PLg Grupo 3 (2012-2013)

Miembros del grupo

- Marina Bezares Álvarez
- Daniel Escoz Solana
- Antonio Irízar López
- Raúl Marcos Lorenzo
- Pedro Morgado Alarcón
- Arturo Pareja García

1. Definición léxica

Formación de literales e identificadores

```
litnat ≡ _dign0 (_dig)* | "0"
litfloat ≡ litnat _partedec (_parteexp)? | litnat _parteexp
litchar ≡ "'"_alfanum1"'"
ident ≡ _min (_alfanum1)*
true ≡ "true"
false ≡ "false"
```

Palabras reservadas

```
program ≡ "program:"
subprograms ≡ "subprograms"
subprogram ≡ "subprogram:"
varconsts ≡ "vars-consts"
instructions ≡ "instructions"
var ≡ "var"
const ≡ "const"
float ≡ "float"
integer ≡ "integer"
int ≡ "int"
boolean ≡ "boolean"
natural ≡ "natural"
nat ≡ "nat"
character ≡ "character"
char ≡ "char"
in ≡ "in"
out ≡ "out"
swap1 ≡ "swap1"
swap2 ≡ "swap2"
call ≡ "call"
```

Símbolos y operadores

```
asig ≡ "="
lpar ≡ "("
rpar ≡ ")"
illave ≡ "{"
fllave ≡ "}"
pyc ≡ ";"
men ≡ "<"
```

```
menoig ≡ "<="
may ≡ ">"
mayoig ≡ ">="
igual ≡ "=="
noigual ≡ "!="
mas ≡ "+"
menos ≡ "-"
mul ≡ "*"
div ≡ "/"
mod ≡ "%"
and ≡ "and"
or ≡ "or"
not ≡ "not"
lsh ≡ "<<"
rsh ≡ ">>"
coma ≡ ","
barrabaja ≡ "_"
```

Expresiones auxiliares

```
_min ≡ ['a'-'z']
_may ≡ ['A'-'Z']
_letra ≡ _min | _may
_dig ≡ ['0'-'9']
_dign0 ≡ ['1'-'9']
_alfanum1 ≡ _letra | _dig
_partedec ≡ "." ((_dig)*_dign0 | "0")
_parteexp ≡ ("e" | "E") "-"? litnat
fin ≡ <end-of-file>
```

2. Definición sintáctica del lenguaje

2.1 Descripción de los operadores

Operador	Prioridad	Aridad	Asociatividad
Igualdad (==)	0	2	Ninguna
Desigualdadd (!=)	0	2	Ninguna
Menor que (<)	0	2	Ninguna
Menor o igual (<=)	0	2	Ninguna
Mayor que (>)	0	2	Ninguna
Mayor o igual (>=)	0	2	Ninguna
Suma (+)	1	2	Izquierdas
Resta (-)	1	2	Izquierdas
Disyunción lógica (or)	1	2	Izquierdas
Multiplicación (*)	2	2	Izquierdas
División (/)	2	2	Izquierdas
Módulo (%)	2	2	Izquierdas
Conjunción (%)	2	2	Izquierdas
Despl. Izquierda (<<)	3	2	Derechas

Despl. Derecha (>>)	3	2	Derechas
Negación aritmética (-)	4	1	Sí
Negación lógica (not)	4	1	No
Conversión	4	1	No

2.2 Formalización de la sintaxis

```
Program → program ident illave SConsts STypes SVars SSubprogs SInsts fllave fin

SConsts → consts illave Consts fllave | ε
Consts → Consts pyc Const | Const
Const → const TPrim ident asig ConstLit | ε

ConstLit → Lit | menos Lit

STypes → tipos illave Types fllave | ε
Types → Types pyc Type | Type
Type → tipo TypeDesc ident | ε

SVars → vars illave Vars fllave | ε
Vars → Vars pyc Var | Var
Var → var TypeDesc ident | ε

SSubprogs → subprograms illave Subprogs fllave | subprograms illave fllave | ε
Subprogs → Subprogs Subprog | Subprog
Subprog → subprogram ident ipar SParams fpar illave SVars SInsts fllave

SFParams → FParams | ε
FParams → FParams coma FParam | FParam
FParam → TypeDesc ident | TypeDesc mul ident

TypeDesc → TPrim | TArray | TTupla | ident

TPrim → natural | integer | float | boolean | character
Cast → char | int | nat | float

TArray → TypeDesc icorchete ident fcorchete | TypeDesc icorchete litnat fcorchete

TTupla → ipar Tupla fpar | ipar fpar
Tupla → TypeDesc coma Tupla | TypeDesc

SInsts → instructions illave Insts fllave
Insts → Insts pyc Inst | Inst
Inst → Desig asig Expr
     | in ipar Desig fpar
     | out ipar Expr fpar
     | swap1 ipar fpar
     | swap2 ipar fpar
     | if Expr then Insts ElseIf
     | while Expr do Insts endwhile
     | InstCall
     | ε
ElseIf → else Insts endif | endif
InstCall → call ident lpar SRParams rpar

SRParams → RParams | ε
RParams → RParams coma RParam | RParam
RParam → ident asig Expr

Desig → ident | Desig icorchete Expr fcorchete | Desig barrabaja litnat

Expr → Term Op0 Term | Term
Term → Term Op1 Fact | Term or Fact | Fact
Fact → Fact Op2 Shft | Fact and Shft | Shft
```

```
Shft → Unary Op3 Shft | Unary
Unary → Op4 Unary | lpar Cast rpar Paren | Paren
Paren → lpar Expr rpar | Lit | Desig

Op0 → igual | noigual | men | may | menoig | mayoig
Op1 →  menos | mas
Op2 →  mod | div | mul
Op3 → lsh | rsh
Op4 → not | menos

Lit → LitBool | LitNum | litchar
LitBool → true | false
LitNum → litnat | litfloat
```

3. Estructura y construcción de la tabla de símbolos

3.1 Estructura de la tabla de símbolos

id: Si es un tipo construido es el nombre del tipo. Si es una variables o una constante es el identificador.

clase: Indica si es la declaración de un tipo construido, una variable, una constante, un subprograma, un parámetro por valor o un parámetro por referencia.

nivel: Indica si la variable es de nivel `global`, en el programa principal o bien de nivel `local` si la variable es de un subprograma

dir: Dirección de memoria asignada. Solo para variables y constantes no para tipos construidos.

tipo: Almacena los conjuntos de propiedades con la información necesaria del tipo.

valor: Si es una constante, almacena su valor. Si no, es indefinido.

3.2 Construcción de la tabla de símbolos

3.2.1 Funciones semánticas

creaTS() : TS

> Crea una tabla de símbolos vacía.

creaTS(ts:TS) : TS

> Dada una tabla de símbolos crea otra tabla de símbolos que contiene toda la información de la tabla recibida por parámetro. Esta constructora se usa para las tablas de símbolos de los subprogramas

añade(ts:TS, id:String, clase:String, nivel:String, dir:Int, tipo:CTipo, valor:?) : TS

> Añade a la tabla de símbolos el nuevo tipo construido, una variable o una constante. CTipo es el conjunto de propiedades con la información necesaria del tipo. Está explicado más adelante.

campo?(ts:TS, campos:CCampo, id:String) : Boolean

> Devuelve true cuando la lista de campos de Campo contenga campo id.

desplazamiento(tipo:CTipo, id:String) : Integer

> Devuelve el tamaño que ocupa en memoria el identificador id. Si no hay un identificador con ese nombre devuelve terr

existeID(ts:TS, id:String) : Boolean

> Dada una tabla de símbolos y el campo id de un identificador, indica si el identificador existe en la tabla de símbolos (sensible a mayúsculas y minúsculas), es decir, si ha sido previamente declarado.

obtieneCtipo(typeDesc:TypeDesc) : CTipo

> Dado un descriptor de tipos devuelve el CTipo asociado

obtieneTipoString(ident:String) : String

> Dado un identificador, devuelve su tipo en un String.

stringToNat(v:String) : Natural

> Convierte el atributo pasado como string a un valor natural.

stringToFloat(v:String) : Float

> Convierte el atributo pasado como string a un valor decimal.

stringToChar(v:String) : Character

> Convierte el atributo pasado como string a un carácter

CTipo

CTipo es el conjunto de propiedades con la información necesaria del tipo. CTipo guarda información diferente dependiendo de si es un tipo construido, un array, una tupla, una variable de todo lo anterior dicho o bien una variable o constante de tipo básico.

CTipo en tipos construidos

Cuando la tabla de símbolos guarda un tipo construido, el campo tipo guarda la siguiente información.

```
<id:String, t:reg, tipo:Ctipo, tam:int>
```

CTipo en arrays

Cuando la tabla de símbolos guarda un array, el campo tipo guarda la siguiente información.

```
<id:String, t:array, nelems:int, tbase:Ctipo, tam:int>
```

Ctipo en tuplas

Cuando la tabla de símbolos guarda un array el campotipo guarda la siguiente información.

```
<id:String, t:tupla, nelems:int, campos:CCampos, tam:int>
```

Donde `campos` es una lista de elementos de la forma:

```
<id:int, tipo:CTipo>
```

Ctipo en variables cuando guardan una referencia a otro tipo

Cuando la tabla de símbolos guarda una variable, con una referencia a otro tipo, el campo tipo guarda la siguiente información.

```
<id:String, t:ref, id:String, tam:int>
```

Ctipo en constantes y variables que guardan un tipo primitivo

Cuando la tabla de símbolos guarda una constante o, una variable con tipos primitivos, el campo tipo guarda la siguiente información.

```
<t:int, tam:1>
<t:nat, tam:1>
<t:float, tam:1>
```

```
<t:bool, tam:1>
<t:char, tam:1>
```

Ctipo en subprogramas

Cuando la tabla de símbolos guarda la cabecera de un subprograma, el campo tipo guarda la siguiente información.

```
<id:String, t:subprog, params[...]>
```

La lista `params` guarda los parámetros de entrada que recibe el subprograma. Se distinguen entre los parámetros que son por valor o los que son por referencia. El campo idparam es el string que identifica el parámetro al hacer la llamada al subprograma.

```
<tipo:CTipo, modo:valor, idparam:String>
<tipo:CTipo, modo:variable, idparam:String>
```

3.2.2 Atributos semánticos

ts: tabla de símbolos sintetizada

id: nombre del identificador

clase: Indica si es la declaración de un tipo construido, una variable, una constante, un subprograma, un parámetro por valor o un parámetro por referencia.

nivel Indica si el identificador es de ámbito global o local

dir: Dirección de memoria. Dónde se guarda la variable o la constante

tipo Almacena los conjuntos de propiedades con la información necesaria del tipo

valor: Si es una constante, almacena su valor. Si no, es indefinido.

3.2.3 Gramáticas de atributos

A continuación se detalla la construcción de los atributos relevantes para la creación de la tabla de símbolos. Otros atributos, como la tabla de símbolos heredada (que tan solo se propaga) o el tipo y el valor de las expresiones se detallarán más adelante en sus correspondientes secciones.

