Sintaxis de Gobstones
Versión 3.0

Pablo E. Martínez López
fidel@unq.edu.ar

Eduardo Bonelli
ebonelli@unq.edu.ar

24 de noviembre de 2014

Este documento presenta una breve referencia de la estructura de un programa Gobstones en su versión 3.0, con la intención de servir de consulta rápida. Debe recordarse que el objetivo de Gobstones es plasmar una secuencia didáctica para la enseñanza de la programación en un nivel introductorio.

En la versión 3.0 para Gobstones realizamos algunos cambios con la mejora de la secuencia didáctica como guía. Estos cambios NO son compatibles con las versiones anteriores, por lo que un programa Gobstones en versión 2.12 o inferior NO es un programa Gobstones válido en su versión 3.0.

La idea original de Gobstones fue autoría de Pablo E. Martínez López y Eduardo A. Bonelli. Sin embargo, el lenguaje ha recibido valiosísimas contribuciones en su definición de un equipo grande de programadores, docentes e investigadores, quiénes conforman actualmente la comunidad Gobstones. Con el riesgo de dejar mucha gente afuera, pero sabiendo que son tantos los que hicieron contribuciones que es imposible nombrarlos a todos, quiero mencionar a quiénes, a nuestro entender, hicieron los aportes más significativos a la definición del lenguaje: Francisco Soulignac, Federico Sawady, Pablo Barenbaum y Daniel Ciolek. A ellos, y a todos los que contribuyeron a Gobstones, ¡gracias!

A. Sintaxis de Gobstones

Para la presentación se utiliza notación estilo BNF, que es común en la manera de transmitir estructura de lenguajes de programación. En esta notación se presentan conjuntos de elementos a través de un nombre, lo cual se escribe como

$$<conjunto>$$

y se asocia ese nombre con las diferentes formas que puede tener a través de una cláusula de formato llamada *producción*, la cual se escribe como

$$<conjunto> \quad \rightarrow \quad definición$$

y donde la definición puede contener alternativas separadas por el símbolo |, o sea,

$$<conjunto> \quad \rightarrow \quad alternativa_1 \mid alternativa_2 \mid \ldots \mid alternativa_n$$

y los símbolos en `negrita` determinan elementos finales.

Por ejemplo, la siguiente definición

$<conj>$ → $<elem_1>$ `->` $<elem_2>$
$<elem>$ → `X` | `Y`

determina que los elementos del conjunto $<elem>$ son `X` e `Y`, y los del conjunto $<conj>$ son `X->X`, `X->Y`, `Y->X` e `Y->Y`. Observar que $<elem_1>$ adopta todas las posibles formas de un elemento del conjunto $<elem>$, y lo mismo para $<elem_2>$, y que poner dos elementos juntos simplemente los coloca uno a continuación del otro.

La notación permite además elementos opcionales, escrito como

$[<elemOpcional>]$

la eliminación de los elementos de un conjunto de otro conjunto dado, escrito como

$(<conjBase>/(<conjAEliminar>))$

e incorpora notaciones para especificar cero o más apariciones de un elemento dado

$\{<elemento>\}*$

y una o más apariciones de este elemento

$\{<elemento>\}+$

Por ejemplo, modificando la definición anterior a

$<conj>$ → $<elem_1>$ [`->`$<elem_2>$]
$<elem>$ → `X` | `Y`

los elementos de $<conj>$ serían los de antes, más `X` e `Y`, pues el símbolo `->` y el siguiente elemento se anotaron como opcionales. Si en cambio se definiese

$<ej>$ → $(<conj>/(<idem>))$
$<conj>$ → $<elem_1>$ `->` $<elem_2>$
$<elem>$ → `X` | `Y`
$<idem>$ → `X -> X`

los elementos del conjunto $<ej>$ serían `X->Y`, `Y->X` e `Y->Y`. Observar que `X->X` está en el conjunto $<conj>$ pero no en el conjunto $<ej>$.

Con esta notación, podemos escribir la forma de un programa GOBSTONES, lo cual se lleva a cabo en las siguientes subsecciones.

A.1. Programas Gobstones

Un programa GOBSTONES es un elemento del conjunto $<gobstones>$. Cada programa está conformado por un cuerpo principal (que puede ser plano o interactivo) y por una lista de definiciones de procedimientos y funciones.

