                    ]                                                   31
```

In line 18 `MyTeam` is imported, and in line 30 it is included in the list of known teams. After bringing your project up-to-date, `MyTeam` has officially entered the Soccer-Fun competition! You can start a match and select teams with the command Game:Match.

`TeamMiniEffie` consists of players that all run to the ball and try to kick in the opponent's goal. The teams starting with `Team_Opponent_` and `Team_Student_` are concerned with the training exercises. Marc Schoolderman has created a good team to practice against, called `Team_Harmless`. Note that you do not have access to its source code. If your team manages to beat his team, you might become champion of a competition.