La tabla de símbolos comienda a guardar las declaraciones a partir de la dirección 0 de memoria. Ya que la dirección 0 está reservada para la `cima de la pila` y la dirección 1 para la `base`. La base apunta al inicio de los datos del procedimiento actualmente activo.

```
Program → program ident illave SConsts STypes SVars SSubprogs SInsts fllave fin
    Program.tsh = creaTS()
    Program.dirh = 2
    SConsts.tsh = Program.tsh
    STypes.tsh = SConsts.ts
    SVars.tsh = STypes.ts
    SVars.dirh = SProgram.dirh
    SSubprogs.tsh = SVars.ts
    SInsts.tsh = SSubprogs.ts
    SVars.nivelh = global

SConsts → const illave Consts fllave
    Consts.tsh = SConsts.tsh
    SConsts.ts = Consts.ts

SConsts → ε
    SConsts.ts = SConsts.tsh

Consts → Consts pyc Const
    Consts1.tsh = Consts0.tsh
    Const.tsh = Consts1.ts
    Consts0.ts = añade(Const.ts, Const.id, Const.clase, Const.nivel, ?, Const.tipo, Const.valor)
```

```
Consts → Const
    Const.tsh = Consts.tsh
    Consts.ts = añade(Const.ts, Const.id, Const.clase, Const.nivel, ?, Const.tipo, Const.valor)


Const → const TPrim ident asig ConstLit
    Const.ts = Const.tsh
    Const.id = ident.lex
    Const.clase = const
    Const.nivel = global
    Const.tipo = <t:TPrim.tipo, tam:1>
    Const.valor = ConstLit.valor

Const → ε
    Const.ts = Const.tsh

ConstLit → Lit
    ConstLit.valor = Lit.valor
    ConstLit.tipo = Lit.tipo

ConstLit → menos Lit
    ConstLit.valor = -(Lit.valor)
    ConstLit.tipo = opUnario(menos, Lit.tipo)

STypes → tipos illave Types fllave
    Types.tsh = STypes.tsh
    STypes.ts = Types.ts

STypes → ε
    STypes.ts = STypes.tsh

Types → Types pyc Type
    Types1.tsh = Types0.tsh
    Type.tsh = Types1.ts
    Types0.ts = añade(Types1.ts, Type.id, Type.clase, Type.nivel, ?, Type.tipo)

Types → Type
    Type.tsh = Types.tsh
    Types.ts = añade(Type.ts, Type.id, Type.clase, Type.nivel, ?, Type.tipo)


Type → tipo TypeDesc ident
    Type.ts = Type.tsh
    TypeDesc.tsh = Type.tsh
    Type.id = ident.lex
    Type.clase = Tipo
    Type.nivel = global
     Type.tipo = <t:TypeDesc.tipo, tipo:obtieneCTipo(TypeDesc), tam:desplazamiento(TypeDesc.tipo, Var.tsh ), Type.id)>

Type → ε
    Type.ts = Type.tsh

SVars → vars illave Vars fllave
    Vars.tsh = SVars.tsh
    Vars.dirh = SVars.dirh
    SVars.ts = Vars.ts
    SVars.dir = Vars.dir

SVars → ε
    SVars.ts = SVars.tsh
    SVars.dir = SVars.dirh

Vars → Vars pyc Var
    Vars1.tsh = Vars0.tsh
    Vars1.dirh = Vars0.dirh
    Var.tsh = Vars1.ts
    Var.dirh = Vars1.dir
    Vars0.dir = Var.dir + desplazamiento(Var.tipo, Vars1.id)
```

```
    Vars0.ts = añade(Var.ts, Var.id, Var.clase, Var.nivel, Vars0.dir, Var.tipo)
    Vars1.nivelh = Vars0.nivelh
    Var.nivelh = Vars0.nivelh

Vars → Var
    Var.tsh = Vars.tsh
    Var.dirh = Vars.dirh
    Vars.dir = Var.dir + desplazamiento(Var.tipo, Var.id)
    Vars.ts = añade(Var.ts, Var.id, Var.clase, Var.nivel, Var.dir, Var.tipo)
    Var.nivelh = Vars.nivelh

Var → var TypeDesc ident
    Var.ts = Var.tsh
    Var.dir = Var.dirh
    Var.id = ident.lex
    Var.clase = Var
    Var.nivel = Var.nivelh
    Var.tipo = si (TypeDesc.tipo == TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<id:Var.id, t:ref, TypeDesc.tipo tam: desplazamiento(TypeDesc.tipo, Var.tsh )>}
    TypeDesc.tsh = Var.tsh

Var → ε
    Var.ts = Var.tsh
    Var.dir = Var.dirh

TypeDesc → TPrim
    TypeDesc.tipo = TPrim.tipo

TypeDesc → TArray
    TypeDesc.tipo = TArray.tipo
    TArray.tsh = TypeDesc.tsh

TypeDesc → TTupla
    TypeDesc.tipo = TTupla.tipo
    TTupla.tsh = TypeDesc.tsh

TypeDesc → ident
    TypeDesc.tipo = ident.lex

TPrim → natural
    TPrim.tipo = natural

TPrim → integer
    TPrim.tipo = integer

TPrim → float
    TPrim.tipo = float

TPrim → boolean
    TPrim.tipo = boolean

TPrim → character
    TPrim.tipo = character

TArray → TypeDesc icorchete ident fcorchete
    TypeDesc.tsh = TArray.tsh
    TArray.tsh = TypeDesc.tsh

TArray → TypeDesc icorchete litnat fcorchete
    TypeDesc.tsh = TArray.tsh
    TArray.tsh = TypeDesc.tsh

TTupla → ipar Tupla fpar
    Tupla.tsh = TTupla.tsh
    TTupla.tipo = Tupla.tipo

TTupla → ipar fpar

Tupla → TypeDesc coma Tupla
```

```
    TypeDesc.tsh = Tupla0.tsh
    Tupla1.tsh = Tupla0.tsh
    Tupla0.tipo = TypeDesc.tipo ++ Tupla1.tipo

Tupla → TypeDesc
    TypeDesc.tsh = Tupla.tsh
    Tupla.tipo = TypeDesc.tipo


SSubprogs → subprograms illave Subprogs fllave
    Subprogs.tsh = SSubprogs.tsh
    SSbprogs.ts = Subprog.ts

SSubprogs → subprograms illave fllave
    SSubprogs.tsh = Subprog.tsh

SSubprogs → ε
    SSubprogs.tsh = Subprog.tsh

Subprogs → Subprogs Subprog
    Subprogs1.tsh =  Subprogs0.tsh
    Subprog.tsh = Subprogs0.tsh
    Subprogs0.ts = Subprog.ts

Subprogs → Subprog
    Subprog.tsh = Subprogs.tsh
    Subprogs.ts = Subprog.ts

Subprog → subprogram ident ipar SFParams fpar illave SVars SInsts fllave
    SFParams.dirh = 0
    SFParams.tsh = CreaTS(Subprog.ts)
    SVars.tsh = SFParams.ts
    SVars.dirh = SFParams.dir
    SInsts.tsh = SVars.ts
    Subprog.ts = añade(Subprog.tsh, ident, subprog, global, ? , <dir:Subprog.etqh, params:SFParams.params>)
    SVars.nivelh = local

SFParams → FParams
    FParams.tsh = SFParams.tsh
    SFParams.ts = FParams.ts
    FParams.dirh = SFParams.dirh
    SFParams.dir = FParams.dir
    SFParams.params = FParams.params

SFParams → ε
    SFParams.ts = SFParams.tsh
    SFParams.dir = SFParams.dirh
    SFParams.params = []

FParams → FParams coma FParam
    FParams1.tsh = FParams0.tsh
    FParams1.dirh = FParams0.dirh
    FParam.tsh = FParams1.tsh
    FParam.dirh = FParams1.dirh
    FParams0.dir = FParam.dir + desplazamiento(FParam.tipo, FParam.id)
    FParams0.ts = añade(FParam.ts, FParam.id, FParam.clase, FParam.nivel, FParam.dir, FParam.tipo)
    FParams0.params = FParams1.params ++ FParam.params

FParams → FParam
    FParam.dirh = FParams.dirh
    FParam.tsh = FParams.tsh
    FParams.ts = añade(FParam.ts, FParam.id, FParam.clase, FParam.nivel, FParam.dir, FParam.tipo)
    FParams.dir = FParam.dir + desplazamiento(FParam.tipo, FParam.id)
    FParams.params = FParap.params

FParam → TypeDesc ident
    FParam.ts = FParam.tsh
    FParam.dir = FParam.dirh
    Fparam.id = ident.lex
```

```
    FParam.clase = pvalor
    FParam.nivel = local
    FParam.tipo = (si (TypeDesc.tipo== TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<t:ref, id:FParam.id, tam: desplazamiento(TypeDesc.tipo, Param.id)>} )
    FParam.params = [<id:FParam.id, tam:desplazamiento(TypeDesc.tipo, Param.id), ref:falso, despl:DParam.dirh>]
    TypeDesc.tsh = FParam.tsh

FParam → TypeDesc mul ident
    FParam.ts = FParam.tsh
    FParam.dir =  FParam.dirh
    Fparam.id = ident.lex
    FParam.clase = pvariable
    FParam.nivel = local
    FParam.tipo = (si (TypeDesc.tipo == TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<t:ref, id:FParam.id, tam: 1>} )
    FParam.params = [<id:FParam.id, tam:desplazamiento(TypeDesc.tipo, Param.id), ref:cierto, despl:DParam.dirh>]
    TypeDesc.tsh = FParam.tsh

Lit → LitBool
    Lit.valor = LitBool.valor
    Lit.tipo = LitBool.tipo

Lit → LitNum
    Lit.valor = LitNum.valor
    Lit.tipo = LitNum.tipo

Lit → litChar
    Lit.valor = stringToChar(litchar)
    Lit.tipo = character

LitBool → true
    LitBool.valor = true
    Litbool.tipo = boolean

LitBool → false
    LitBool.valor = false
    Lit.tipo = boolean

LitNum → litnat
    LitNum.valor = stringToNat(litnat.lex)
    LitNum.tipo = natural

LitNum → litfloat
    LitNum.valor = stringToFloat(litfloat.lex)
    LitNum.tipo = float
```

4. Especificación de las restricciones contextuales

4.1 Descripción informal de las restricciones contextuales

Enumeración y descripción de las restricciones contextuales extraídas directamente del enunciado.

4.1.1 Sobre declaraciones

- Las variables, constantes y tipos que se usen en la sección de instrucciones o en la sección de subprogramas habrán debido de ser convenientemente declarados en su correspondiente sección.
- No se pueden declarar dos variables, constantes o tipos con el mismo identificador.

4.1.2 Sobre instrucciones de asignación

Una instrucción de asignación debe cumplir además estas condiciones:

- La variable en la parte izquierda debe haber sido declarada.
- No se pueden asignar o hacer instrucciones in a constantes.
- A una variable de tipo real es posible asignarle un valor real, entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un carácter, booleano o tipo construido.
- A una variable de tipo entero es posible asignarle un valor entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un valor real, carácter, boolean o tipo construido.
- A una variable de tipo natural únicamente es posible asignarle un valor natural.
- A una variable de tipo carácter únicamente es posible asignarle un valor de tipo carácter.
- A una variable de tipo booleano únicamente es posible asignarle un valor de tipo booleano.
- A una variable de tipo construido únicamente es posible asignarle un valor de ese mismo tipo construido.

4.1.3. Sobre comparaciones

No se puede comparar naturales con caracteres, ni enteros con caracteres, ni reales con caracteres, ni booleanos con caracteres. Tampoco se puede comparar naturales con booleanos, ni enteros con booleanos, ni reales con booleanos, ni caracteres con booleanos.

4.1.4. Sobre operadores

- Los operadores +, -, *, / sólo operan con valores numéricos. No podemos aplicarlos ni a los caracteres, ni a los booleanos ni a los tipos construidos.
- En la operación módulo % el primer operando puede ser entero o natural, pero el segundo operando sólo puede ser natural. El resultado de a % b será el resto de la división de a entre b. El tipo del resultado será el mismo que el del primer operando.
- Los operadores lógicos ‘or’, ‘and’, ‘not’ sólo operan sobre valores booleanos. No podemos aplicarlos ni a los numéricos, ni a los caracteres ni a los tipos construidos.
- Los operadores << y >> sólo operan con valores numéricos naturales.

4.1.5. Sobre operadores de conversión

- **(float)** puede ser aplicado a cualquier tipo excepto al tipo booleano y a los tipos construidos.
- **(int)** puede ser aplicado a cualquier tipo excepto al tipo booleano y a los tipos construidos.
- **(nat)** puede ser aplicado al tipo natural y al tipo carácter. No admite operandos reales, enteros, booleanos o de tipos construidos.
- **(char)** puede ser aplicado al tipo carácter y al tipo natural. No admite operandos reales, enteros, booleanos o de tipos construidos.

4.1.6 Sobre los subprogramas

Sobre la invocación de subprogramas

- No se puede invocar a un subprograma que no esté previamente declarado
- Hay que comprobar que los parámetros reales con los que se invoca al subprograma son correctos. Es decir, comprobar que:
  - Se invoque con el mismo número de parámetros que el declarado en la cabecera del subprograma.
  - Que cada parámetro se invoque con un identificador que haya sido declarado en la cabecera.
  - Que no haya dos parámetros reales invocados con el mismo identificador
  - Comprobar que, cuando pasamos un parámetro por referencia, sea un designador.
  - Que los parámetros que pasamos estén previamente declarados en la table de símbolos
  - Que los parámetros reales que pasemos sean compatibles con el tipo del parámetros formal declarado en la cabecera de la función.

Sobre la declaración de subprogramas hay que comprobar

- Que no declaremos dos parámetros formales de entrada, con el mismo identificador.
- Que no haya un subprograma declarado previamente con el mismo identificador.

4.2 Funciones semánticas

A continuación, describimos las funciones semánticas adicionales utilizadas en la descripción.

casting

```
casting (Type tipoCast, Type tipoOrg) : Type
    Dados dos tipos diferentes comprobamos si podemos hacer el casting: [ (tipoCast) tipoOrg ] Si podemos, devolvemos el
```

TipoCast	TipoOrg	Tipo devuelto
natural	natural	natural
natural	character	natural
natural	cualquier otro tipo	terr
boolean	-	terr
character	character	character
character	natural	character
character	cualquier otro tipo	terr
integer	boolean	terr
integer	tipo númerico o character	integer
float	boolean	terr
float	tipo númerico o character	terr

Nota: cualquier casting en el que esté involucrado un tipo construido da como tipo devuelto 'terr'.

unario

```
unario(Type OpUnario, Type tipoUnario) : Type
    Dado un operador unario y el tipo al que es aplicado comprobamos si se puede aplicar. Por ejemplo, no podemos aplica
```

OpUnario	tipoUnario	Tipo devuelto
"-"	natural	integer
"-"	integer	integer
"-"	float	float
"-"	cualquier otro tipo	terr
not	boolean	boolean
not	cualquier otro tipo	terr

Nota: no se puede aplicar ningún operador unario a ningún tipo construido.

tipoFunc

```
tipoFunc(Type tipo1, Operator op, Type tipo2) : Type
    Dados dos tipos diferentes y un operador comprobamos que los tipos puedan aplicar el operador. Devolvemos el tipo co
```

Si pusiésemos todas las posibilidades la tabla resultante quedaría muy extensa. Para simplificar, se pondrán dos tablas. En la

primera, se pondrán los operadores conmutativos. Es decir, aquellos que se comportan igual sean los tipos asignados al primer parámetro de la función o al segundo. En la segunda se pondrán los operadores no conmutativos. En los que importa quién sea el tipo1 y el tipo2.

