Gramática de Gobstones v3.20

Pablo Barenbaum

2018

Índice

1. Gramática

1.1. Sintaxis léxica

- Se ignoran: `'\t'` (`TAB`, chr 9), `'\n'` (`LF`, chr 10), `'\r'` (`CR`, chr 13), `' '` (`SPACE`, chr 32).
- Se admiten comentarios comenzados por `"//"` (estilo C++), por `"--"` (estilo Haskell) y por `"#"` (estilo shell) que se extienden hasta el fin de línea (`LF`).
- Se admiten comentarios delimitados por `"/*" "*/"` y por `"{-" "-}"` que pueden anidarse.
- El tokenizador reconoce directivas *pragma* de la forma `"/*@parte`$_1$`@parte`$_2$`@...@parte`$_n$`@*/"`. La idea es que esto pueda ser un mecanismo extensible de directivas. Ver más abajo las directivas soportadas.

1.2. Lista de símbolos terminales

Todos los símbolos terminales se acompañan de su nombre EN_MAYUSCULAS.

- **Constantes numéricas.** Son una secuencia de dígitos decimales que representan enteros positivos. No pueden tener ceros innecesarios a la izquierda.

  NUM ::= `0|[1-9][0-9]*`

- **Identificadores.** Comienzan por un caracter alfabético y están seguidos por cero o más caracteres, ya sea alfabéticos, numéricos, guión bajo (_) o comilla simple ('). Se aceptan caracteres Unicode arbitrarios. Decimos que un caracter c es **alfabético** cuando tiene dos variantes, una minúscula y una mayúscula, es decir cuando:

  c`.toUpperCase() != `c`.toLowerCase()`

  Los identificadores LOWERID comienzan por un caracter en minúsculas (y sirven para identificar índices, parámetros, funciones, variables, campos). Los identificadores UPPERID comienzan por un caracter en mayúsculas (y sirven para identificador constructores, procedimientos, tipos).

  LOWERID ::= `[:lower:]([:alpha:]|[0-9]|_|')*`

  UPPERID ::= `[:upper:]([:alpha:]|[0-9]|_|')*`

- **Constantes de cadena.** Las cadenas están delimitadas por comillas dobles ('"'). Todos los caracteres desde la comilla que abre hasta la que cierra se toman literalmente salvo la contrabarra ('\') que se interpreta como un caracter de escape. Los escapes admitidos son:

  - \\ – contrabarra (\)
  - \" – comilla doble (")
  - \a – BEL, chr 7
  - \b – BS, chr 8
  - \f – FF, chr 12
  - \n – LF, chr 10
  - \r – CR, chr 13
  - \t – TAB, chr 9
  - \v – VT, chr 11

  STRING ::= `"(\a|\b|\f|\n|\r|\t|\v|\\|\"|[^\"])*"`

- **Lista completa de tokens.**

STRING	Constante de cadena.

	NUM	Constante numérica.
	LOWERID	Identificador empezado con minúsculas.
	UPPERID	Identificador empezado con mayúsculas.
program	PROGRAM	Para declarar la rutina principal.
interactive	INTERACTIVE	Para declarar una rutina principal interactiva con interactive program.
procedure	PROCEDURE	Para declarar procedimientos.
function	FUNCTION	Para declarar funciones.
return	RETURN	Para devolver valores de funciones y de la rutina principal.
if	IF	Para la alternativa condicional.
then	THEN	Palabra clave opcional para la rama 'then'.
elseif	ELSEIF	Para las ramas 'elseif'.
else	ELSE	Para la rama 'else'.
choose	CHOOSE	Para la expresión alternativa condicional.
when	WHEN	Para las condiciones del 'choose'.
otherwise	OTHERWISE	Para la última rama del 'choose' y el 'matching..select'.
repeat	REPEAT	Repetición simple.
foreach	FOREACH	Repetición indexada.
in	IN	Para declarar el rango de la repetición indexada.
while	WHILE	Repetición condicional.
switch	SWITCH	Para hacer pattern matching.
to	TO	Palabra clave opcional después del sujeto sobre el que se hace pattern matching.
matching	MATCHING	Para la expresión 'matching..select'.
select	SELECT	Para la expresión 'matching..select'.
on	ON	Para la expresión 'matching..select'.
let	LET	Palabra clave opcional para la asignación. Es obligatoria en el caso de las asignaciones de tuplas.
not	NOT	Negación lógica.
mod	MOD	Resto de la división entera.
div	DIV	Cociente de la división entera.
type	TYPE	Para la declaración de nuevos tipos.
is	IS	Palabra clave para acompañar la declaración de un nuevo tipo.
record	RECORD	Para tipos registro.
variant	VARIANT	Para tipos variantes.
case	CASE	Para las alternativas de tipos variantes.
field	FIELD	Para los campos de tipos registro.
_	UNDERSCORE	Para marcar el caso default en un switch/match.
TIMEOUT	TIMEOUT	Para la rama timeout en un interactive program.
{	LBRACE	Llave izquierda.
}	RBRACE	Llave derecha.
(	LPAREN	Paréntesis izquierdo.
)	RPAREN	Paréntesis derecho.
[	LBRACK	Corchete izquierdo (para listas y rangos).
]	RBRACK	Corchete derecho.

,	COMMA	Coma.
;	SEMICOLON	Punto y coma (separador de instrucciones opcional).
...	ELLIPSIS	Fragmento de programa intencionalmente incompleto.
..	RANGE	Para rangos.
:=	ASSIGN	Asignación.
&&	AND	Conjunción.
||	OR	Disyunción.
<-	GETS	Para inicialización de campos.
|	PIPE	Para actualización de campos.
->	ARROW	Usado en las ramas de un 'switch'.
==	EQ	Comparación por igualdad.
/=	NE	Comparación por desigualdad.
<=	LE	Comparación por menor o igual.
>=	GE	Comparación por mayor o igual.
<	LT	Comparación por menor estricto.
>	LT	Comparación por mayor estricto.
++	CONCAT	Operador de concatenación de listas.
+	PLUS	Suma de números.
-	MINUS	Resta de números y menos unario.
*	TIMES	Producto de números.
^	POW	Potencia.

1.3. Pragmas

1.3.1. Pragmas `BEGIN_REGION` y `END_REGION`

Las directivas `BEGIN_REGION` y `END_REGION` sirven para mantener registro de "regiones". Una región es una cadena de texto que sirve para identificar o *taggear* un fragmento del programa. Las regiones no tienen ningún significado especial para el intérprete de Gobstones, pero todas las excepciones que eleva el intérprete vienen acompañadas de una posición que incluye el nombre de la región actual. Las regiones pueden anidarse. Este comportamiento se implementa por medio de dos directivas.

- "`/*@BEGIN_REGION@`*nombre_de_la_región*`@*/`" — mete el nombre de una región en la pila de regiones.
- "`/*@END_REGION@*/`" — saca la región del tope de la pila de regiones.