$<gobstones>$ → $<programdef> <defs>$ | $<defs> <programdef>$

```
<defs>          →  <def> | <def><defs>
<def>           →  procedure <procName> <params> <procBody>
                |  function <funcName> <params> <funBody>
<programdef>    →  <bprogdef> | <iprogdef>
<bprogdef>      →  program <programBody>
<iprogdef>      →  interactive program <iprogBody>

<params>   →   <varTuple>
```

El cuerpo de un procedimiento es una lista de comandos encerrados entre llaves. El cuerpo de una función es similar, excepto que termina con el comando `return`. El cuerpo principal del programa en su versión plana tiene un `return` opcional, pero solo de variables. El cuerpo principal de un programa interactivo es diferente, y se compone exclusivamente de una asociación entre teclas y bloques de código.

```
<procBody>      →  { <cmds> }
<funcBody>      →  { <cmds> return <gexpTuple1>[;] }
<programBody>   →  { <cmds> [return <gexpTuple1>[;]] }
<iprogBody>     →  { <keyassocs> }
```

Los comandos se definen en la siguiente sección (A.2), las tuplas de expresiones y variables, en la sección A.5, y los nombres de funciones y procedimientos, también en la sección A.5. Las asociaciones de teclas a bloques se definen en la sección A.4.

A.2. Comandos

Los comandos pueden ser simples o compuestos, y pueden estar agrupados en bloques.

```
<blockcmd>   →  { <cmds> }
<cmds>       →  [<necmds>[;]]
<necmds>     →  <cmd> | <cmd>[;] <necmds>
<cmd>        →  <simplecmd> | <compcmd>
```

Los comandos simples son los comandos básicos del cabezal, la invocación de procedimientos y la asignación (de variables, y de resultados de llamados a función).

```
<simplecmd>   →  Skip
              |  <procCall>
              |  <varName> := <gexp>
              |  <varTuple1> := <funcCall>
<procCall>    →  <proc> <args>
<proc>        →  <procName> | <predefProc>
<predefProc>  →  Poner | Sacar | Mover
              |  IrAlBorde | VaciarTablero
```

Las invocaciones a función y los argumentos para las invocaciones se describen en la sección A.3, y los nombres de variables en la sección A.5.

Los comandos compuestos son las alternativas (condicional e indexada), las repeticiones (condicional e indexada) y los bloques.

```
<compcmd>   →  if (<gexp>) [then] <blockcmd> [else <blockcmd>]
            |  switch(<gexp>) to <branches>
            |  repeat (<gexp>) <blockcmd>
            |  while (<gexp>) <blockcmd>
            |  foreach <varName> in <sequence> <blockcmd>
            |  <blockcmd>
```

```
<sequence>   →  [<seqdef>]
<seqdef>     →  <range> | <enum>
<enum>       →  <gexp> | <enum>,<gexp>
<range>      →  <gexp>..<gexp> | <gexp>,<gexp>..<gexp>
<branches>   →  _ -> <blockcmd>
             |  <lits> -> <blockcmd>[;] <branches>
<lits>       |  <literal> | <literal>, <lits>
```

El conjunto de literales <*literal*> se define en la sección A.3.

Observar que las condiciones de las alternativas y de la repetición condicional deben ir obligatoriamente entre paréntesis.

A.3. Expresiones

La expresiones se obtienen combinando ciertas formas básicas en distintos niveles. El nivel básico tiene las variables, las expresiones atómicas para indicarle al cabezal que cense del tablero, los literales, y las invocaciones de función y primitivas. Sobre ese nivel se construyen las expresiones aritméticas (sumas, productos, etc.) con la precedencia habitual. Sobre el nivel aritmético se construyen las expresiones relacionales (comparación entre números y otros literales) y sobre ellas, las expresiones booleanas (negación, conjunción y disyunción) también con la precedencia habitual.

```
<gexp>     →  <bexp>
<bexp>     →  <bterm> | <bterm>||<bexp>          (infixr)
<bterm>    →  <bfact> | <bfact>&&<bterm>         (infixr)
<bfact>    →  not <batom> | <batom>
<batom>    →  <nexp>
           |  <nexp><rop><nexp>
<nexp>     →  <nterm> | <nexp><nop><nterm>       (infixl)
<nterm>    →  <nfactH> | <nterm>*<nfactH>        (infixl)
<nfactH>   →  <nfactL> | <nfactL><mop><nfactL>
<nfactL>   →  <natom> | <nfactL>^<natom>         (infixl)

<natom>    →  <varName> | <liter> | -<natom>
           |  <funcCall>
           |  (<gexp>)

<rop>      →  == | /= | < | <= | >= | >
<nop>      →  + | -
<mop>      →  div | mod

<funcCall>     →  <func> <args>
<args>         →  <gexpTuple>
<func>         →  <funcName> | <predefFunc>
<predefFunc>   →  nroBolitas | hayBolitas | puedeMover
               |  siguiente | previo | opuesto
               |  minBool | maxBool
               |  minDir | maxDir
               |  minColor | maxColor
```

Las tuplas de expresiones se definen en la sección A.5.