También para que se vea mejor, dentro de las tablas, separaremos los tipos de operadores. Operadores conmutativos:

Tipo1	Op	Tipo2	Tipo devuelto
tipo numérico	cualquier op. de comparación	tipo numérico	boolean
boolean	cualquier op. de comparación	boolean	boolean
character	cualquier op. de comparación	character	boolean
boolean	cualquier op. aritmética	-	terr
character	cualquier op. aritmética	-	terr
float	cualquier op. aritmética	cualquier tipo numérico	float
integer	cualquier op. aritmética	integer o natural	integer
natural	cualquier op. aritmética	natural	natural
boolean	cualquier op. lógica	boolean	boolean
cualquier otro tipo	cualquier op. lógica	-	terr
natural	"<<"	natural	natural
natural	">>"	natural	natural
tipo no natural	"<<"	-	terr
-	"<<"	tipo no natural	terr

Nota: el tipo devuelto de aplicar cualquier tipo de operador a un tipo construido es 'terr'.

Operadores no conmutativos:

Tipo1	Op	Tipo2	Tipo devuelto
integer o natural	"%"	natural	natural
-	"%"	tipo no natural	terr
ni integer ni natural	"%"	-	terr

asignaciónVálida

```
asignaciónVálida(Type tipoDesig, Type tipoExp) : Boolean
    Dado un tipo de un designador y un tipo de una expresión, comprueba si ambos son tipos compatibles. Por ejemplo, no
```

Para que se vea mejor, dentro de las tablas, separaremos los tipos posibles de tipoDesig.

TipoDesig	TipoExp	Tipo devuelto
natural	natural	true
natural	cualquier otro tipo	false
integer	natural	true
integer	integer	true
integer	cualquier otro tipo	false
float	tipo numérico	true

float	cualquier otro tipo	false
boolean	boolean	boolean
boolean	cualquier otro tipo	false
character	character	true
character	cualquier otro tipo	false

Nota: En el caso de los tipos construidos, devolverá true siempre que los dos tipos sean compatibles, y false en c.o.c. Dos tipos se consideran compatibles cuando el tipo de sus componentes es el mismo y, en el caso de los arrays, su tamaño es el mismo.

esVariable

```
esVariable(TS ts, String id) : Boolean
    Indica si el ident dado, representado por su id, es una variable o una constante. Si devuelve true quiere decir que
```

existe

```
existe(TS ts, String id) : Boolean
    Indica si el identificador existe en la tabla de símbolos

existe(TS ts, String is, nivel) : Boolean
    Indica si el identificador existe en la tabla de símbolos en el nivel inidicado.
```

añadirSubprograma

```
añadirSubprograma(TS ts, String ident, CCampo params, Integer address) : void
    Añade a la tabla de símbolos el subprograma definido por los argumentos.
```

numParametros

```
numParametros(TS ts, String id) : Integer
    Devuelve el número de parámetros que tiene el subprograma con el identificador id. Si el subprograma no está en la t
```

estaDeclarado

```
estaDeclarado(TS ts, String idparam, String idsubprog) : Boolean
    Comprueba si el parámetro idparam está declarado en el subprograma idsubprog. Si no está declarado el identificador,
```

compatible

```
compatible(CTipo tipo1, CTipo tipo2) : Boolean
    Dados dos tipos nos indica si son campatibles entre ellos
```

getOffset

```
getOffset(Integer numElems) : Integer
    Devuelve la posición del elemento dado dentro de la tupla.
```

parametrosNoRepetidos

```
parametrosNoRepetidos(TS ts, String id)
    Dado el nombre de un identificador de un subprograma "id" comprobamos que no hay dos identificadores de parámetros e
```

Nota:

En todas las funciones, si alguno de los tipos de entrada es el tipo terr, devolvemos siempre terr.

4.3 Atributos semánticos

- **op:** atributo que indica cuál es el operador usado.
- **ts:** tabla de símbolos. Se crea en la parte de declaraciones.
- **tsh:** tabla de símbolos heredada. Se hereda en la parte de instrucciones.
- **err:** atributo que indica si se ha detectado algún error. Es un atributo de tipo booleano.
- **nparams:** contador que cuenta cuántos parámetros se han pasado en la llamada (call) a un subprograma.
- **nombresubprog:** lleva el identificador el subprograma. Se usa para las restricciones contextuales en el paso de parámetros a funciones.
- **listaparamnombres:** lleva una lista con los nombres de los parámetros que han sido introducidos en una llamada a función.