Por ejemplo ante el siguiente programa:

```
/*@BEGIN_REGION@A@*/
procedure P() {
  /*@BEGIN_REGION@B@*/
  x := f(
  /*@END_REGION@*/
}
/*@END_REGION@*/
```

El parser idealmente debería reportar que hay un error de sintaxis en la región `B`.

1.3.2. Pragma `ATTRIBUTE`

La directiva `ATTRIBUTE` sirve para decorar definiciones de tipos, programas, procedimientos y funciones con atributos clave/valor. La directiva `ATTRIBUTE` debe **preceder** inmediatamente la definición de un tipo, programa, procedimiento o función, y su sintaxis es la siguiente:

- `"/*@ATTRIBUTE@clave@valor@*/"` — asocia el valor indicado a la clave indicada. Las claves y valores son *strings*.

Una definición de tipo, programa, procedimiento o función puede estar precedida de varias directivas `ATTRIBUTE`, asociando a dicha definición un diccionario de atributos. Por ejemplo en el siguiente programa:

```
/*@ATTRIBUTE@descripcion@Poner una piedra en la casilla actual.@*/
/*@ATTRIBUTE@esAtomico@SI@*/
procedure PonerPiedra() {
  ...
}

/*@ATTRIBUTE@esAtomico@NO@*/
function f() {
  ...
}

/*@ATTRIBUTE@descripcion@Programa principal.@*/
program {
  ...
}
```

El procedimiento `PonerPiedra` tiene asociado el siguiente diccionario de atributos (en formato JSON):

```
{
  descripcion: "Poner una piedra en la casilla actual.",
  esAtomico: "SI"
}
```

y la función `f` tiene asociado el siguiente diccionario de atributos:

```
{
  esAtomico: "NO"
}
```

El intérprete no otorga ningún significado ni funcionalidad especial a los atributos. Son un mecanismo abierto cuya finalidad es dependiente de la implementación.

1.3.3. Pragma `LANGUAGE`

Establece opciones del lenguaje. El objetivo de esta directiva es habilitar extensiones y funcionalidades experimentales. La etapa de análisis sintáctico no se ve afectada por la directiva `LANGUAGE`.

La directiva `LANGUAGE` puede potencialmente afectar las etapas de análisis estático (*linter*), compilación y el comportamiento en tiempo de ejecución.

La sintaxis de la directiva es:

- `"/*@LANGUAGE@nombre-de-la-opcion@*/"` — habilita la opción indicada.

Por el momento hay una única directiva `LANGUAGE` implementada:

- `"/*@LANGUAGE@DestructuringForeach@*/"` — habilita la posibilidad de usar patrones complejos como índice del `foreach` (en lugar de permitir solamente variables).

1.4. Producciones de la gramática

Los símbolos no terminales se describen con su nombre ⟨*enCursiva*⟩. Las producciones se escriben siguiendo las convenciones usuales de EBNF. La gramática es liberal en algunos sentidos. En particular:

- El `return` se considera un *statement* que puede aparecer en la posición en la que podría aparecer cualquier otra instrucción. La restricción de que el `return` únicamente aparezca como última instrucción del bloque, y únicamente al final de las declaraciones de funciones y del programa principal, es una restricción que se posterga para la etapa de chequeo semántico (*lint*).
- Se admite el guión bajo (`'_'`) como un posible patrón en el lado izquierdo de las ramas de un `switch` y en el lado izquierdo de las ramas de un `interactive program`. En una lista de ramas debería haber un único guión bajo, y debería ser el último patrón de la lista pero esto, de nuevo, se relega a la etapa de lint.
- Los patrones del interactive program (`TIMEOUT`, `K_ENTER`, etc.) se admiten como patrones en cualquier switch.

Las convenciones de asociatividad y precedencia de operadores no se reflejan en las producciones, sino en la tabla de precedencia. (Para expresar esto en la gramática sería necesario estratificar las expresiones en términos, factores, átomos, etc., tal como se hace en la gramática oficial).

⟨*start*⟩ ⟶ (⟨*definition*⟩)*

⟨*definition*⟩ ⟶ ⟨*defProgram*⟩
| ⟨*defInteractiveProgram*⟩
| ⟨*defProcedure*⟩
| ⟨*defFunction*⟩
| ⟨*defType*⟩

⟨*defProgram*⟩ ⟶ `PROGRAM` ⟨*stmtBlock*⟩

⟨*defInteractiveProgram*⟩ ⟶ `INTERACTIVE PROGRAM` (⟨*stmtSwitchBranch*⟩)* `RBRACE`

⟨*defProcedure*⟩ ⟶ `PROCEDURE LPAREN` ⟨*loweridSeq*⟩ `RPAREN` ⟨*stmtBlock*⟩

⟨*defFunction*⟩ ⟶ `FUNCTION LPAREN` ⟨*loweridSeq*⟩ `RPAREN` ⟨*stmtBlock*⟩

⟨*defType*⟩ ⟶ `FUNCTION LPAREN` ⟨*loweridSeq*⟩ `RPAREN` ⟨*stmtBlock*⟩
| `TYPE UPPERID IS RECORD LBRACE` (⟨*fieldDeclaration*⟩)* `RBRACE`
| `TYPE UPPERID IS VARIANT LBRACE` (⟨*constructorDeclaration*⟩)* `RBRACE`