Los literales pueden ser numéricos, booleanos, de color o de dirección.

```
<literal>  →  <literN> | <literB> | <literC> | <literD>
<literN>   →  <num>
<literB>   →  False | True
<literC>   →  Verde | Rojo | Azul | Negro
<literD>   →  Norte | Sur | Este | Oeste
```

La forma de los números se definen en la sección A.6

A.4. Programas interactivos

Los programas interactivos quedan definidos por una asociación entre especificaciones de teclas y bloques de código. La definición es la siguiente:

```
<keyassocs>          →  [<defaultkeyassoc>] | <keyassoc><keyassocs>
<keyassoc>           →  <keydef> -> <blockcmd>
<defaultkeyassoc>    →  _ -> <blockcmd>
```

Los bloques se definieron en la sección A.2. Las especificaciones de teclas se definen en la sección A.6.

A.5. Definiciones auxiliares

En esta sección se definen diversos conjuntos utilizados como auxiliares en las definiciones previas. Los nombres de variables y de funciones son identificadores que comienzan con minúsculas. Los nombres de los procedimientos son identificadores que empiezan con mayúsculas.

```
<varName>     →  <lowerid>
<funcName>    →  <lowerid>
<procName>    →  <upperid>
```

Las tuplas son listas de elementos encerrados entre paréntesis y separados por comas. Opcionalmente, una tupla puede estar vacía, o sea, no contener ningún elemento.

```
<varTuple>     →  () | <varTuple1>
<varTuple1>    →  (<varNames>)
<varNames>     →  <varName> | <varName>,<varNames>

<gexpTuple>    →  () | <gexpTuple1>
<gexpTuple1>   →  (<gexps>)
<gexps>        →  <gexp> | <gexp>,<gexps>
```

A.6. Definiciones lexicográficas

Las definiciones lexicográficas establecen la forma de las palabras que conforman el lenguaje. Ellas incluyen los números, los identificadores, las palabras reservadas, los operadores reservados y los comentarios.

Los números son simplemente secuencias de dígitos.

```
<num>          →  <digit> | <nonzerodigit> <digits> | -<num>
<digits>       →  <digit> | <digit> <num>
<digit>        →  0 | <nonzerod>
<nonzerod>     →  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Los identificadores son de dos tipos: los que comienzan con minúscula y los que comienzan con mayúscula. El símbolo de tilde (') puede ser parte de un identificador (excepto que no puede ser el primero de los símbolos). Las palabras reservadas no pueden ser identificadores.

```
<lowerid>      →  (<lowname>/(<reservedid>))
<lowname>      →  <lowchar> | <lowchar> <chars>
<lowchar>      →  a | ... | z

<upperid>      →  (<uppname>/(<reservedid>))
<uppname>      →  <uppchar> | <uppchar> <chars>
<uppchar>      →  A | ... | Z

<chars>        →  <char> | <char> <chars>
<basicchar>    →  <lowchar> | <uppchar> | <digit>
<char>         →  <basicchar> | ' | _
```

Las especificaciones de teclas son palabras reservadas que empiezan con `K_` (por tecla en inglés, *key*), y pueden ser las siguientes

```
<keydef>       →  K_<key>
<key>          →  <uppchar> | <digit> | <specialkey> | <arrowkey>
<specialkey>   →  SPACE | ENTER | TAB | BACKSPACE | DELETE | ESCAPE | CTRL_D
<arrowkey>     →  ARROW_LEFT | ARROW_RIGHT | ARROW_UP | ARROW_DOWN
```

Las palabras y los símbolos reservados son todos aquellos utilizados en algún comando predefinido o como separadores.

```
<reservedid>   →  if | then | else | not | True | False
               |  Verde | Rojo | Azul | Negro
               |  Norte | Sur | Este | Oeste
               |  switch | to | while | repeat | Skip | foreach | in
               |  interactive | program | procedure | function | return
               |  Mover | Poner | Sacar | IrAlBorde | VaciarTablero
               |  div | mod | siguiente | previo | opuesto
               |  hayBolitas | nroBolitas | puedeMover
               |  minBool | maxBool
               |  minDir | maxDir
               |  minColor | maxColor
<reservedop>   →  := | .. | _ | ->
               |  , | ; | ( | ) | { | } | [ | ]
               |  || | && | + | * | - | ^
               |  == | /= | < | <= | >= | >
               |  -- | {- | -}
               |  // | /* | */
               |  #  | """
```

Finalmente, los comentarios son de línea o de párrafo. Los primeros empiezan con uno de los símbolos reservados `--` o `//` y terminan con el fin de línea, y los segundos empiezan con los símbolos reservados `{-` o `/*` y terminan con la primera aparición del símbolo `-}` o `*/` respectivamente.

```
<comment>      →  <linecomm> | <parcomm>
<linecomm>     →  -- (<anySymbols>/(\n)) \n
               |  // (<anySymbols>/(\n)) \n
               |  # (<anySymbols>/(\n)) \n
<parcomm>      →  {- (<anySymbols>/(-})) -}
               |  /* (<anySymbols>/(*/)) */
               |  """ (<anySymbols>/(""")) """
```