4.4 Gramática de atributos

```
Program → program ident illave SConsts STypes SVars SSubprogs SInsts fllave fin
    Program.tsh = creaTS()
    SConsts.tsh = Program.tsh
    STypes.tsh = SConsts.ts
    SVars.tsh = STypes.ts
    SSubprogs.tsh = SVars.ts
    SInsts.tsh = SVars.ts
    Program.err = SConsts.err ∨ STypes.err ∨ SVars.err ∨ SSubprogs.err ∨ SInsts.err

SConsts → const illave Consts fllave
    Consts.tsh = SConsts.tsh
    SConsts.ts = Consts.ts
    SConsts.err = Consts.err

SConsts → ε
    SConsts.ts = SConsts.tsh
    SConsts.err = false

Consts → Consts pyc Const
    Consts1.tsh = Consts0.tsh
    Const.tsh = Consts1.ts
    Consts0.ts = añade(Const.ts, Const.id, Const.clase, Const.nivel, Conts0.dir, Const.tipo)
    Consts.err = existe(Const.ts, Const.id)

Consts → Const
    Const.tsh = Consts.tsh
    Consts.ts = añade(Const.ts, Const.id, Const.clase, Const.nivel, Const.dir, Const.tipo)
    Consts.err = existe(Const.ts, Const.id)

Const → const TPrim ident asig ConstLit
    Const.ts = Const.tsh
    Const.id = ident.lex
    Const.clase = const
    Const.nivel = global
    Const.tipo = <t:TPrim.tipo, tam:1>

    Const.valor = ConstLit.valor
    Const.err = ¬(compatibles(TPrim.tipo, ConstLit.tipo))

Const → ε
    Const.ts = Const.tsh
    Const.err = false

ConstLit → Lit
    ConstLit.tipo = Lit.tipo

ConstLit → menos Lit
    ConstLit.tipo = opUnario(menos, Lit.tipo)

STypes → tipos illave Types fllave
    Types.tsh = STypes.tsh
```

```
    STypes.ts = Types.ts
    STypes.err = Types.err

STypes → ε
    STypes.ts = STypes.tsh
    STypes.err = false

Types → Types pyc Type
    Types1.tsh = Types0.tsh
    Type.tsh = Types1.ts
    Types0.ts = añade(Types1.ts, Type.id, Type.clase, Type.nivel, Types0.dir, Type.tipo)
    Types0.err = existe(Types1.ts, Type.id)

Types → Type
    Type.tsh = Types.tsh
    Types.ts = añade(Type.ts, Type.id, Type.clase, Type.nivel, Type.dir, Type.tipo)
    Types.err = existe(Type.ts, Type.id)

Type → tipo TypeDesc ident
    Type.ts = Type.tsh
    Type.id = ident.lex
    Type.clase = Tipo
    Type.nivel = global
    Type.tipo = <t:TypeDesc.tipo, tipo:obtieneCTipo(TypeDesc), tam:desplazamiento(TypeDesc.tipo, Var.tsh ), Type.id)>

Type → ε
    Type.ts = Type.tsh
    Type.err = false

SVars → vars illave Vars fllave
    Vars.tsh = SVars.tsh
    SVars.ts = Vars.ts
    SVars.err = Vars.err

SVars → ε
    SVars.ts = SVars.tsh
    SVars.err = false

Vars → Vars pyc Var
    Vars1.tsh = Vars0.tsh
    Var.tsh = Vars1.ts
    Vars0.ts = añade(Var.ts, Var.id, Var.clase, Var.nivel, Vars0.dir, Var.tipo)
    Vars0.err = existe(Var.ts, Var.id, Var.nivel)

Vars → Var
    Var.tsh = Vars.tsh
    Vars.ts = añade(Var.ts, Var.id, Var.clase, Var.nivel, Var.dir, Var.tipo)
    Vars.err = existe(Var.ts, Var.id, Var.nivel)

Var → var TypeDesc ident
    Var.ts = Var.tsh
    Var.id = ident.lex
    Var.clase = Var
    Var.nivel = global
    Var.tipo = si (TypeDesc.tipo == TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<id:Var.id, t:ref, TypeDesc.tipo, tam: desplazamiento(TypeDesc.tipo, Var.tsh )>}
Var → ε
    Var.ts = Var.tsh
    Var.err = false

SSubprogs → subprograms illave Subprogs fllave
    Subprogs.tsh = SSubprogs.tsh
    SSubprogs.err = Subprogrs.err

SSubprogs → subprograms illave fllave

SSubprogs → ε
    SSubprogs.err = false
```

```
Subprogs → Subprogs Subprog
    Subprogs1.tsh = Subprogs0.tsh
    Subprog.tsh = Subprogs0.tsh
    Subprogs0.err = Subprogs1.err ∨ Subprog.err

Subprogs → Subprog
    Subprog.tsh = Subprogs.tsh
    Subprogs.err = Subprog.err

Subprog → subprogram ident ipar SFParams fpar illave SVars SInsts fllave
    SFParams.tsh = CreaTS(Subprog.tsh)
    SVars.tsh = SFParams.ts
    SInsts.tsh = SVars.ts
    Subprog.err = existe(Subprog.tsh, ident) ∨ SParams.err ∨ SVars.err ∨ SInsts.err ∨ parametrosNoRepetidos(SParams.ts,

SFParams → FParams
    FParams.tsh = SFParams.tsh
    SFParams.ts = FParams.ts
    SFParams.dir = FParams.dir
    SFParams.err = FParams.err

SFParams → ε
    SFParams.ts = SFParams.tsh
    SFParams.err = false

FParams → FParams coma FParam
    FParams1.tsh = FParams0.tsh
    FParam.tsh = FParams1.tsh
    FParams0.ts = añade(FParam.ts, FParam.id, FParam.clase, FParam.nivel, FParam.dir, FParam.tipo)
    FParams0.err = existe(FParam.ts, FParam.id, FParam.nivel)

FParams → FParam
    FParam.tsh = FParams.tsh
    FParams.ts = añade(FParam.ts, FParam.id, FParam.clase, FParam.nivel, FParam.dir, FParam.tipo)
    FParams.err = existe(FParam.ts, FParam.id, FParam.nivel)

FParam → TypeDesc ident
    FParam.ts = FParam.tsh
    Fparam.id = ident.lex
    FParam.clase = pvalor
    FParam.nivel = local
    FParam.tipo = (si (TypeDesc.tipo== TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<t:ref, id:FParam.id, tam: desplazamiento(TypeDesc.tipo, Param.id)>} )

FParam → TypeDesc mul ident
    FParam.ts = FParam.tsh
    Fparam.id = ident.lex
    FParam.clase = pvariable
    FParam.nivel = local
    FParam.tipo = (si (TypeDesc.tipo == TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<t:ref, id:FParam.id, tam: desplazamiento(TypeDesc.tipo, Param.id)>} )

TypeDesc → TPrim

TypeDesc → TArray
    TArray.tsh = TypeDesc.tsh
    TypeDesc.err = TArray.err

TypeDesc → TTupla
    TTupla.tsh = TypeDesc.tsh
    TypeDesc.err = TTupla.err

TypeDesc → ident
    TypeDesc.err = ¬existe(TypeDesc.tsh, ident.lex) ∨ TypeDesc.tsh[ident].clase != tipo

TPrim → natural | integer | float | boolean | character
Cast → char | int | nat | float

TArray → TypeDesc icorchete ident fcorchete
```

```
    TypeDesc.tsh = TArray.tsh
    TArray.err = ¬existe(TArray.tsh, ident.lex) ∨ obtieneTipoString(ident) != nat ∨ TArray.tsh[ident].clase != constante

TArray → TypeDesc icorchete litnat fcorchete
    TypeDesc.tsh = TArray.tsh

TTupla → ipar Tupla fpar
    Tupla.tsh = TTupla.tsh
    TTupla.err = Tupla.err

TTupla → ipar fpar
    TTupla.err = false

Tupla → TypeDesc coma Tupla
    TypeDesc.tsh = Tupla0.tsh
    Tupla1.tsh = Tupla0.tsh
    Tupla0.err = TypeDesc.err ∨ Tupla1.err

Tupla → TypeDesc
    TypeDesc.tsh = Tupla.tsh
    Tupla.err = TypeDesc.err

SInsts → instructions illave Insts fllave
    Insts.tsh = SInsts.tsh
    SInsts.err = Insts.err

Insts → Insts pyc Inst
    Insts1.tsh = Insts0.tsh
    Inst.tsh = Insts0.tsh
    Insts0.err = Insts1.err ∨ Inst.err

Insts → Inst
    Inst.tsh = Insts.tsh
    Insts.err = Inst.err

Inst → Desig asig Expr
    Desig.tsh = Inst.tsh
    Expr.tsh = Inst.tsh
    Inst.err = (¬asignacionValida(Desig.tipo, Expr.tipo)) ∨ Expr.err ∨ Desig.err

Inst → in ipar Desig fpar
    Desig.tsh = Inst.tsh
    Inst.err = Desig.err

Inst → out ipar Expr fpar
    Expr.tsh = Inst.tsh
    Inst.err = Expr.err

Inst → swap1 ipar fpar
    Inst.err = false

Inst → swap2 ipar fpar
    Inst.err = false

Inst → if Expr then Insts ElseIf
    Expr.tsh = Inst.tsh
    Insts1.tsh = Inst0.tsh
    ElseIf.tsh = Inst.tsh
    Inst.err = Expr.err ∨ Insts.err ∨ ElseIf.err

Inst → while Expr do Insts endwhile
    Expr.tsh = Inst.tsh
    Insts.tsh = Inst.tsh
    Inst.err = Expr.err ∨ Insts.err

Inst → InstCall
    InstCall.tsh = Inst.tsh
    Inst.err = InstCall.err
```

```
Inst → ε
    Inst.err = false

ElseIf → else Insts endif
    Insts.tsh = ElseIf.tsh
    ElseIf.err = Insts.err

ElseIf → endif
    ElseIf.err = false

InstCall → call ident lpar SRParams rpar
    SRParams.tsh = InstCall.tsh
    SRParams.nparams = 0
    SRParams.nombresubprogh = ident.lex
    SRParmas.listaparamnombresh = []
    InstCall.err = SRParams.err ∨ ¬existe(SRParams.tsh, ident.lex) ∨ SRParams.nparams != numParametros(SRParams.tsh, ide

SRParams → RParams
    RParams.tsh = SRParams.tsh
    RParams.nparamsh = SRParams.nparamsh
    SRParams.nparams = RParams.nparams
    RParams.nombresubprogh = SRParams.nombresubprogh
    RParams.listaparamnombresh = SRParams.listaparamnombresh
    SRParams.err = RParams.err

SRParams → ε
    SRParams.err = false
    SRParams.nparams = SRParams.nparamsh
    SRParams.listaparamnombres = SRParams.listaparamnombresh

RParams → RParams coma RParam
    RParams1.tsh = RParams0.tsh
    RParam.tsh = RParams0.tsh
    RParams0.err = RParams1.err ∨ Rparam.err
    RParams1.nparamsh = RParams0.nparamsh
    RParam.nparamsh = RParams1.nparams
    RParams.nparams = RParam.nparams
    RParams1.nombresubprogh = RParams0.nombresubprogh
    RParam.nombresubprogh = RParams0.nombresubprogh
    RParams1.listaparamnombresh = RParams0.listaparamnombresh
    RParam.listaparamnombresh = RParams1.listaparamnombres

RParams → RParam
    RParam.tsh = RParams.tsh
    RParam.nparamsh = RParams.nparamsh
    RParams.nparams = RParam.nparams
    RParam.nombresubprogh = RParams.nombresubprogh
    RParam.listaparamnombresh = RParams.listaparamnombresh
    RParams.listaparamnombres = RParam.listaparamnombres
    RParams.err = RParam.err

RParam → ident asig Expr
    Expr.tsh = RParam.tsh
    RParam.nparams = RParams.nparamsh + 1
    RParam.listaparamnombres = RParam.listaparamnombresh ++ ident
    RParam.err = Expr.err ∨ ¬existe(Exp.tsh, ident.lex) ∨ ¬esVariable(Expr.tsh, ident.lex)
    ∨ ¬estaDeclarado(RParam.tsh, ident.lex, RParam.nombresubprogh) ∨ ¬compatible(ident.tipo,Expr.tipo) ∨ ¬Expr.desig ∨ (

Desig → ident
    Desig.tipo = Desig.tsh[ident.lex].tipo
    Desig.err = ¬existe(Desig.tsh, ident.lex) ∨ ¬esVariable(Desig.tsh, ident.lex)

Desig → Desig icorchete Expr fcorchete
    Desig0.tipo = Desig1.tipo
    Desig0.err = Desig1.err ∨ Expr.err ∨ ¬tamañoCorrecto()

Desig → Desig barrabaja litnat
    Desig0.tipo = Desig1.tipo
    Desig0.err = Desig1.err ∨ ¬tamañoCorrecto()
```

```
Expr → Term Op0 Term
    Expr.desig = false
    Expr.tipo = tipoFunc(Term0.tipo, Op0.op, Term1.tipo)
    Term0.tsh = Expr.tsh
    Term1.tsh = Expr.tsh
    Expr.desig = false

Expr → Term
    Expr.tipo = Term.tipo
    Term.tsh = Expr.tsh
    Expr.desig = false
    Expr.desig = Term.desig

Term → Term Op1 Fact
    Term0.tipo = tipoFunc(Term1.tipo, Op1.op, Fact.tipo)
    Term1.tsh = Term0.tsh
    Fact.tsh = Term0.tsh
    Term0.desig = false

Term → Term or Fact
    Term0.tipo = tipoFunc(Term1.tipo, or, Fact.tipo)
    Term1.tsh = Term0.tsh
    Fact.tsh = Term0.tsh
    Term0.desig = false

Term → Fact
    Term.tipo = Fact.tipo
    Fact.tsh = Term.tsh
    Term.desig = Fact.desig

Fact → Fact Op2 Shft
    Fact0.tipo = tipoFunc(Fact1.tipo, Op2.op, Shft.tipo)
    Fact1.tsh = Fact0.tsh
    Shft.tsh = Fact0.tsh
    Fact0.desig = false

Fact → Fact and Shft
    Fact0.tipo = tipoFunc(Fact1.tipo, and, Shft.tipo)
    Fact1.tsh = Fact0.tsh
    Shft.tsh = Fact0.tsh
    Fact0.desig = false

Fact → Shft
    Fact.tipo = Shft.tipo
    Shft.tsh = Fact.tsh
    Fact.desig = Shft.desig

Shft → Unary Op3 Shft
    Shft0.tipo = tipoFunc(Unary.tipo, Op3.op, Shft.tipo)
    Unary.tsh = Shft0.tsh
    Shft1.tsh = Shft0.tsh
    Shft0.desig = false

Shft → Unary
    Shft.tipo = Unary.tipo
    Unary.tsh = Shft.tsh
    Shft.desig = Unary.desig

Unary → Op4 Unary
    Unary0.tipo = opUnario(Op4.op, Unary1.tipo)
    Unary1.tsh = Unary0.tsh
    Unary0.desig = false

Unary → lpar Cast rpar Paren
    Unary.tipo = casting(Cast.tipo, Paren.tipo)
    Paren.tsh = Unary.tsh
    Unary.desig = false
```

```
Unary → Paren
    Unary.tipo = Paren.tipo
    Paren.tsh = Unary.tsh
    Unary.desig = Paren.desig

Paren → lpar Expr rpar
    Paren.tipo = Expr.tipo
    Expr.tsh = Paren.tsh
    Paren.desig = false

Paren → Lit
    Parent.tipo = Lit.tipo
    Lit.tsh = Paren.tsh
    Paren.desig = false
    Paren.err = false

Paren → Desig
    Paren.desig = true
    Paren.err = Desig.err

Op0 → igual
    Op0.op = igual

Op0 → noigual
    Op0.op = noigual

Op0 → men
    Op0.op = men

Op0 → may
    Op0.op = may

Op0 → menoig
    Op0.op = menoig

Op0 → mayoig
    Op0.op = mayoig

Op1 → menos
    Op1.op = menos

Op1 → mas
    Op1.op = mas

Op2 → mod
    Op2.op = mod

Op2 → div
    Op2.op = div

Op2 → mul
    Op2.op = mul

Op3 → lsh
    Op3.op = lsh

Op3 → rsh
    Op3.op = rsh

Op4 → not
    Op4.op = not

Op4 → menos
    Op4.op = menos

Lit → LitBool
    Lit.tipo = boolean

Lit → LitNum
```

```
    Lit.tipo = LitNum.tipo

Lit → litchar
    Lit.tipo = char

LitNum → litnat
    LitNum.tipo = natural

LitNum → litfloat
    LitNum.tipo = float
```

5. Especificación de la traducción

5.1 Lenguaje objeto y máquina virtual

5.1.1 Arquitectura

- Mem: Memoria principal con celdas direccionables con datos. Los datos de la memoria no incluyen información sobre de qué tipo son, las instrucciones sí.
- Prog: Memoria de programa con celdas direccionables con instrucciones.
- CProg: Contador de programa con un registro para la dirección de la instrucción actualmente en ejecución
- Pila: Pila de datos con celdas direccionables con datos. No se incluye información sobre el tipo.
- CPila: Cima de la pila de datos con un registro para la dirección del dato situado actualmente en la cima de la pila.
- P: Flag de parada que detiene la ejecución si tiene valor 1.
- S1: Flag de swap1. Si tiene valor 1 intercambia suma por resta y viceversa.
- S2: Flag de swap2. Si tiene valor 1 intercambia multiplicación por división y viceversa.

5.1.2 Comportamiento interno

Pseudocódigo del algoritmo de su ejecución:

> CPila ← -1
> CProg ← 0
> S1 ← 0
> S2 ← 0
> P ← 0
> mientras P = 0
> ejecutar Prog[CProg]
> fmientras

- Mem[dirección]: Dato de una celda de memoria principal localizado a través de una dirección.
- Prog[dirección]: Instrucción de una celda de memoria de programa localizado a través de una dirección.

La dirección -1 en CPila indica que la pila está vacía.

5.1.3 Repertorio de instrucciones

Operaciones con la Pila:

apila(valor)

```
CPila ← CPila + 1
Pila[CPila] ← valor
CProg ← CProg + 1
```

apila-dir(dirección)

```
CPila ← CPila + 1
Pila[CPila] ← Mem[dirección]
CProg ← CProg + 1
```

apila-ind

```
Pila[CPila] ← Mem[Pila[CPila]]
CProg ← CProg + 1
```

apila-ret

```
Pila[Cpila] ← CProg
Cpila ← CPila +1
CProg ← Cprog + 1
```

mueve(nCeldas)

```
para i ← 0 hasta nCeldas-1 hacer
Mem[Pila[CPila]+i] ← Mem[Pila[CPila-1]+i]
CPila ← Cpila - 2
CProg ← CProg + 1
```

Nota: Si la dirección de memoria no ha sido cargada previamente con datos usando la siguiente instrucción (desapila-dir), esta instrucción dará un error de ejecución.

ir_ind

```
CprogPila[CPila]
Cpila←Cpila-1
```

desapila-dir(dirección)

```
Mem[dirección] ← Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

desapila-ind

```
Mem[Pila[CPila]] ← Pila[CPila-1]
CPila ← CPila - 2
CProg ← CProg + 1
```

desapila-ret

```
Mem[Pila[Cpila]] ← CProg
Cpila ← CPila -1
CProg ← Cprog + 1
```

copia

```
CPila ← CPila + 1
Pila[CPila] ← Pila[CPila-1]
CProg ← CProg + 1
```

Saltos

ir-a(direccion)

```
CProg ← direccion
```

ir-v(direccion)

```
si Pila[CPila]: CProg ← direccion
si no: CProg ← CProg + 1
CPila ← CPila-1
```

ir-f(direccion)

```
si Pila[CPila]: CProg ← CProg + 1
si no: CProg ← direccion
CPila ← CPila-1
```

Operaciones aritméticas

mas

```
si S1 = 0: Pila[CPila - 1] ← Pila[CPila - 1] + Pila[CPila]
si S1 = 1: Pila[CPila - 1] ← Pila[CPila - 1] - Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

menos (binario)

```
si S1 = 0: Pila[CPila - 1] ← Pila[CPila - 1] - Pila[CPila]
si S1 = 1: Pila[CPila - 1] ← Pila[CPila - 1] + Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

mul

```
si S2 = 0: Pila[CPila - 1] ← Pila[CPila - 1] * Pila[CPila]
si S2 = 1: Pila[CPila - 1] ← Pila[CPila - 1] / Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

div

```
si S2 = 0: Pila[CPila - 1] ← Pila[CPila - 1] / Pila[CPila]
si S2 = 1: Pila[CPila - 1] ← Pila[CPila - 1] * Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

mod