⟨*constructorDeclaration*⟩ ⟶ CASE UPPERID LBRACE (⟨*fieldDeclaration*⟩)* RBRACE
⟨*fieldDeclaration*⟩ ⟶ FIELD LOWERID
⟨*loweridSeq*⟩ ⟶ ϵ
| ⟨*loweridSeq*$_1$⟩
⟨*loweridSeq*$_1$⟩ ⟶ LOWERID (COMMA ⟨*loweridSeq*$_1$⟩)?
⟨*statement*⟩ ⟶ ⟨*stmtEllipsis*⟩
| ⟨*stmtBlock*⟩
| ⟨*stmtReturn*⟩
| ⟨*stmtIf*⟩
| ⟨*stmtRepeat*⟩
| ⟨*stmtForeach*⟩
| ⟨*stmtWhile*⟩
| ⟨*stmtSwitch*⟩
| ⟨*stmtLet*⟩
| ⟨*stmtVariableAssignment*⟩
| ⟨*stmtProcedureCall*⟩
⟨*stmtEllipsis*⟩ ⟶ ELLIPSIS
⟨*stmtBlock*⟩ ⟶ LBRACE (⟨*statement*⟩ (SEMICOLON)?)* RBRACE
⟨*stmtReturn*⟩ ⟶ RETURN LPAREN ⟨*expressionSeq*$_1$⟩ RPAREN
⟨*stmtIf*⟩ ⟶ IF ⟨*stmtIfBranch*⟩ (ELSEIF ⟨*stmtIfBranch*⟩)* (ELSE ⟨*stmtBlock*⟩)?
⟨*stmtIfBranch*⟩ ⟶ LPAREN ⟨*expression*⟩ RPAREN (THEN)? ⟨*stmtBlock*⟩
⟨*stmtRepeat*⟩ ⟶ REPEAT LPAREN ⟨*expression*⟩ RPAREN ⟨*stmtBlock*⟩
⟨*stmtForeach*⟩ ⟶ FOREACH ⟨*pattern*⟩ IN ⟨*expression*⟩ ⟨*stmtBlock*⟩
⟨*stmtWhile*⟩ ⟶ WHILE LPAREN ⟨*expression*⟩ RPAREN ⟨*stmtBlock*⟩
⟨*stmtSwitch*⟩ ⟶ SWITCH LPAREN ⟨*expression*⟩ RPAREN (TO)? LBRACE
(⟨*stmtSwitchBranch*⟩)* RBRACE
⟨*stmtSwitchBranch*⟩ ⟶ ⟨*pattern*⟩ ARROW ⟨*stmtBlock*⟩
⟨*stmtLet*⟩ ⟶ LET ⟨*stmtVariableAssignment*⟩
| LET ⟨*stmtTupleAssignment*⟩
⟨*stmtVariableAssignment*⟩ ⟶ LOWERID ASSIGN ⟨*expression*⟩
⟨*stmtTupleAssignment*⟩ ⟶ LPAREN RPAREN ASSIGN ⟨*expression*⟩
| LPAREN LOWERID COMMA ⟨*loweridSeq*$_1$⟩ RPAREN ASSIGN ⟨*expression*⟩
⟨*stmtProcedureCall*⟩ ⟶ UPPERID LPAREN ⟨*expressionSeq*⟩ RPAREN
⟨*pattern*⟩ ⟶ ⟨*patternWildcard*⟩
| ⟨*patternVariable*⟩
| ⟨*patternNumber*⟩
| ⟨*patternStructure*⟩
| ⟨*patternTuple*⟩
| ⟨*patternTimeout*⟩
⟨*patternWildcard*⟩ ⟶ UNDERSCORE
⟨*patternVariable*⟩ ⟶ LOWERID
⟨*patternNumber*⟩ ⟶ NUM
| MINUS NUM
⟨*patternStructure*⟩ ⟶ UPPERID (LPAREN ⟨*loweridSeq*⟩ RPAREN)?

⟨*patternTuple*⟩ ⟶ LPAREN RPAREN
| LPAREN LOWERID COMMA ⟨*loweridSeq*$_1$⟩ RPAREN
⟨*patternTimeout*⟩ ⟶ TIMEOUT LPAREN NUM RPAREN
⟨*expression*⟩ ⟶ ⟨*exprAtom*⟩
| ⟨*expression*⟩ ⟨*infixOperator*⟩ ⟨*expression*⟩
| ⟨*prefixOperator*⟩ ⟨*expression*⟩
| LPAREN ⟨*expression*⟩ RPAREN
⟨*exprAtom*⟩ ⟶ ⟨*exprEllipsis*⟩
| ⟨*exprVariable*⟩
| ⟨*exprFunctionCall*⟩
| ⟨*exprConstantNumber*⟩
| ⟨*exprConstantString*⟩
| ⟨*exprChoose*⟩
| ⟨*exprMatching*⟩
| ⟨*exprList*⟩
| ⟨*exprRange*⟩
| ⟨*exprTuple*⟩
| ⟨*exprStructure*⟩
| ⟨*exprStructureUpdate*⟩
⟨*exprEllipsis*⟩ ⟶ ELLIPSIS
⟨*exprVariable*⟩ ⟶ LOWERID
⟨*exprFunctionCall*⟩ ⟶ LOWERID LPAREN ⟨*expressionSeq*⟩ RPAREN
⟨*exprConstantNumber*⟩ ⟶ NUM
⟨*exprConstantString*⟩ ⟶ STRING
⟨*exprChoose*⟩ ⟶ CHOOSE (⟨*expression*⟩ WHEN LPAREN ⟨*expression*⟩ RPAREN)*
⟨*expression*⟩ OTHERWISE
⟨*exprMatching*⟩ ⟶ MATCHING ⟨*expression*⟩ SELECT (⟨*expression*⟩ ON ⟨*pattern*⟩)*
⟨*expression*⟩ OTHERWISE
⟨*exprList*⟩ ⟶ LBRACK ⟨*expressionSeq*⟩ RBRACK
⟨*exprRange*⟩ ⟶ LBRACK ⟨*expression*⟩ RANGE ⟨*expression*⟩ RBRACK
| LBRACK ⟨*expression*⟩ COMMA ⟨*expression*⟩ RANGE ⟨*expression*⟩
RBRACK
⟨*exprTuple*⟩ ⟶ LPAREN RPAREN
| LPAREN ⟨*expression*⟩ COMMA ⟨*expressionSeq*$_1$⟩ RPAREN
⟨*exprStructure*⟩ ⟶ UPPERID (LPAREN ⟨*fieldBindingSeq*⟩ RPAREN)?
⟨*exprStructureUpdate*⟩ ⟶ UPPERID LPAREN ⟨*expression*⟩ PIPE ⟨*fieldBindingSeq*⟩ RPAREN
⟨*fieldBinding*⟩ ⟶ LOWERID GETS ⟨*expression*⟩
⟨*fieldBindingSeq*⟩ ⟶ ϵ
| ⟨*fieldBindingSeq*$_1$⟩
⟨*fieldBindingSeq*$_1$⟩ ⟶ ⟨*fieldBinding*⟩ (COMMA ⟨*fieldBindingSeq*$_1$⟩)?

⟨*infixOperator*⟩ ⟶ OR
| AND
| EQ
| NE
| LE
| GE
| LT
| GT
| CONCAT
| PLUS
| MINUS
| TIMES
| DIV
| MOD
| POW

⟨*prefixOperator*⟩ ⟶ MINUS
| NOT

⟨*expressionSeq*⟩ ⟶ ϵ
| ⟨*expressionSeq*$_1$⟩

⟨*expressionSeq*$_1$⟩ ⟶ ⟨*expression*⟩ (COMMA ⟨*expressionSeq*$_1$⟩)?

1.5. Precedencia de operadores

Los operadores se organizan según la siguiente tabla. Cada fila corresponde a un nivel de precedencia, ordenados de menor a mayor precedencia. Las *fixities* posibles son: operador binario asociativo a derecha (**InfixR**), operador binario asociativo a izquierda (**InfixL**), operador binario no asociativo (**Infix**), operador unario prefijo (**Prefix**).

