# BoGL Syntax

Martin Erwig

March 5, 2020

We generally adopt the Haskell rules for lexical syntax. In particular, *name* is used for function and parameter names and stands for alphanumeric strings starting with a lowercase letter. Similarly, *Name* is used for symbols (which are basically nullary constructors) and stands for alphanumeric strings starting with an uppercase letter. It is also used for the names of games. The nonterminal *int* stands for integer constants. Comments are also handled as in Haskell.

Unlike Haskell, we require all names (of values and types) to be defined before they can be used in other definitions.

A game definition consists of a name, a number of type definitions, followed by any number of value and function definitions. The type definitions must include a definition for a board and an input type. The board definition determines the board dimensions and the type of values that can appear on a board.

**Games and Type Definitions**

| | | | |
|---|---|---|---|
| *game* | ::= | `game` *Name typedef* board input typedef* valuedef** | (*game definition*) |
| *board* | ::= | `type Board = Array (`*int*`,`*int*`) of` *type* | (*board type*) |
| *input* | ::= | `type Input =` *type* | (*input type*) |
| *typedef* | ::= | `type` *Name* `=` *type* | (*type definition*) |

Value definitions (for values and functions) require a signature declaring their type plus an equation for defining the value. If a value is parameterized, that is, if it is followed by a parameter list, it represents a function.

A board definition (for a variable *name* with declared type `Board`) is equivalent to an array definition and may contain a number of definitions for individual positions (given by pairs of integers) and for sets of positions, which can be given by using a variable for either or both coordinates. The semantics of a definition such as `board!(x,2) = Empty` is to set all fields in the second row to `Empty`.

**Value Definitions**

| | | | |
|---|---|---|---|
| *valuedef* | ::= | *signature equation* | (*value definition*) |
| *signature* | ::= | *name* `:` *type* | (*type signature*) |
| *equation* | ::= | *name* `=` *expr* | (*value equation*) |
| | \| | *name parlist* `=` *expr* | (*function equation*) |
| | \| | *boarddef** | (*board definition*) |
| *boarddef* | ::= | *name* `! (`*pos*`,`*pos*`) =` *expr* | (*board equation*) |
| *pos* | ::= | *int* \| *name* | (*board positions*) |
| *parlist* | ::= | `(`$par_1$`, ... ,`$par_k$`)` | (*parameter list,* $k \geq 1$) |
| *par* | ::= | *name* \| *parlist* | (*parameter*) |

The type system builds on a collection of basic types, which include predefined numbers and booleans as well as the type `Symbol`, which is the type for all symbol values (*Name*). The special type `Input` is a synonym for the type of information that is gathered from the user in each move during an execution of the game.

An enumeration type is given by a set of symbols, and an extended type adds to a base type one or more symbols, which are values and can be thought of nullary data constructors. For example, the Haskell type `Maybe Int` can be represented by the extended type `Int & {Nothing}`. Extended types facilitate the extension of types by values for representing "special" situations. For example, `Player & {Empty}` is a type typically used in a board type definition.

To avoid ambiguities and to simplify type checking we require that a symbol can be used only in one enumeration type. The type of a symbol `A` used in a definition `type T = U & {...,A ,...}` should be `T`.

A tuple type can contain extended types, and a plain type is either a tuple type or an extended type. The language has first-order function types whose argument and result types can be any plain type.

**Types**

| | | | |
|---|---|---|---|
| *btype* | ::= | `Bool \| Int \| Input` | (*base type*) |
| *etype* | ::= | {*Name*$_1$, ..., *Name*$_k$} | (*enumeration type*) |
| *xtype* | ::= | *btype* \| *etype* \| *xtype* & *etype* | (*extended type*) |
| *ttype* | ::= | (*ptype*$_1$, ..., *ptype*$_k$) | (*tuple type*, $k \geq 2$) |
| *ptype* | ::= | *xtype* \| *ttype* | (*plain type*) |
| *ftype* | ::= | *ptype* `->` *ptype* | (*function type*) |
| *type* | ::= | *ptype* \| *ftype* | (*type*) |

Atomic expressions are either basic integer values or symbols. `True` and `False` are symbols of type `Bool`, all other symbols have either the type `Symbol` or the type of the type definition they occur in. For example, the type definition `type Player = {A,B}` that occurs in the BoGL prelude defines `A` and `B` to be symbols of type `Player`. Note that the case for infix application includes the Haskell notation for array lookup `board!(x,y)`.

**Expressions**

| | | | |
|---|---|---|---|
| *expr* | ::= | *int* | (*integer*) |
| | \| | *Name* | (*symbol*) |
| | \| | *name* | (*variable*) |
| | \| | (*expr*) | (*parenthesized expression*) |
| | \| | (*expr*$_1$, ... ,*expr*$_k$) | (*tuple*, $k \geq 2$) |
| | \| | *name*(*expr*$_1$, ... ,*expr*$_k$) | (*function application*) |
| | \| | *expr binop expr* | (*infix application*) |
| | \| | `let` *name* = *expr* `in` *expr* | (*local definition*) |
| | \| | `if` *expr* `then` *expr* `else` *expr* | (*conditional*) |
| | \| | `while` *expr* `do` *expr* | (*while loop*) |
| *binop* | ::= | `+ \| - \| == \| ! \| ...` | (*binary operation*) |