```
Pila[CPila - 1] ← Pila[CPila - 1] % Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

menos (unario)

```
Pila[CPila] ← - Pila[CPila]
CProg ← CProg + 1
```

Operaciones de desplazamiento

lsh

```
Pila[CPila - 1] ← Pila[CPila - 1] << Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

rsh

> Pila[CPila - 1] ← Pila[CPila - 1] >> Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

Operaciones de comparación

igual

> Pila[CPila - 1] ← Pila[CPila - 1] == Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

noigual

> Pila[CPila - 1] ← Pila[CPila - 1] != Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

may

> Pila[CPila - 1] ← Pila[CPila - 1] > Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

men

> Pila[CPila - 1] ← Pila[CPila - 1] < Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

mayoig

> Pila[CPila - 1] ← Pila[CPila - 1] >= Pila[CPila]
> CPila ← CPila - 1
> CProg ← CPoprog + 1

menoig

> Pila[CPila - 1] ← Pila[CPila - 1] <= Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

Operaciones lógicas

and

> Pila[CPila - 1] ← Pila[CPila - 1] && Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

or

> Pila[CPila - 1] ← Pila[CPila - 1] || Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

not

> Pila[CPila] ← ! Pila[CPila]
> CProg ← CProg + 1

Operaciones de conversión

castFloat

> Pila[CPila] ← (float) Pila[CPila]
> CProg ← CProg + 1

castInt

> Pila[CPila] ← (int) Pila[CPila]
> CProg ← CProg + 1

castNat

> Pila[CPila] ← (nat) Pila[CPila]
> CProg ← CProg + 1

castChar

> Pila[CPila] ← (char) Pila[CPila]
> CProg ← CProg + 1

Operaciones de Entrada-Salida

in(type)

> CPila ← CPila + 1
> Pila[CPila] ← Leer un valor de tipo type de BufferIN
> CProg ← CProg + 1

out

> Escribir en BufferOUT ← Pila[CPila]
> CPila ← CPila - 1
> CProg ← CProg + 1

Operaciones de intercambio

swap1

> si S1 = 0: S1 ← 1
> si S1 = 1: S1 ← 0

swap2

> si S2 = 0: S2 ← 1
> si S2 = 1: S2 ← 0

Otras operaciones

range(size)

> si Pila[CPila] < 0 II Pila[CPila] >= size: P ← 1

stop

> P ← 1

Consideraciones sobre “Repertorio de instrucciones”

En la operación castNat, hemos creado la operación en la máquina virtual (nat), que no está predefinida en Java, pero cuyo comportamiento está definido en las tablas correspondientes a los tipos definidos.

5.2 Funciones semánticas

tamTipo(CTipo): dado un registro de tipo, devuelve el tamaño del tipo desplTupla(indice, CTipo): dado un registro de tipo y un indice, devuelve el offset hasta el indice (incluido) numCeldas(CTipo): Dado un tipo te devuelve el numero de celdas de memoria.

5.3 Atributos semánticos

- cod: Atributo sintetizado de generación de código.
- op: Enumerado que nos dice cuál es el operador utilizado.
- etq: Contador de instrucciones. Cuenta instucciones de la máquina a pila generadas.
- etqh: Contador de instrucciones heredado.
- refh: Atributo que indica si la expresión no tiene que generar el apila-ind para cargar el valor. Si la expresión es un parámetro por referencia refh vale true. Si no, vale false.

5.4 Gramática de atributos

```
Program → program ident illave SConsts STypes SVars SSubprogs SInsts fllave fin
    Program.cod =  ir_a(SSubprogs.etq) || SSubprogs || SInsts.cod || stop
    SSubprogs.etqh = 5 /* es 5 por inicializaciones de la pila. */
    SInsts.etqh = SSubprogs.etq

SSubprogs → subprograms illave Subprogs fllave
    SSubprogs.cod = Subprogs.cod
    Subprogs.etqh = SSubprogs.etqh
    SSubprogs.etq = Subprogs.etq

SSubprogs → subprograms illave fllave
    SSubprogs.cod = []
    SSubprogs.etq = SSubprogs.etqh

SSubprogs → ε
    SSubprogs.cod = []
    SSubprogs.etq = SSubprogs.etqh

Subprogs → Subprogs Subprog
    Subprogs0.cod  = Subprogs1.cod || Subprog.cod
    Subprogs1.etqh = Subprogs0.etqh
    Subprog.etqh   = Subprogs1.etq
    Subprogs0.etq  = Subprog.etq

Subprogs → Subprog
    Subprogs.cod = Subprog.cod
    Subprog.etqh = Subprogs.etqh
    Subprogs.etq = Subprog.etq

Subprog → subprogram ident ipar SFParams fpar illave SVars SInsts fllave
    Subprog.cod = apila-dir(0) || apila(SVars.dir) || mas || desapila-dir(0) ||
                SInsts.cod ||
                apila_dir(1) || apila(2) || menos || apila_ind || ir_ind
    SInsts.etqh = Subprog.etqh
    Subprog.etq = SInsts.etq + 5

SInsts → instructions illave Insts fllave
    SInsts.cod = Insts.cod
    Insts.etqh = SInsts.etqh
    SInsts.etq = Insts.etq

Insts → Insts pyc Inst
    Insts0.cod = Insts1.cod || Inst.cod
    Insts1.etqh = Insts0.etqh
    Inst.etqh = Insts1.etq
    Insts0.etq = Inst.etq
```

```
Insts → Inst
    Insts.cod = Inst.cod
    Inst.etqh = Insts.etqh
    Insts.etq = Inst.etq

Inst → Desig asig Expr
    Inst.cod = Expr.cod || Desig.cod || si esPrimitivo(Desig.tipo) entonces desapila-ind
                sino mueve(tamTipo(Desig.tipo,Desig.tsh))
    Expr.etqh = Inst.etqh
    Desig.etqh = Expr.etq
    Inst.etq = Desig.etq + 1
    Expr.refh = false

Inst → in ipar Desig fpar
    Inst.cod = in(Desig.type) ||Desig.cod|| desapila-ind
    Desig.etqh = Inst.etq + 1
    Inst.etq = Desig.etq + 1

Inst → out ipar Expr fpar
    Inst.cod = Expr.cod || out
    Expr.etqh = Inst.etqh
    Inst.etq = Expr.etqh + 1
    Expr.refh = false

Inst → swap1 ipar fpar
    Inst.cod = swap1
    Inst.etq = Inst.etqh + 1

Inst → swap2 ipar fpar
    Inst.cod = swap2
    Inst.etq = Inst.etqh +1

Inst → if Expr then Insts ElseIf
    Inst.cod = Expr.cod || ir_f(Insts.etq + 1) || Insts.cod || ir_a(Elseif.etq) || ElseIf.cod
    Expr.etqh = Inst.etqh
    Insts.etqh = Expr.etq + 1
    ElseIf.etqh = Insts.etq + 1
    Inst.etq = ElseIf.etq
    Expr.refh = false

Inst → while Expr do Insts endwhile
    Inst.cod = Expr.cod || ir_f(Insts.etq + 1) || Insts.cod || ir_a(Inst.etqh)
    Expr.etqh = Inst.etqh
    Insts.etqh = Expr.etq + 1
    Inst.etq = Insts + 1
    Expr.refh = false

Inst → InstCall
    Inst.cod = IsntCall.cod
    InstCall.etqh = Inst.etqh
    Inst.etq = InstCall.etq

Inst → ε
    Inst.cod = []
    Inst.etq = Inst.etqh

ElseIf → else Insts endif
    ElseIf.cod = Inst.cod
    Insts.etqh = ElseIf.etqh
    ElseIf.etq = Insts.etq

ElseIf → endif
    ElseIf.cod = []
    ElseIf.etq = ElseIf.etqh

InstCall → call ident lpar SRParams rpar
    InstCall.cod =
                //Reestructuramos los punteros CP y BASE
```

```
                apila-ret || apila-dir(0) || apila(1) || mas || desapila-ind || apiladir(1) || apila-dir(0) || apila(2)
                //Paso de parámetros
                SRParams.cod||
                // Saltar al subprograma
                apila-dir(0) || desapila-dir(1) || apila-dir(0) || apila(tamParametros(InstCall.tsh, ident)) || mas || d
                //Al volver del subprograma devolver los punteros CP y BASE a su sitio
                apila-dir(1) || apila(3) || menos || desapila-dir(0) || apila-dir(1) || apila(1) || menos || apila-ind |

    SRParams.nparams = 0
    SRParams.etqh = InstCall.etqh + 14
    InstCall.etq = SRParams.etq + 16

SRParams → RParams
    SRParams.cod = RParams.cod
    RParams.etqh = SRParams.etqh
    SRParams.etq = RParams.etq
    RParams.nparamsh = SRParams.nparamsh
    SRParams.nparams = RParams.nparams

SRParams → ε
    SRParams.cod = []
    SRParms.etq = SRParams.etqh
    SRParams.nparams = SRParams.nparamsh

RParams → RParams coma RParam
    RParams0.cod = RParams1.cod || RParam.cod
    RParams1.etqh = RParams0.etqh
    RParam.etqh = RParams1.etq
    RParams.etq = RParam.etq
    RParams1.nparamsh = RParams0.nparamsh
    RParam.nparamsh = RParams1.nparams
    RParams.nparams = RParam.nparams

RParams → RParam
    RParams.cod = RParam.cod
    RParam.etqh = RParams.etqh
    RParams.etq = RParam.etq
    RParam.nparamsh = RParams.nparamsh
    RParams.nparams = RParam.nparams

RParam → ident asig Expr
    RParam.cod = Expr.cod || apila_dir(0) || apila(RParams. nparams) || mas
                si (RParam.tsh[ident.lex].clase == pvariable)
                    || desapila-ind
                sino si (esPrimitivo(RParam.tsh[ident.lex].tipo)
                        || desapila-ind
                    sino // es un tipo compuesto
                        || mueve(tamTipo(RParam.tsh[ident.lex].tipo, Rparam.tsh))

    RParam.nparams = RParams.nparamsh + 1
    Expr.etqh = RParam.etqh
    RParam.etq = Expr.etq + 4
    Expr.refh = RParam.tsh[ident.lex] == pvariable

Desig → ident
    Desig.cod = si (Desig.tsh[ident.lex].nivel == global) entonces
                    apila(Desig.tsh[ident.lex].dir)
                    Desig.etq = Desig.etq + 1

                si no // el nivel el local
                    si (Desig.tsh[ident.lex].clase == var || Desig.tsh[ident.lex].clase == pvalor) entonces
                        apila_dir(1) || apila(Desig.tsh[ident.lex].dir) || mas
                        Desig.etq = Desig.etq + 3

                    si no si (Desig.tsh[ident.lex].clase == pvariable )
                        apila_dir(1) || apila(Desig.tsh[ident.lex].dir) || mas || apila_ind
                        Desig.etq = Desig.etq + 4

Desig → Desig icorchete Expr fcorchete
```

```
    Desig0.cod = Desig1.cod || Expr.cod || range(tamTipo(Desig1.type)) || apila(tamTipo(Desig1.type)) || mul || mas
    Desig1.etqh = Desig0.etqh
    Expr.etqh = Desig1.etq
    Desig0.etq = Expr.etq + 3
    Expr.refh = false

Desig → Desig barrabaja litnat
    Desig0.cod = Desig1.cod || apila(desplTupla(litnat.lex, Desig1.type)) || mas
    Desig1.etqh = Desig0.etqh
    Desig0.etq = Desgi1.etq + 2

Expr → Term Op0 Term
    Expr0.cod = Term1.cod || Term2.cod || Op0.op
    Term1.etqh = Expr.etqh
    Term2.etqh = Term1.etq
    Expr.etq = Term2.etq + 1
    Term0.refh = Expr.refh
    Term1.refh = Expr.refh

Expr → Term
    Expr.cod = Term.cod
    Term.etqh = Expr.etqh
    Expr.etq = Term.etq
    Term.refh = Expr.refh

Term → Term Op1 Fact
    Term0.cod = Term1.cod || Fact.cod || Op1.op
    Term1.etqh = Term0.etqh
    Fact.etqh = Term1.etq
    Term0.etq = Fact.etq + 1
    Term1.refh = Term0.refh
    Fact.refh = Term0.refh

Term → Term or Fact
    Term0.cod → Term1.cod || copia || ir-v(Fact.etq ) || desapila || Fact.cod
    Term1.etqh = Term0.etqh
    Fact.etqh = Term1.etq + 3
    Term0.etq = Fact.etq
    Expr.refh = false
    Term1.refh = Term0.refh
    Fact.refh = Term0.refh

Term → Fact
    Term.cod = Fact.cod
    Fact.etqh = Term.etqh
    Term.etq = Fact.etq
    Fact.refh = Term.refh

Fact → Fact Op2 Shft
    Fact0.cod = Fact1.cod || Shft.cod || Op2.op
    Fact1.etqh = Fact0.etqh
    Shft.etqh = Fact1.etq
    Term0.etq = Shft.etq + 1
    Fact1.refh = Fact0.refh
    Shft.refh = Fact0.refh