- **InfixR:** OR (`||`)
- **InfixR:** AND (`&&`)
- **Prefix:** NOT (`not`)
- **Infix:** EQ (`==`) NE (`/=`) LE (`<=`) GE (`>=`) LT (`<`) GT (`>`)
- **InfixL:** CONCAT (`++`)
- **InfixL:** PLUS (`+`) MINUS (`-`)
- **InfixL:** TIMES (`*`)
- **InfixL:** DIV (`div`) MOD (`mod`)
- **InfixR:** POW (`^`)
- **Prefix:** MINUS (`-`, menos unario)

1.6. Árbol de sintaxis abstracta

El resultado de analizar sintácticamente un programa es un objeto. El objeto representa un árbol sintáctico y está construido recursivamente de la siguiente manera:

- Las hojas del árbol son instancias de la clase `Token` cuyo atributo `tag` representa el tipo de token (por ejemplo, `T_UPPERID`) y cuyo atributo `value` representa el valor leído (por ejemplo, "`PonerN`").
- Los nodos internos del árbol son instancias de la clase `ASTNode` o, más precisamente, de alguna de sus subclases. Un nodo interno tiene dos atributos:
  - `tag`: indica el "tipo" de nodo del que se trata. Por ejemplo una instancia de la clase `ASTStmtIf` representa un comando `if-then-else`, y el tag asociado es el símbolo `N_StmtIf`.
  - `children`: es la lista de hijos del nodo. Por ejemplo, las instancias de la clase `ASTStmtIf` tienen exactamente tres hijos: el primero es una expresión que representa la condición del if, el segundo es un comando que representa la rama `then`, y el tercero puede ser `null` (si la rama `else` se encuentra ausente), o un comando que representa la rama `else`.
- Como se mencionó arriba, en algunos (pocos) casos las hojas del árbol también pueden ser `null`, que se usa para indicar la ausencia de algunas componentes opcionales del árbol (tales como la rama `else` de un `if`).
- Los hijos eventualmente también pueden ser listas de ASTs.

Abajo se especifica la forma que tiene un árbol usando una sintaxis similar a la de la declaración de un tipo de datos inductivo en Haskell. Los tipos escritos `EN_MAYÚSCULAS` representan el tipo de los tokens. Los tipos escritos `EnMinúsculas` representan categorías abstractas de ASTs (por ejemplo, `Statement` es la categoría abstracta de aquellos AST que representan comandos). Las palabras escritas `EnVerde` representan categorías concretas (en la terminología de Haskell, constructores; en el caso de JavaScript corresponden a subclases de `ASTNode`). Por ejemplo, `StmtIf` es la categoría concreta de los AST que representan un comando if-then-else. En JavaScript, su clase asociada se llama `ASTStmtIf`, y su tag asociado es el símbolo `N_StmtIf`.

```
ID = String
NUM = Integer
STRING = String
Main ::= Main(definitions : [Definition])
Definition ::= DefProgram(body : Statement)
             | DefInteractiveProgram(branches : [SwitchBranch])
             | DefProcedure(name : ID, parameters : [ID], body : Statement)
             | DefFunction(name : ID, parameters : [ID], body : Statement)
             | DefType(typeName : ID, constructorDeclarations : [ConstructorDeclaration])
```

```
Statement ::= StmtBlock(statements : [Statement])
            | StmtReturn(result : Expression)
            | StmtIf(condition : Expression, thenBlock : Statement, elseBlock : (Statement)?)
            | StmtRepeat(times : Expression, body : Statement)
            | StmtForeach(pattern : Pattern, range : Expression, body : Statement)
            | StmtWhile(condition : Expression, body : Statement)
            | StmtSwitch(subject : Expression, branches : [SwitchBranch])
            | StmtAssignVariable(variable : ID, value : Expression)
            | StmtAssignTuple(variables : [ID], value : Expression)
            | StmtProcedureCall(procedureName : ID, args : [Expression])
SwitchBranch ::= SwitchBranch(pattern : Pattern, body : Statement)
Pattern ::= PatternWildcard()
          | PatternVariable(variableName : ID)
          | PatternNumber(number : NUM)
          | PatternStructure(constructorName : ID, parameters : [ID])
          | PatternTuple(parameters : [ID])
          | PatternTimeout(timeout : NUM)
Expression ::= ExprVariable(variableName : ID)
             | ExprConstantNumber(number : NUM)
             | ExprConstantString(string : STRING)
             | ExprChoose(condition : Expression, trueExpr : Expression, falseExpr :
               Expression)
             | ExprMatching(subject : Expression, branches : [MatchingBranch])
             | ExprList(elements : [Expression])
             | ExprRange(first : Expression, second : (Expression)?, last : Expression)
             | ExprTuple(elements : [Expression])
             | ExprStructure(constructorName : ID, fieldBindings : [FieldBinding])
             | ExprStructureUpdate(constructorName : ID, original : Expression, fieldBindings :
               [FieldBinding])
             | ExprFunctionCall(functionName : ID, args : [Expression])
MatchingBranch ::= MatchingBranch(pattern : Pattern, body : Expression)
ConstructorDeclaration ::= ConstructorDeclaration(constructorName : ID, fieldNames : [ID])
FieldBinding ::= FieldBinding(fieldName : ID, value : Expression)
```

2. Análisis semántico

La etapa de análisis semántico (o *linting*) verifica de manera estática que el programa cumpla las siguientes condiciones.

Programa principal.

- El programa **PUEDE** estar completamente vacío (sin ninguna definición).
- Si el programa no está completamente vacío, **DEBE** tener exactamente una definición de programa (ya sea un `DefProgram` o `DefInteractiveProgram`).

- El programa **NO PUEDE** tener dos (o más) definiciones de programa (ya sean `DefProgram` o `DefInteractiveProgram`).

Nombres globales.

- Hay cinco tipos de nombres globales:
  - Nombres de procedimientos, declarados con `procedure`.
  - Nombres de funciones, declaradas con `function`.
  - Nombres de tipos, declarados con `type`.
  - Nombres de constructores. Coinciden con el nombre del tipo en la declaración de un tipo `record`. Pueden coincidir o no con el nombre del tipo en la declaración de un tipo `variant`, en cuyo caso se declaran con `case`.
  - Nombres de campos, declarados con `field`.
- El programa **NO PUEDE** tener dos (o más) funciones con el mismo nombre.
- El programa **NO PUEDE** tener dos (o más) procedimientos con el mismo nombre.
- El programa **NO PUEDE** tener dos (o más) tipos con el mismo nombre.
- El programa **NO PUEDE** tener dos (o más) constructores con el mismo nombre.
- El programa **NO PUEDE** tener dos (o más) campos con el mismo nombre correspondientes a un mismo constructor. (P.ej. la definición: `type A is record { field x field x }` se rechaza).
- El programa **PUEDE** tener dos (o más) campos con el mismo nombre si corresponden a distintos constructores, ya sea del mismo tipo o de distintos tipos. (P.ej. las definiciones: `type A is record { field x } type B is record { field x }` se aceptan).
- El programa **NO PUEDE** tener una función y un campo que tengan el mismo nombre.
- El programa **PUEDE** tener tipos y constructores con el mismo nombre. P.ej. la definición: `type A is variant { case A }` se acepta. De hecho este es el comportamiento normal cuando se declara un registro: `type A is record { }` define simultáneamente un tipo `A` y un constructor `A`.
- El programa **PUEDE** tener procedimientos con el mismo nombre que un tipo o que un constructor.

Return.

- Los procedimientos **NO PUEDEN** tener un `return`.
- Los programas declarados con `interactive program` **NO PUEDEN** tener un `return`.
- Las funciones **DEBEN** tener un `return` como último comando del bloque.
- Los programas declarados con `program` **PUEDEN** tener un `return` como último comando del bloque.

- **NO PUEDE** haber un `return` en ninguna otra posición salvo las mencionadas, es decir, como el último comando de una función o como el último comando de un `program`.

Nombres locales.

- Hay tres tipos de nombres locales:
  - Parámetros — están ligados en la lista de parámetros de un procedimiento, o en la lista de parámetros de una función, o en la lista de variables ligadas al hacer *pattern matching* con `switch`. Por ejemplo `x` e `y` son parámetros en los comandos:
    1. `switch (c) { Coord(x, y) -> ... }`
    2. `switch (c) { (x, y) -> ... }`
    3. `switch (c) { x -> ... }`
  - Índices — están ligados por un `foreach`.
  - Variables — su valor se declara en una asignación `x := e` o en una asignación a una tupla `(x1, ..., xN) := e`.
- Alcance de los nombres locales:
  - Parámetros de procedimientos y funciones — locales a todo el cuerpo del procedimiento o función.
  - Variables ligadas en los patrones de un `switch` — locales al bloque a la derecha de la correspondiente flecha `->`.
  - Índices — locales al cuerpo del `foreach`.
  - Variables — locales a todo el cuerpo del procedimiento o función.
- Los nombres locales **PUEDEN** coincidir con los nombres globales (nombres de funciones y campos), sin que unos opaquen a otros.
- Los nombres locales **NO PUEDEN** coincidir con otros nombres locales que comparten el mismo alcance. En particular:
  - Las funciones y procedimientos **NO PUEDEN** tener nombres de parámetros repetidos.
  - Los índices de dos `foreach` anidados **NO PUEDEN** tener el mismo nombre.
  - Los índices de dos `foreach` que no están anidados **PUEDEN** tener el mismo nombre.
  - Los nombres de variables, índices, y parámetros **NO PUEDEN** coincidir.
- En una asignación de tuplas `let (x1, ..., xn) := e` las variables `x1, ..., xn` **DEBEN** ser todas distintas.

Pattern matching.

- Hay seis tipos de patrones:
  1. Patrón comodín (_).
  2. Patrón variable (identificador ligado localmente).
  3. Patrón numérico (un número n, que puede ser positivo, negativo o cero).

4. Patrón estructura (`C` o alternativamente `C(`x_1`, ..., `x_n`)`).
5. Patrón tupla (`(`x_1`, ..., `x_n`)`).
6. Patrón timeout (`TIMEOUT(`n`)`).

- Actualmente los patrones se pueden utilizar en cuatro lugares:
  1. en las ramas de un `interactive program`,
  2. en las ramas de un `switch`,
  3. en las ramas de un `matching..select`,
  4. como patrón de un `foreach`, siempre y cuando la opción `DestructuringForeach` haya sido habilitada con la directiva `/*@LANGUAGE@DestructuringForeach@*/`. De lo contrario, el patrón de un `foreach` puede ser únicamente un patrón variable (índice).
- En un patrón estructura, el nombre del constructor en cuestión **DEBE** ser un constructor existente de algún tipo.
- En un patrón estructura, el constructor **PUEDE** tener 0 parámetros, independientemente del número de campos que tenga el constructor correspondiente.
- Si un patrón estructura tiene 1 o más parámetros, el número **DEBE** coincidir con el número de campos del constructor correspondiente.
- En una secuencia de ramas los patrones estructura **PUEDEN** aparecer en cualquier orden, sin respetar necesariamente el orden en que están declarados.
- Una secuencia de ramas **NO PUEDE** tener dos patrones numéricos con el mismo valor numérico.
- Una secuencia de ramas **NO PUEDE** tener dos patrones estructura asociados al mismo constructor.
- Una secuencia de ramas **NO PUEDE** tener dos patrones tupla con la misma longitud.
- Una secuencia de ramas **NO PUEDE** tener dos patrones timeout.
- Una secuencia de ramas **PUEDE** tener o no tener un patrón comodín. (No es obligatorio como última rama).
- Si una secuencia de ramas tiene un patrón comodín o un patrón variable, dicho patrón **DEBE** estar en la última rama.
- Una secuencia de ramas **NO PUEDE** tener un patrón comodín o un patrón variable en ninguna otra rama que no sea la última.
- Un patrón comodín **PUEDE** ser inalcanzable. Es decir, un patrón comodín puede estar presente incluso si la secuencia de ramas cubre todas las alternativas de constructores posibles.
- En una secuencia de ramas todos los patrones **DEBEN** ser compatibles. Son patrones compatibles:

- Dos patrones numéricos.
- Dos patrones estructura cuyos constructores pertenecen al mismo tipo.
- Un patrón timeout con cualquier otro constructor del tipo `_EVENT` (que corresponde a los eventos que pueden darse en un programa interactivo).
- Un patrón comodín con cualquier otro patrón.
- Un patrón variable con cualquier otro patrón.

Dos tipos son incompatibles en cualquier otro caso. En particular, son patrones incompatibles:

- Un patrón numérico y un patrón estructura.
- Un patrón numérico y un patrón tupla.
- Dos patrones estructura cuyos constructores pertenecen a distintos tipos.
- Dos patrones tupla con distinto número de componentes.
- Un patrón estructura y un patrón tupla.

- Los patrones de las ramas del `interactive program` **DEBEN** ser eventos y **NO PUEDEN** ser patrones variable, es decir, pueden ser patrones timeout, patrones estructura del tipo `_EVENT` o patrones comodín.
- Los patrones de las ramas de los `switch` y los `matching..select` **NO PUEDEN** ser eventos, es decir: pueden ser patrones numéricos, patrones estructura de tipos que no sean `_EVENT`, patrones tupla, patrones variable, o patrones comodín.
- El patrón de un `foreach` **NO PUEDE** ser un evento.

Expresiones.