Fact → Fact and Shft
    Fact0.cod = Fact1.cod || copia || ir-f(Shft.etq ) || desapila || Shft.cod
    Fact1.etqh = = Fact0.etqh
    Shft.etqh = Fact1.etq + 3
    Fact0.etq = Shft.etq
    Fact1.refh = Fact0.refh
    Shft.refh = Fact0.refh

Fact → Shft
    Fact.cod = Shft.cod
    Shft.etqh = Fact.etqh
    Fact.etq = Shft.etq
    Shft.refh = Fact.refh
```

```
Shft → Unary Op3 Shft
    Shft0.cod = Unary.cod || Shft1.cod || Op3.op
    Unary.etqh = Shft0.etqh
    Shft1.etqh = Unary.etq
    Shft0.etq = Shft1.etq + 1
    Unary.refh = Shft0.refh
    Shft1.refh = Shft0.refh

Shft → Unary
    Shft.cod = Unary.cod
    Unary.etqh = Shft.eqth
    Shft.etq = Unary.etq
    Unary.refh = Shft.refh

Unary → Op4 Unary
    Unary0.cod = Unary1.cod || Op4.op
    Unary1.etqh = Unary0.eqth
    Unary0.eqt = Unary1.etq + 1
    Unary1.refh = Unary0.refh

Unary → lpar Cast rpar Paren
    Unary.cod = Paren.cod || Cast.type
    Paren.etqh = Unary.eqth
    Unary.etq = Paren.eqt + 1
    Paren.refh = Unary.refh

Unary → Paren
    Unary.cod = Paren.cod
    Paren.eqth = Unary.etqh
    Unary.etq = Paren.etq
    Paren.refh = Unary.refh

Paren → lpar Expr rpar
    Paren.cod = Expr.cod
    Expr.etqh = Paren.eqth
    Paren.etq = Expr.etq
    Expr.tsh = Paren.tsh

Paren → Lit
    Paren.cod = apila(Lit.valor)
    Paren.etq = Paren.etqh + 1

Paren → Desig
    Paren.cod = Desig.cod ||
                si (esPrimitivo(Desig.tipo) && Desig.tsh[Desig.lex].clase == constante)
                    apila(Desig.tsh[Desig.lex].valor)
                    Desig.etq = Desig.etq + 1
                fsi
                si (esPrimitivo(Desig.tipo) && !Paren.refh)
                    apila-ind
                    Desig.etq = Desig.etq + 1
                fsi
    Desig.etqh = Paren.etqh
    Paren.etq = Desig.etq + 1

Cast → char
    Cast.type = char
Cast → int
    Cast.type = int
Cast → nat
    Cast.type = nat
Cast → float
    Cast.type = float

Op0 → igual
    Op0.op = igual
Op0 → noigual
    Op0.op = noigual
```

```
Op0 → men
    Op0.op = men
Op0 → may
    Op0.op = may
Op0 → menoig
    Op0.op = menoig
Op0 → mayoig
    Op0.op = mayoig
Op1 → menos
    Op1.op = menos
Op1 → mas
    Op1.op = mas
Op2 → mod
    Op2.op = mod
Op2 → div
    Op2.op = div
Op2 → mul
    Op2.op = mul
Op3 → lsh
    Op3.op = lsh
Op3 → rsh
    Op3.op = rsh
Op4 → not
    Op4.op = not
Op4 → menos
    Op4.op = menos
```

10 Esquema de traducción para la construcción de grafos de dependencias

```
Program ::= PROGRAM IDENT ILLAVE SConsts STypes SVars SSubprogs SInsts FLLAVE
    {$$ = program_R1($4, $5, $6, $7, $8);}

SConsts ::= CONSTS ILLAVE Consts FLLAVE
    {$$ = sConsts_R1($3);}
SConsts ::=
    {$$ = sConsts_R1();}

Consts ::= Consts PYC Const
    {consts_R1($1, $3);}
Consts ::= Const
    {$$ = const_R2($1);}

Const ::= CONST TPrim IDENT ASIG ConstLit
    {$$ = const_R1($2, $3, $5);}
Const ::=
    {$$ = const_R2();}

ConstLit ::= Lit
    {$$ = constLit_R1($1);}
ConstLit ::= MENOS Lit
    {$$ = constLit_R1($2);}

STypes ::= TIPOS ILLAVE Types FLLAVE
    {$$ = sTypes_R1($3);}
STypes ::=
    {$$ = sTypes_R2();}

Types ::= Types PYC Type
    {$$ = types_R1($1, $3);}
Types ::= Type
    {$$ = types_R2($1);}

Type ::= TIPO TypeDesc IDENT
```

```
    {$$ = type_R1($2, $3);}
Type ::=
    {$$ = type_R2();}

SVars ::= VARS ILLAVE Vars FLLAVE
    {$$ = sVars_R1($3);}
SVars ::=
    {$$ = sVars_R2();}

Vars ::= Vars PYC Var
    {$$ = vars_R1($1, $3);}
Vars ::=
    {$$ = vars_R2($1);}

Var ::= VAR TypeDesc IDENT
    {$$ = var_R1($2, $3);}
Var ::=
    {$$ = var_R2();}

TypeDesc ::= TPrim
    {$$ = typeDesc_R1($1);}
TypeDesc ::= TArray
    {$$ = typeDesc_R2($1);}
TypeDesc ::= TTupla
    {$$ = typeDesc_R3($1);}
TypeDesc ::= IDENT
    {$$ = typeDesc_R4($1);}

TPrim ::= NATURAL
    {$$ = tPrim_R1();}
TPrim ::= INTEGER
    {$$ = tPrim_R2();}
TPrim ::= FLOAT
    {$$ = tPrim_R3();}
TPrim ::= BOOLEAN
    {$$ = tPrim_R4();}
TPrim ::= CHARACTER
    {$$ = tPrim_R5();}

Cast ::= CHAR
    {$$ = cast_R1();}
Cast ::= INT
    {$$ = cast_R2();}
Cast ::= NAT
    {$$ = cast_R3();}
Cast ::= FLOAT
    {$$ = cast_R4();}

TArray ::= TypeDesc ICORCHETE IDENT FCORCHETE
    {$$ = tArray_R1($1, $3);}
TArray ::= TypeDesc ICORCHETE LITNAT FCORCHETE
    {$$ = tArray_R2($1, $3);}

TTupla ::= IPAR Tupla FPAR
    {$$ = tTupla_R1($2);}
TTupla ::= IPAR FPAR
    {$$ = tTupla_R2();}

Tupla ::= TypeDesc COMA Tupla
    {$$ = tupla_R1($1, $3);}
Tupla ::= TypeDesc
    {$$ = tupla_R2($1);}

SInsts ::= INSTRUCTIONS ILLAVE Insts FLLAVE
    {$$ = sInsts_R1($3);}

Insts ::= Insts PYC Inst
    {$$ = insts_R1($1, $3);}
Insts ::= Inst
```

```
    {$$ = insts_R2($1);}

Inst ::= Desig ASIG Expr
    {$$ = inst_R1($1, $3, $2);}
Inst ::= IN PAR Desig FPAR
    {$$ = inst_R2($3);}
Inst ::= OUT IPAR Expr FPAR
    {$$ = inst_R3($3);}
Inst ::= SWAP1 IPAR FPAR
    {$$ = inst_R4();}
Inst ::= SWAP2 IPAR FPAR
    {$$ = inst_R5();}
Inst ::= IF Expr THEN Insts ElseIf
    {$$ = inst_R6($2, $4, $5);}
Inst ::= WHILE Expr DO Insts ENDWHILE
    {$$ = inst_R7($2, $4);}
Inst ::= InstCall
    {$$ = inst_R8($1);}
Inst ::=
    {$$ = inst_R9();}

ElseIf ::= ELSE Insts ENDIF
    {$$ = elseIf_R1($2);}
ElseIf ::= ENDIF
    {$$ = elseIf_R2();}

InstCall ::= CALL IDENT IPAR SRParams FPAR
    {$$ = instCall_R1($2, $4);}

SRParams ::= RParams
    {$$ = srParams_R1($1);}
SRParams ::=
    {$$ = srParams_R2();}

RParams ::= RParams COMA RParam
    {$$ = rParams_R1($1, $3);}
RParams ::= RParam
    {$$ = rParams_R2($1);}

RParam ::= IDENT ASIG Expr
    {$$ = rParam_R1($1, $3);}

SSubprogs ::= SUBPROGRAMS ILLAVE Subprogs FLLAVE
    {$$ = sSubprogs_R1($3);}
SSubprogs ::= SUBPROGRAMS ILLAVE FLLAVE
    {$$ = sSubprogs_R2();}
SSubprogs ::=
    {$$ = sSubprogs_R3();}

Subprogs ::= Subprogs Subprog
    {$$ = subprogs_R1($1, $2);}
Subprogs ::= Subprog
    {$$ = subprogs_R2($1);}

Subprog ::= SUBPROGRAM IDENT IPAR SFParams FPAR ILLAVE SVars SInsts FLLAVE
    {$$ = subprog_R1($2, $4, $7, $8);}

SFParams ::= FParams
    {$$ = sfParams_R1($1);}
SFParams ::=
    {$$ = sfParams_R2();}

FParams ::= FParams COMA FParam
    {$$ = fParams_R1($1, $3);}
FParams ::= FParam
    {$$ = fParams_R2($1);}

FParam ::= TypeDesc IDENT
    {$$ = fParam_R1($1, $2);}
```

```
FParam ::= TypeDesc MUL IDENT
    {$$ = fParam_R2($1, $3);}

Desig ::= IDENT
    {$$ = desig_R1($1)));}
Desig ::= Desig ICORCHETE Expr FCORCHETE
    {$$ = desig_R2($1, $3);}
Desig ::= Desig BARRABAJA LITNAT
    {$$ = desig_R3($1, $3);}

Expr ::= Term Op0 Term
    {$$ = expr_R1($1, $2, $3);}
Expr ::= Term
    {$$ = expr_R2($1);}

Term ::= Term Op1 Fact
    {$$ = term_R1($1, $2, $3);}
Term ::= Term OR Fact
    {$$ = term_R2($1, $3);}
Term ::= Fact
    {$$ = term_R3($1);}

Fact ::= Fact Op2 Shft
    {$$ = fact_R1($1, $2, $3);}
Fact ::= Fact AND Shft
    {$$ = fact_R2($1, $3);}
Fact ::= Shft
    {$$ = fact_R3($1);}

Shft ::= Unary Op3 Shft
    {$$ = shft_R1($1, $2, $3);}
Shft ::= Unary
    {$$ = shft_R2($1);}

Unary ::= Op4 Unary
    {$$ = unary_R1($1, $2);}
Unary ::= IPAR Cast FPAR Paren
    {$$ = unary_R2($2, $4);}
Unary ::= Paren
    {$$ = unary_R3($1);}

Paren ::= IPAR Expr FPAR
    {$$ = paren_R1($2);}
Paren ::= Lit
    {$$ = paren_R2($1);}
Paren ::= Desig
    {$$ = paren_R3($1);}

Op0 ::= IGUAL
    {$$ = op0_R1();}
Op0 ::= NOIGUAL
    {$$ = op0_R2();}
Op0 ::= MEN
    {$$ = op0_R3();}
Op0 ::= MAY
    {$$ = op0_R4();}
Op0 ::= MENOIG
    {$$ = op0_R5();}
Op0 ::= MAYOIG
    {$$ = op0_R6();}

Op1 ::= MENOS
    {$$ = op1_R1();}
Op1 ::= MAS
    {$$ = op1_R2();}

Op2 ::= MOD
    {$$ = op2_R1();}
Op2 ::= DIV
```

```
    {$$ = op2_R2();}
Op2 ::= MUL
    {$$ = op2_R3();}

Op3 ::= LSH
    {$$ = op3_R1();}
Op3 ::= RSH
    {$$ = op3_R2();}

Op4 ::= NOT
    {$$ = op4_R1();}
Op4 ::= MENOS
    {$$ = op4_R2();}

Lit ::= LitBool
    {$$ = lit_R1($1);}
Lit ::= LitNum
    {$$ = lit_R2($1);}
Lit ::= LITCHAR
    {$$ = lit_R3($1));}

LitBool ::= TRUE
    {$$ = litBool_R1();}
LitBool ::= FALSE
    {$$ = litBool_R2();}

LitNum ::= LITNAT
    {$$ = litNum_R1($1);}
LitNum ::= LITFLOAT
    {$$ = litNum_R2($1);}
```

11 Descripción de las funciones de atribución