- Los usos de parámetros, índices y variables (`ExprVariable`) **PUEDEN** no corresponder a parámetros, índices o variables definidas. La restricción de que las variables se pueden usar solamente después de que se les haya dado un valor es una restricción dinámica.
- Cuando se construye una instancia de un tipo (registro o variante) con un constructor (`ExprStructure`), el nombre del constructor **DEBE** ser el nombre de un constructor existente.
- Cuando se construye o se actualiza una instancia con un constructor (`ExprStructure`, `ExprStructureUpdate`), **NO PUEDE** haber nombres de campos repetidos en la lista de *bindings*.
- Cuando se construye o se actualiza una instancia con un constructor (`ExprStructure`, `ExprStructureUpdate`), los nombres de los campos que aparecen en la lista de *bindings* **DEBEN** ser nombres de campos válidos para dicho constructor. (Correctitud).
- Cuando se construye una instancia con un constructor (`ExprStructure`), los nombres de los campos que aparecen en la lista de *bindings* **DEBEN** cubrir todos los posibles nombres de campos válidos para dicho constructor. (Completitud).
- Cuando se construye o se actualiza una instancia con un constructor (`ExprStructure`, `ExprStructureUpdate`), el constructor **NO PUEDE** ser un constructor del tipo `_EVENT` (esto técnicamente depende del entorno global en el que se evalúe el programa, pero típicamente los eventos son las teclas como `K_ENTER`, etc.).

Invocaciones a funciones y procedimientos.

- En las invocaciones a procedimientos, el nombre del procedimiento **DEBE** corresponder a un procedimiento definido.
- En las invocaciones a funciones, el nombre de la función **DEBE** corresponder o bien a una función definida o bien al nombre de un campo (usado como función observadora).
- El chequeo de coincidencia del número de argumentos (aridades) se hace estáticamente. Todos los chequeos de tipos son dinámicos.
- El número de argumentos pasados a un procedimiento **DEBE** coincidir con el número de parámetros declarados en su definición.
- El número de argumentos pasados a una función **DEBE** coincidir con el número de parámetros declarados en su definición.
- El número de argumentos pasados a un campo (usado como función observadora) **DEBE** ser exactamente 1.

3. Máquina virtual

Un programa en Gobstones/XGobstones se compila a una serie de instrucciones para una máquina virtual *ad hoc*. La implementación de un compilador y una máquina virtual, en lugar de un mero intérprete recursivo, agrega bastante complejidad a la base de código, y es razonable preguntarse hasta qué punto hacer un compilador no es *overkill*.

La principal razón para contar con un compilador en este caso no tiene que ver con cuestiones de eficiencia, sino con la posibilidad de tener representado explícitamente el estado de ejecución de un programa. Esto permite:

1. Hacer un *step* controlado del programa, para poder visualizar el avance paso a paso de la ejecución.
2. Potencialmente pausar y continuar la ejecución de un programa.
3. Ejecutar programas que potencialmente se cuelgan abortándolos por medio de diferentes criterios de *timeout*.

Esta funcionalidad no se puede implementar de manera directa en un intérprete recursivo (las posibles implementaciones, por ejemplo usando continuaciones, son difíciles de entender y mantener).

El código para la máquina virtual es una instancia de la clase `Code`, que básicamente consta de una lista de instrucciones. Las instrucciones están basadas en una arquitectura sencilla "orientada a pila".

El estado de ejecución de un programa consta de los siguientes datos:

- Un código (`Code`) que, como dijimos, es básicamente una lista de instrucciones.
- Una pila no vacía de estados globales. El estado global actual es el estado en el tope de la pila. En Gobstones/XGobstones el estado global corresponde al tablero, pero podría corresponder a otras entidades mutables en otras variantes concebibles del lenguaje. Cada vez que se ingresa

a una función definida por el usuario, el estado global actual se copia y se mete en la pila de estados globales (con la instrucción `SaveState`). Cada vez que finaliza una función definida por el usuario, se saca el estado global actual del tope de la pila de estados globales (con la instrucción `RestoreState`). Esto permite implementar la semántica de Gobstones, de acuerdo con la cual las funciones no tienen efectos (todos sus efectos se deshacen).

- Una "pila de llamadas" (`callStack`). La pila de llamadas es una pila no vacía de *frames*. Un *frame* es el estado local o contexto de ejecución de la rutina actual (también conocido como *stack frame* o *registro de activación* en la literatura). Un *frame* es una instancia de la clase `Frame` y consta de los siguientes datos:
  - El *instruction pointer*, que es un índice dentro de la lista de instrucciones.
  - Una pila de valores locales.
  - Un diccionario de nombres locales, que asocia los nombres de parámetros, índices y variables a un valor.

Cuando decimos "el *instruction pointer*" nos referimos al *instruction pointer* del *frame* que está en el tope de la pila de llamadas. De igual modo, cuando decimos "la pila" nos referimos a la pila del *frame* que está en el tope de la pila de llamadas. Ídem para el diccionario de nombres locales.

3.1. Instrucciones

Las instrucciones de la máquina virtual se describen a continuación:

- `PushInteger`(number : `INT`)
  Mete la constante numérica dada en la pila.
- `PushString`(string : `STRING`)
  Mete la constante de cadena dada en la pila.
- `PushVariable`(variableName : `ID`)
  Mete el valor actual de la variable local variableName en la pila. Si variableName no tiene asociado un valor, lanza una excepción GbsRuntimeError.
- `SetVariable`(variableName : `ID`)
  Saca el valor del tope de la pila y modifica la variable local variableName para que tome dicho valor. Si la variable no existe en el diccionario de nombres locales, se crea una entrada. Si la variable ya existía, se verifica que el tipo del valor que tenía y el tipo del valor actual sean compatibles.
- `UnsetVariable`(variableName : `ID`)
  Elimina la variable local variableName del diccionario de nombres locales. Si la variable no existe en el diccionario de nombres locales, esta instrucción no causa ningún otro efecto más que incrementar el instruction pointer.
- `Label`(label : `LABEL`)
  Pseudo-instrucción para definir una etiqueta, es decir un punto en el código que puede ser el destino de un salto o una invocación. No puede haber etiquetas repetidas en todo el código.

- `Jump`(targetLabel : `LABEL`)
  Salta incondicionalmente a la etiqueta indicada (debe existir).

- `JumpIfFalse`(targetLabel : `LABEL`)
  Si el tope de la pila es una estructura cuyo constructor es False, salta a la etiqueta indicada, que debe existir. Esta instrucción saca el elemento del tope de la pila. La pila no puede estar vacía.

- `JumpIfStructure`(constructorName : `ID`, targetLabel : `LABEL`)
  Si el tope de la pila es una estructura cuyo constructor es el indicado por 'constructorName', salta a la etiqueta indicada, que debe existir. No saca el elemento del tope de la pila. La pila no puede estar vacía.

- `JumpIfTuple`(size : `INT`, targetLabel : `LABEL`)
  Si el tope de la pila es una tupla con la cantidad de componentes indicada por 'size', salta a la etiqueta indicada, que debe existir. No saca el elemento del tope de la pila. La pila no puede estar vacía.

- `Call`(targetLabel : `LABEL`, nargs : `INT`)
  Hace una invocación a una subrutina. Más precisamente, mete un nuevo stack frame en la pila de llamadas, con el instruction pointer apuntando a la posición del código designada por la etiqueta 'targetLabel'. Saca 'nargs' valores del tope de la pila del stack frame llamador y los apila en la pila del nuevo stack frame. Notar que al desapilarlos y reapilarlos el orden se invierte, de tal manera que el primer parámetro queda en el tope de la pila.

- `Return`()
  Retorna de una invocación a una subrutina. Más precisamente, saca el stack frame del tope de la pila de llamadas. Si la pila de llamadas queda vacía, es el return del programa principal y el programa finaliza. Cuando se ejecuta la instrucción Return, debe haber 0 o 1 valores en la pila. En caso de que haya un valor, se saca de la pila del stack frame actual y se apila en el stack frame del llamador.

- `MakeTuple`(size : `INT`)
  Crea una tupla del tamaño indicado. Los elementos se sacan de la pila (el último elemento de la tupla debe encontrarse en el tope de la pila).

- `MakeList`(size : `INT`)
  Crea una lista del tamaño indicado. Los elementos se sacan de la pila (el último elemento de la lista debe encontrarse en el tope de la pila). Los elementos de la lista deben tener tipos compatibles; de lo contrario se lanza una excepción GbsRuntimeError.

- `MakeStructure`(typeName : `ID`, constructorName : `ID`, fieldNames : [`ID`])
  Crea una estructura usando el constructor indicado ('constructorName') del tipo indicado ('typeName'), con los campos indicados por la lista de nombres 'fieldNames'. Los valores de cada campo se sacan de la pila (el valor del último campo de la lista debe encontrarse en el tope de la pila).

- `UpdateStructure`(typeName : `ID`, constructorName : `ID`, fieldNames : [`ID`])
  Actualiza una estructura usando el constructor indicado ('constructorName') del tipo indicado

('typeName'), con los campos indicados por la lista de nombres 'fieldNames'. Los valores de cada campo se sacan de la pila (el valor del último campo de la lista debe encontrarse en el tope de la pila). A continuación se saca de la pila el valor, que debe ser una estructura para actualizar. Si el valor no es una estructura, o si es una estructura pero no está construida con el constructor esperado, se lanza una excepción GbsRuntimeError. Los tipos de los valores contenidos en los campos originales deben ser compatibles con los tipos de los nuevos valores de los respectivos campos; de lo contrario se lanza una excepción GbsRuntimeError.

- `ReadTupleComponent`(index : `INT`)
  Mete en la pila la 'index'-ésima componente de la tupla que se encuentra actualmente en el tope de la pila. Los índices empiezan desde 0. Debe haber un valor en el tope de la pila. No se saca dicho valor de la pila (queda como segundo elemento). Si el valor en el tope de la pila no es una tupla, se lanza una excepción GbsRuntimeError. Si el índice está fuera de rango, se lanza una excepción GbsRuntimeError.

- `ReadStructureField`(fieldName : `ID`)
  Mete en la pila el valor del campo 'fieldName' de la estructura que se encuentra actualmente en el tope de la pila. Debe haber un valor en el tope de la pila. No se saca dicho valor de la pila (queda como segundo elemento). Si el valor en el tope de la pila no es una estructura, se lanza una excepción GbsRuntimeError. Si el campo 'fieldName' no es uno de los campos presentes en la estructura, se lanza una excepción GbsRuntimeError.

- `ReadStructureFieldPop`(fieldName : `ID`)
  Mete en la pila el valor del campo 'fieldName' de la estructura que se encuentra actualmente en el tope de la pila. Debe haber un valor en el tope de la pila. El valor original se saca de la pila. Si el valor en el tope de la pila no es una estructura, se lanza una excepción GbsRuntimeError. Si el campo 'fieldName' no es uno de los campos presentes en la estructura, se lanza una excepción GbsRuntimeError.

- `PrimitiveCall`(primitiveName : `ID`, nargs : `INT`)
  Invoca a una operación primitiva. Desapila 'nargs' argumentos de la pila (con el último argumento en el tope). A continuación, invoca a la primitiva pasándole como parámetros el estado global, seguido de los 'nargs' argumentos desapilados. La función primitiva puede devolver un valor o 'null'. En caso de que devuelva un valor, se apila dicho resultado en la pila. Si devuelve null, prosigue con la ejecución.

- `SaveState`()
  Crea una copia el estado global actual y lo mete en la pila de estados globales.

- `RestoreState`()
  Saca un estado de la pila de estados globales.

- `TypeCheck`(type : `TYPE`)
  Verifica que el tipo del valor del tope de la pila sea compatible con el tipo indicado. La pila no debe estar vacía. Esta operación no saca el elemento del tope de la pila.

3.2. Primitivas

Los tipos, procedimientos y funciones disponibles en el lenguaje de manera primitiva se listan a continuación:

- `type Bool` — constructores:
  - `True`
  - `False`
- `type Color` — constructores:
  - `Azul`
  - `Negro`
  - `Rojo`
  - `Verde`
- `type Dir` — constructores:
  - `Norte`
  - `Este`
  - `Sur`
  - `Oeste`
- `type Event` — constructores:
  - `INIT`
  - `TIMEOUT`
  - `K_[CTRL_][ALT_][SHIFT_]A`
  - `...`
  - `K_[CTRL_][ALT_][SHIFT_]Z`
  - `K_[CTRL_][ALT_][SHIFT_]0`
  - `...`
  - `K_[CTRL_][ALT_][SHIFT_]9`
  - `K_[CTRL_][ALT_][SHIFT_]SPACE`
  - `K_[CTRL_][ALT_][SHIFT_]RETURN`
  - `K_[CTRL_][ALT_][SHIFT_]TAB`
  - `K_[CTRL_][ALT_][SHIFT_]BACKSPACE`
  - `K_[CTRL_][ALT_][SHIFT_]ESCAPE`
  - `K_[CTRL_][ALT_][SHIFT_]INSERT`
  - `K_[CTRL_][ALT_][SHIFT_]DELETE`
  - `K_[CTRL_][ALT_][SHIFT_]HOME`
  - `K_[CTRL_][ALT_][SHIFT_]END`
  - `K_[CTRL_][ALT_][SHIFT_]PAGEUP`
  - `K_[CTRL_][ALT_][SHIFT_]PAGEDOWN`
  - `K_[CTRL_][ALT_][SHIFT_]F1`