```
funcion program_R1 (SConsts, STypes, SVars, SSubprogs, SInsts) {
    Program.tsh = creaTS()
    Program.dirh = 2
    SConsts.tsh = Program.tsh
    STypes.tsh = SConsts.ts
    SVars.tsh = STypes.ts
    SVars.dirh = SProgram.dirh
    SSubprogs.tsh = SVars.ts
    Program.err = SConsts.err ∨ STypes.err ∨ SVars.err ∨ SSubprogs.err ∨ SInsts.err
    SInsts.tsh = SSubprogs.ts
    Program.cod =  ir_a(SSubprogs.etq) || SSubprogs || SInsts.cod || stop
    SSubprogs.etqh = 5
    SInsts.etqh = SSubprogs.etq
    SVars.nivelh = global

    return Program
}

funcion sConsts_R1 (Consts) {
    Consts.tsh = SConsts.tsh
    SConsts.ts = Consts.ts
    SConsts.err = Consts.err

    return SConsts
}

funcion sConsts_R2 () {
    SConsts.ts = SConsts.tsh
    SConsts.err = false

    return SConsts
}
```

```
funcion consts_R1 (Consts1, Const) {
    Consts1.tsh = Consts0.tsh
    Const.tsh = Consts1.ts
    Consts0.ts = añade(Const.ts, Const.id, Const.clase, Const.nivel, ?, Const.tipo, Const.valor)
    Consts.err = existe(Const.ts, Const.id)

    return Consts0
}

funcion consts_R2 (Const) {
    Const.tsh = Consts.tsh
    Consts.ts = añade(Const.ts, Const.id, Const.clase, Const.nivel, ?, Const.tipo, Const.valor)
    Consts.err = existe(Const.ts, Const.id)

    return Const
}

funcion const_R1 (TPrim, ident, ConstLit) {
    Const.ts = Const.tsh
    Const.id = ident.lex
    Const.clase = const
    Const.nivel = global
    Const.tipo = <t:TPrim.tipo, tam:1>
    Const.valor = ConstLit.valor
    Const.err = ¬(compatibles(TPrim.tipo, ConstLit.tipo))

    return Const
}

funcion const_R2 () {
    Const.ts = Const.tsh
    Const.err = false

    return Const
}

funcion constLit_R1 (Lit) {
    ConstLit.valor = Lit.valor
    ConstLit.tipo = Lit.tipo

    return ConstLit
}

funcion constLit_R2 (Lit) {
    ConstLit.valor = -(Lit.valor)
    ConstLit.tipo = opUnario(menos, Lit.tipo)

    return ConstLit
}

funcion sTypes_R1 (Types) {
    Types.tsh = STypes.tsh
    STypes.ts = Types.ts
    STypes.err = Types.err

    return STypes
}

funcion sTypes_R2 () {
    STypes.ts = STypes.tsh
    STypes.err = false

    return STypes
}

funcion types_R1 (Types1, Type) {
    Types1.tsh = Types0.tsh
    Type.tsh = Types1.ts
    Types0.ts = añade(Types1.ts, Type.id, Type.clase, Type.nivel, ?, Type.tipo)
```

```
    Types0.err = existe(Types1.ts, Type.id)

    return Types0
}

funcion types_R2 (Type) {
    Type.tsh = Types.tsh
    Types.ts = añade(Type.ts, Type.id, Type.clase, Type.nivel, ?, Type.tipo)
    Types.err = existe(Type.ts, Type.id)

    return Types
}

funcion type_R1 (TypeDesc, ident) {
    Type.ts = Type.tsh
    TypeDesc.tsh = Type.tsh
    Type.id = ident.lex
    Type.clase = Tipo
    Type.nivel = global
    Type.tipo = <t:TypeDesc.tipo, tipo:obtieneCTipo(TypeDesc), tam:desplazamiento(TypeDesc.tipo, Var.tsh ), Type.id)>

    return Type
}

funcion type_R2 () {
    Type.ts = Type.tsh
    Type.err = false

    return Type
}

funcion sVars_R1 (Vars) {
    Vars.tsh = SVars.tsh
    Vars.dirh = SVars.dirh
    SVars.ts = Vars.ts
    SVars.dir = Vars.dir
    SVars.err = Vars.err
    Vars.nivelh = SVars.nivelh

    return SVars
}

funcion sVars_R2 () {
    SVars.ts = SVars.tsh
    SVars.dir = SVars.dirh
    SVars.err = false

    return SVars
}

funcion vars_R1 (Vars1, Var) {
    Vars1.tsh = Vars0.tsh
    Vars1.dirh = Vars0.dirh
    Var.tsh = Vars1.ts
    Var.dirh = Vars1.dir
    Vars0.dir = Var.dir + desplazamiento(Var.tipo, Vars1.id)
    Vars0.ts = añade(Var.ts, Var.id, Var.clase, Var.nivel, Vars0.dir, Var.tipo)
    Vars0.err = existe(Var.ts, Var.id, Var.nivel)
    Vars1.nivelh = Vars0.nivelh
    Var.nivelh = Vars0.nivelh

    return Vars0
}

funcion vars_R2 (Var) {
    Var.tsh = Vars.tsh
    Var.dirh = Vars.dirh
    Vars.dir = Var.dir + desplazamiento(Var.tipo, Var.id)
    Vars.ts = añade(Var.ts, Var.id, Var.clase, Var.nivel, Var.dir, Var.tipo)
```

```
    Vars.err = existe(Var.ts, Var.id, Var.nivel)
    Var.nivelh = Vars.nivelh

    return Vars
}

funcion var_R1 (TypeDesc, ident) {
    Var.ts = Var.tsh
    Var.dir = Var.dirh
    Var.id = ident.lex
    Var.clase = Var
    Var.nivel = nivelh
    Var.tipo = si (TypeDesc.tipo == TPrim) {<t:TypeDesc.tipo, tam:1>}
           si no {<id:Var.id, t:ref, TypeDesc.tipo tam: desplazamiento(TypeDesc.tipo, Var.tsh )>}
    TypeDesc.tsh = Var.tsh

    return Var
}

funcion var_R2 () {
    Var.ts = Var.tsh
    Var.dir = Var.dirh
    Var.err = false

    return Var
}

funcion typeDesc_R1 (TPrim) {
    TypeDesc.tipo = TPrim.tipo

    return TypeDesc
}

funcion typeDesc_R2 (TArray) {
    TypeDesc.tipo = TArray.tipo
    TArray.tsh = TypeDesc.tsh
    TypeDesc.err = TArray.err

    return TypeDesc
}

funcion typeDesc_R3 (TTupla) {
    TypeDesc.tipo = TTupla.tipo
    TTupla.tsh = TypeDesc.tsh
    TypeDesc.err = TTupla.err

    TypeDesc
}

funcion typeDesc_R4 (ident) {
    TypeDesc.tipo = ident.lex
    TypeDesc.err = ¬existe(TypeDesc.tsh, ident.lex) ∨ TypeDesc.tsh[ident].clase != tipo

    return TypeDesc
}

funcion tPrim_R1 () {
    TPrim.tipo = natural

    return TPrim
}

funcion tPrim_R2 () {
    TPrim.tipo = integer

    return TPrim
}

funcion tPrim_R3 () {
```

```
    TPrim.tipo = float

    return TPrim
}

funcion tPrim_R4 () {
    TPrim.tipo = boolean

    return TPrim
}

funcion tPrim_R5 () {
    TPrim.tipo = character

    return TPrim
}

funcion cast_R1 () {
    Cast.type = char

    return Cast
}

funcion cast_R2 () {
    Cast.type = int

    return Cast
}

funcion cast_R3 () {
    Cast.type = nat

    return Cast
}

funcion cast_R4 () {
    Cast.type = float

    return Cast
}

funcion tArray_R1 (TypeDesc, ident) {
    TypeDesc.tsh = TArray.tsh
    TArray.tsh = TypeDesc.tsh
    TArray.err = ¬existe(TArray.tsh, ident.lex) ∨ obtieneTipoString(ident) != nat ∨ TArray.tsh[ident].clase != constante

    return TArray
}

funcion tArray_R2 (TypeDesc, litnat) {
    TypeDesc.tsh = TArray.tsh
    TArray.tsh = TypeDesc.tsh

    return TArray
}

funcion tTupla_R1 (Tupla) {
    Tupla.tsh = TTupla.tsh
    TTupla.tipo = Tupla.tipo
    TTupla.err = Tupla.err

    return TTupla
}

funcion tTupla_R2 () {
    TTupla.err = false

    return TTupla
}
```

```
funcion tupla_R1 (TypeDesc, Tupla1) {
    TypeDesc.tsh = Tupla0.tsh
    Tupla1.tsh = Tupla0.tsh
    Tupla0.tipo = TypeDesc.tipo ++ Tupla1.tipo
    Tupla0.err = TypeDesc.err ∨ Tupla1.err

    return Tupla0
}

funcion tupla_R2 (TypeDesc) {
    TypeDesc.tsh = Tupla.tsh
    Tupla.tipo = TypeDesc.tipo
    Tupla.err = TypeDesc.err

    return Tupla
}

funcion sInsts_R1 (Insts) {
    Insts.tsh = SInsts.tsh
    SInsts.err = Insts.err
    SInsts.cod = Insts.cod
    Insts.etqh = SInsts.etqh
    SInsts.etq = Insts.etq

    return SInsts
}

funcion insts_R1 (Insts1, Inst) {
    Insts1.tsh = Insts0.tsh
    Inst.tsh = Insts0.tsh
    Insts0.err = Insts1.err ∨ Inst.err
    Insts0.cod = Insts1.cod || Inst.cod
    Insts1.etqh = Insts0.etqh
    Inst.etqh = Insts1.etq
    Insts0.etq = Inst.etq

    return Insts0
}

funcion insts_R2 (Inst) {
    Inst.tsh = Insts.tsh
    Insts.err = Inst.err
    Insts.cod = Inst.cod
    Inst.etqh = Insts.etqh
    Insts.etq = Inst.etq

    return Insts
}

funcion inst_R1 (Desig, Expr) {
    Desig.tsh = Inst.tsh
    Expr.tsh = Inst.tsh
    Inst.err = (¬asignacionValida(Desig.tipo, Expr.tipo)) ∨ Expr.err ∨ Desig.err
    Inst.cod = Expr.cod || Desig.cod || si esPrimitivo(Desig.tipo) entonces desapila-ind
                sino mueve(tamTipo(Desig.tipo,Desig.tsh))
    Expr.etqh = Inst.etqh
    Desig.etqh = Expr.etq
    Inst.etq = Desig.etq + 1
    Expr.refh = false

    return Inst
}

funcion inst_R2 (Desig) {
    Desig.tsh = Inst.tsh
    Inst.err = Desig.err
    Inst.cod = in(Desig.type) ||Desig.cod|| desapila-ind
    Desig.etqh = Inst.etq + 1
```

```
    Inst.etq = Desig.etq + 1

    return Inst
}

funcion inst_R3 (Expr) {
    Expr.tsh = Inst.tsh
    Inst.err = Expr.err
    Inst.cod = Expr.cod || out
    Expr.etqh = Inst.etqh
    Inst.etq = Expr.etqh + 1
    Expr.refh = false

    return Inst
}

funcion inst_R4 () {
    Inst.err = false
    Inst.cod = swap1
    Inst.etq = Inst.etqh + 1

    return Inst
}

funcion inst_R5 () {
    Inst.err = false
    Inst.cod = swap2
    Inst.etq = Inst.etqh + 1

    return Inst
}

funcion inst_R6 (Expr, Insts, ElseIf) {
    Expr.tsh = Inst.tsh
    Insts1.tsh = Inst0.tsh
    ElseIf.tsh = Inst.tsh
    Inst.err = Expr.err ∨ Insts.err ∨ ElseIf.err
    Inst.cod = Expr.cod || ir_f(Insts.etq + 1) || Insts.cod || ir_a(Elseif.etq) || ElseIf.cod
    Expr.etqh = Inst.etqh
    Insts.etqh = Expr.etq + 1
    ElseIf.etqh = Insts.etq + 1
    Inst.etq = ElseIf.etq
    Expr.refh = false

    return Inst
}

funcion inst_R7 (Expr, Insts) {
    Expr.tsh = Inst.tsh
    Insts.tsh = Inst.tsh
    Inst.err = Expr.err ∨ Insts.err
    Inst.cod = Expr.cod || ir_f(Insts.etq + 1) || Insts.cod || ir_a(Inst.etqh)
    Expr.etqh = Inst.etqh
    Insts.etqh = Expr.etq + 1
    Inst.etq = Insts + 1
    Expr.refh = false

    return Inst
}

funcion inst_R8 (InstCall) {
    InstCall.tsh = Inst.tsh
    Inst.err = InstCall.err
    Inst.cod = IsntCall.cod
    InstCall.etqh = Inst.etqh
    Inst.etq = InstCall.etq

    return Inst
}
```

```
funcion inst_R9 () {
    Inst.err = false
    Inst.cod = []
    Inst.etq = Inst.etqh

    return Inst
}

funcion elseIf_R1 (Insts) {
    Insts.tsh = ElseIf.tsh
    ElseIf.err = Insts.err
    ElseIf.cod = Inst.cod
    Insts.etqh = ElseIf.etqh
    ElseIf.etq = Insts.etq

    return ElseIf
}

funcion elseIf_R2 () {
    ElseIf.err = false
    ElseIf.cod = []
    ElseIf.etq = ElseIf.etqh

    return ElseIf
}

funcion instCall_R1 (ident, SRParams) {
    SRParams.tsh = InstCall.tsh
    SRParams.nparams = 0
    SRParams.nombresubprogh = ident.lex
    SRParmas.listaparamnombresh = []
    InstCall.err = SRParams.err ∨ ¬existe(SRParams.tsh, ident.lex) ∨ SRParams.nparams != numParametros(SRParams.tsh, ide
    InstCall.cod =
                //Reestructuramos los punteros CP y BASE
                apila-ret || apila-dir(0) || apila(1) || mas || desapila-ind || apiladir(1) || apila-dir(0) || apila(2)
                //Paso de parámetros
                SRParams.cod||
                // Saltar al subprograma
                apila-dir(0) || desapila-dir(1) || apila-dir(0) || apila(tamParametros(InstCall.tsh, ident)) || mas || d
                //Al volver del subprograma devolver los punteros CP y BASE a su sitio
                apila-dir(1) || apila(3) || menos || desapila-dir(0) || apila-dir(1) || apila(1) || menos || apila-ind |