- ...
- K_[CTRL_][ALT_][SHIFT_]F12
- K_[CTRL_][ALT_][SHIFT_]AMPERSAND
- K_[CTRL_][ALT_][SHIFT_]ASTERISK
- K_[CTRL_][ALT_][SHIFT_]AT
- K_[CTRL_][ALT_][SHIFT_]BACKSLASH
- K_[CTRL_][ALT_][SHIFT_]CARET
- K_[CTRL_][ALT_][SHIFT_]COLON
- K_[CTRL_][ALT_][SHIFT_]DOLLAR
- K_[CTRL_][ALT_][SHIFT_]EQUALS
- K_[CTRL_][ALT_][SHIFT_]EXCLAIM
- K_[CTRL_][ALT_][SHIFT_]GREATER
- K_[CTRL_][ALT_][SHIFT_]HASH
- K_[CTRL_][ALT_][SHIFT_]LESS
- K_[CTRL_][ALT_][SHIFT_]PERCENT
- K_[CTRL_][ALT_][SHIFT_]PLUS
- K_[CTRL_][ALT_][SHIFT_]SEMICOLON
- K_[CTRL_][ALT_][SHIFT_]SLASH
- K_[CTRL_][ALT_][SHIFT_]QUESTION
- K_[CTRL_][ALT_][SHIFT_]QUOTE
- K_[CTRL_][ALT_][SHIFT_]QUOTEDBL
- K_[CTRL_][ALT_][SHIFT_]LEFTPAREN
- K_[CTRL_][ALT_][SHIFT_]RIGHTPAREN
- K_[CTRL_][ALT_][SHIFT_]LEFTBRACKET
- K_[CTRL_][ALT_][SHIFT_]RIGHTBRACKET
- K_[CTRL_][ALT_][SHIFT_]LEFTBRACE
- K_[CTRL_][ALT_][SHIFT_]RIGHTBRACE
- K_[CTRL_][ALT_][SHIFT_]LEFT
- K_[CTRL_][ALT_][SHIFT_]RIGHT
- K_[CTRL_][ALT_][SHIFT_]UP
- K_[CTRL_][ALT_][SHIFT_]DOWN

- `procedure` Mover(dir)
  - **Precondición:** El cabezal debe poder moverse hacia la dirección indicada.
  - **Propósito:** Mueve el cabezal hacia la dirección indicada.
- `procedure` Poner(color)

- **Propósito:** Pone una bolita del color indicado en la posición del tablero donde se encuentra el cabezal.

- `procedure` `Sacar`(color)
  - **Precondición:** Debe haber una bolita del color indicado en la posición del tablero donde se encuentra el cabezal.
  - **Propósito:** Saca una bolita del color indicado en la posición del tablero donde se encuentra el cabezal.
- `procedure` `IrAlBorde`(dir)
  - **Propósito:** Mueve el cabezal hacia el extremo de la dirección indicada.
- `procedure` `VaciarTablero`()
  - **Propósito:** Vacía el contenido de todas las celdas del tablero.
- `function` `puedeMover`(dir)
  - **Propósito:** Denota verdadero si el cabezal puede moverse hacia la dirección indicada.
- `function` `nroBolitas`(color)
  - **Propósito:** Denota el número de bolitas del color indicado en la posición donde se encuentra el cabezal.
- `function` `hayBolitas`(color)
  - **Propósito:** Denota verdadero si hay bolitas del color indicado en la posición donde se encuentra el cabezal.
- `function` `siguiente`(x)
  - **Precondición:** El parámetro x debe ser un entero, booleano, color o dirección.
  - **Propósito:** Denota el siguiente de x en el orden correspondiente al tipo de x.
- `function` `previo`(x)
  - **Precondición:** El parámetro x debe ser un entero, booleano, color o dirección.
  - **Propósito:** Denota el anterior de x en el orden correspondiente al tipo de x.
- `function` `opuesto`(x)
  - **Precondición:** El parámetro x debe ser un entero, booleano o dirección.
  - **Propósito:** Denota el opuesto de x en el orden correspondiente al tipo de x.
- `function` `minBool`()
  - **Propósito:** Denota el mínimo valor booleano.
- `function` `maxBool`()

- **Propósito:** Denota el máximo valor booleano.

- `function` `minColor`()
  - **Propósito:** Denota el mínimo color.
- `function` `maxBool`()
  - **Propósito:** Denota el máximo color.
- `function` `minDir`()
  - **Propósito:** Denota la mínima dirección.
- `function` `maxDir`()
  - **Propósito:** Denota la máxima dirección.
- `function` `esVacía`(lista)
  - **Propósito:** Denota verdadero si la lista es vacía.
- `function` `primero`(lista)
  - **Precondición:** La lista no puede ser vacía.
  - **Propósito:** Denota la cabeza de la lista, es decir, su primer elemento.
- `function` `sinElPrimero`(lista)
  - **Precondición:** La lista no puede ser vacía.
  - **Propósito:** Denota la cola de la lista, es decir, la lista que consta de todos los elementos excepto el primero.
- `function` `comienzo`(lista)
  - **Precondición:** La lista no puede ser vacía.
  - **Propósito:** Denota el comienzo de la lista, es decir, la lista que consta de todos los elementos excepto el último.
- `function` `último`(lista)
  - **Precondición:** La lista no puede ser vacía.
  - **Propósito:** Denota el último elemento de la lista.

3.3. Snapshots

Las rutinas (`program`, procedimientos y funciones), ya sean primitivas o definidas por el usuario, tienen asociados dos atributos booleanos:

- `recorded`: indica si la rutina debe ser "grabada", es decir, si se debe tomar una *snapshot* en el momento en que se retorna de dicha rutina.
- `atomic`: indica si la rutina debe ser considerada "atómica", es decir, si **no** debe grabarse ninguna *snapshot* que ocurra de manera interna (es decir, *snapshots* correspondientes a rutinas internas).

Se aplican las siguientes reglas:

- El `program` nunca `atomic`. Además el `program` **no** es `recorded`: se toma una *snapshot* al comienzo del `program`, pero no al final. La última *snapshot* será la ocasionada por el último comando que tenga el atributo `recorded`.
- Las funciones nunca son `recorded` y siempre son `atomic`. (Los efectos que se producen durante la ejecución de una función son invisibles desde el exterior, de manera que cualquier cambio en el estado que se produzca durante la ejecución de una función no debe grabarse).
- Los procedimientos primitivos son `recorded`. (Es irrelevante si son `atomic`, ya que durante la ejecución de un procedimiento primitivo nunca se invoca a otro procedimiento).
- Los procedimientos definidos por el usuario, por defecto, no son `recorded` ni `atomic`. (Al ejecutar un procedimiento definido por el usuario, no se graba de manera especial el estado que se obtiene al finalizar el procedimiento, pero sí se graban las *snapshots* de todo lo que ocurre en su interior, lo que típicamente incluye el estado que se obtiene al final de su ejecución).

En el futuro se pretende que los atributos `recorded` y `atomic` de las rutinas definidas por el usuario puedan controlarse a través de pragmas.