    SRParams.nparams = 0
    SRParams.etqh = InstCall.etqh + 14
    InstCall.etq = SRParams.etq + 16

    return InstCall
}

funcion srParams_R1 (RParams) {
    RParams.tsh = SRParams.tsh
    RParams.nombresubprogh = SRParams.nombresubprogh
    RParams.listaparamnombresh = SRParams.listaparamnombresh
    SRParams.err = RParams.err
    SRParams.cod = RParams.cod
    RParams.etqh = SRParams.etqh
    SRParams.etq = RParams.etq
    RParams.nparamsh = SRParams.nparamsh
    SRParams.nparams = RParams.nparams

    return SRParams
}

funcion srParams_R2 () {
    SRParams.err = false
    SRParams.nparams = SRParams.nparamsh
    SRParams.listaparamnombres = SRParams.listaparamnombresh
    SRParams.cod = []
```

```
    SRParms.etq = SRParams.etqh
    SRParams.nparams = SRParams.nparamsh

    return SRParams
}

funcion rParams_R1 (RParams1, RParam) {
    RParams1.tsh = RParams0.tsh
    RParam.tsh = RParams0.tsh
    RParams0.err = RParams1.err ∨ Rparam.err
    RParams1.nombresubprogh = RParams0.nombresubprogh
    RParam.nombresubprogh = RParams0.nombresubprogh
    RParams1.listaparamnombresh = RParams0.listaparamnombresh
    RParam.listaparamnombresh = RParams1.listaparamnombres
    RParams0.cod = RParams1.cod || RParam.cod
    RParams1.etqh = RParams0.etqh
    RParam.etqh = RParams1.etq
    RParams.etq = RParam.etq
    RParams1.nparamsh = RParams0.nparamsh
    RParam.nparamsh = RParams1.nparams
    RParams.nparams = RParam.nparams

    return RParams0
}

funcion rParams_R2 (RParam) {
    RParam.tsh = RParams.tsh
    RParam.nombresubprogh = RParams.nombresubprogh
    RParam.listaparamnombresh = RParams.listaparamnombresh
    RParams.listaparamnombres = RParam.listaparamnombres
    RParams.err = RParam.err
    RParams.cod = RParam.cod
    RParam.etqh = RParams.etqh
    RParams.etq = RParam.etq
    RParam.nparamsh = RParams.nparamsh
    RParams.nparams = RParam.nparams

    return RParams
}

funcion rParam_R1 (ident, Expr) {
    Expr.tsh = RParam.tsh
    RParam.listaparamnombres = RParam.listaparamnombresh ++ ident
    RParam.err = Expr.err ∨ ¬existe(Exp.tsh, ident.lex) ∨ ¬esVariable(Expr.tsh, ident.lex)
    ∨ ¬estaDeclarado(RParam.tsh, ident.lex, RParam.nombresubprogh) ∨ ¬compatible(ident.tipo,Expr.tipo) ∨ ¬Expr.desig ∨ (
    RParam.cod = Expr.cod || apila_dir(0) || apila(RParams. nparams) || mas
                si (RParam.tsh[ident.lex].clase == pvariable)
                    || desapila-ind
                sino si (esPrimitivo(RParam.tsh[ident.lex].tipo)
                        || desapila-ind
                    sino // es un tipo compuesto
                        || mueve(tamTipo(RParam.tsh[ident.lex].tipo, Rparam.tsh))
    RParam.nparams = RParams.nparamsh + 1
    Expr.etqh = RParam.etqh
    RParam.etq = Expr.etq + 4
    Expr.refh = RParam.tsh[ident.lex] == pvariable

    return RParam
}

funcion sSubprogs_R1 (Subprogs) {
    Subprogs.tsh = SSubprogs.tsh
    SSbprogs.ts = Subprog.ts
    SSubprogs.err = Subprogrs.err
    SSubprogs.cod = Subprogs.cod
    Subprogs.etqh = SSubprogs.etqh
    SSubprogs.etq = Subprogs.etq

    return SSubprogs
```

```
}

funcion sSubprogs_R2 () {
    SSubprogs.tsh = Subprog.tsh
    SSubprogs.cod = []
    SSubprogs.etq = SSubprogs.etqh

    return SSubprogs
}

funcion sSubprogs_R3 () {
    SSubprogs.tsh = Subprog.tsh
    SSubprogs.err = false
    SSubprogs.cod = []
    SSubprogs.etq = SSubprogs.etqh

    return SSubprogs
}

funcion subprogs_R1 (Subprogs1, Subprog) {
    Subprogs1.tsh =  Subprogs0.tsh
    Subprog.tsh = Subprogs0.tsh
    Subprogs0.ts = Subprog.ts
    Subprogs0.err = Subprogs1.err ∨ Subprog.err
    Subprogs0.cod  = Subprogs1.cod || Subprog.cod
    Subprogs1.etqh = Subprogs0.etqh
    Subprog.etqh   = Subprogs1.etq
    Subprogs0.etq  = Subprog.etq

    return Subprogs0
}

funcion subprogs_R2 (Subprog) {
    Subprog.tsh = Subprogs.tsh
    Subprogs.ts = Subprog.ts
    Subprogs.err = Subprog.err
    Subprogs.cod = Subprog.cod
    Subprog.etqh = Subprogs.etqh
    Subprogs.etq = Subprog.etq

    return Subprogs
}

funcion subprog_R1 (ident, SFParams, SVars, SInsts) {
    SFParams.dirh = 0
    SFParams.tsh = CreaTS(Subprog.tsh)
    SVars.tsh = SFParams.ts
    SVars.dirh = SFParams.dir
    SInsts.tsh = SVars.ts
    Subprog.ts = añade(Subprog.tsh, ident, subprog, global, ? , <dir:Subprog.etqh, params:SFParams.params>)
    Subprog.err = existe(Subprog.tsh, ident) ∨ SParams.err ∨ SVars.err ∨ SInsts.err ∨ parametrosNoRepetidos(SParams.ts,
    Subprog.cod = apila-dir(0) || apila(SVars.dir) || mas || desapila-dir(0) ||
                SInsts.cod ||
                apila_dir(1) || apila(2) || menos || apila_ind || ir_ind
    SInsts.etqh = Subprog.etqh
    Subprog.etq = SInsts.etq + 5
    SVars.nivelh = local

    return Subprog
}

funcion sfParams_R1 (FParams) {
    FParams.tsh = SFParams.tsh
    SFParams.ts = FParams.ts
    FParams.dirh = SFParams.dirh
    SFParams.dir = FParams.dir
    SFParams.params = FParams.params
    SFParams.err = FParams.err
```

```
    return SFParams
}

funcion sfParams_R2 () {
    SFParams.ts = SFParams.tsh
    SFParams.dir = SFParams.dirh
    SFParams.params = []
    SFParams.err = false

    return SFParams
}

funcion fParams_R1 (FParams, FParam) {
    FParams1.tsh = FParams0.tsh
    FParams1.dirh = FParams0.dirh
    FParam.tsh = FParams1.tsh
    FParam.dirh = FParams1.dirh
    FParams0.dir = FParam.dir + desplazamiento(FParam.tipo, FParam.id)
    FParams0.ts = añade(FParam.ts, FParam.id, FParam.clase, FParam.nivel, FParam.dir, FParam.tipo)
    FParams0.params = FParams1.params ++ FParam.params
    FParams0.err = existe(FParam.ts, FParam.id, FParam.nivel)

    return FParams
}

funcion fParams_R2 (FParam) {
    FParam.dirh = FParams.dirh
    FParam.tsh = FParams.tsh
    FParams.ts = añade(FParam.ts, FParam.id, FParam.clase, FParam.nivel, FParam.dir, FParam.tipo)
    FParams.dir = FParam.dir + desplazamiento(FParam.tipo, FParam.id)
    FParams.params = FParap.params
    FParams.err = existe(FParam.ts, FParam.id, FParam.nivel)

    return FParams
}

funcion fParam_R1 (TypeDesc) {
    FParam.ts = FParam.tsh
    FParam.dir = FParam.dirh
    Fparam.id = ident.lex
    FParam.clase = pvalor
    FParam.nivel = local
    FParam.tipo = (si (TypeDesc.tipo== TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<t:ref, id:FParam.id, tam: desplazamiento(TypeDesc.tipo, Param.id)>} )
    FParam.params = [<id:FParam.id, tam:desplazamiento(TypeDesc.tipo, Param.id), ref:falso, despl:DParam.dirh>]
    TypeDesc.tsh = FParam.tsh

    return FParam
}

funcion fParam_R2 (TypeDesc, ident) {
    FParam.ts = FParam.tsh
    FParam.dir =  FParam.dirh
    Fparam.id = ident.lex
    FParam.clase = pvariable
    FParam.nivel = local
    FParam.tipo = (si (TypeDesc.tipo == TPrim) {<t:TypeDesc.tipo, tam:1>}
               si no {<t:ref, id:FParam.id, tam: 1>} )
    FParam.params = [<id:FParam.id, tam:desplazamiento(TypeDesc.tipo, Param.id), ref:cierto, despl:DParam.dirh>]
    TypeDesc.tsh = FParam.tsh

    return FParam
}

funcion desig_R1 (ident) {
    Desig.tipo = Desig.tsh[ident.lex].tipo
    Desig.err = ¬existe(Desig.tsh, ident.lex) ∨ ¬esVariable(Desig.tsh, ident.lex)
    Desig.cod = si (Desig.tsh[ident.lex].nivel == global) entonces
                    apila(Desig.tsh[ident.lex].dir)
```

```
                    Desig.etq = Desig.etq + 1

                si no // el nivel el local
                    si (Desig.tsh[ident.lex].clase == var || Desig.tsh[ident.lex].clase == pvalor) entonces
                        apila_dir(1) || apila(Desig.tsh[ident.lex].dir) || mas
                        Desig.etq = Desig.etq + 3

                    si no si (Desig.tsh[ident.lex].clase == pvariable )
                        apila_dir(1) || apila(Desig.tsh[ident.lex].dir) || mas || apila_ind
                        Desig.etq = Desig.etq + 4

    return Desig
}

funcion desig_R2 (Desig1, Expr) {
    Desig0.tipo = Desig1.tipo
    Desig0.err = Desig1.err ∨ Expr.err ∨ ¬tamañoCorrecto()
    Desig0.cod = Desig1.cod || Expr.cod || range(tamTipo(Desig1.type)) || apila(tamTipo(Desig1.type)) || mul || mas
    Desig1.etqh = Desig0.etqh
    Expr.etqh = Desig1.etq
    Desig0.etq = Expr.etq + 3
    Expr.refh = false

    return Desig0
}

funcion desig_R3 (Desig, litnat) {
    Desig0.tipo = Desig1.tipo
    Desig0.err = Desig1.err ∨ ¬tamañoCorrecto()
    Desig0.cod = Desig1.cod || apila(desplTupla(litnat.lex, Desig1.type)) || mas
    Desig1.etqh = Desig0.etqh
    Desig0.etq = Desgi1.etq + 2

    return Desig0
}

funcion expr_R1 (Term0, Op0, Term1) {
    Expr.desig = false
    Expr.tipo = tipoFunc(Term0.tipo, Op0.op, Term1.tipo)
    Term0.tsh = Expr.tsh
    Term1.tsh = Expr.tsh
    Expr.desig = false
    Expr0.cod = Term1.cod || Term2.cod || Op0.op
    Term1.etqh = Expr.etqh
    Term2.etqh = Term1.etq
    Expr.etq = Term2.etq + 1
    Term0.refh = Expr.refh
    Term1.refh = Expr.refh

    return Expr
}

funcion expr_R2 (Term) {
    Expr.tipo = Term.tipo
    Term.tsh = Expr.tsh
    Expr.desig = false
    Expr.desig = Term.desig
    Expr.cod = Term.cod
    Term.etqh = Expr.etqh
    Expr.etq = Term.etq
    Term.refh = Expr.refh

    return Expr
}

funcion term_R1 (Term1, Op1, Fact) {
    Term0.tipo = tipoFunc(Term1.tipo, Op1.op, Fact.tipo)
    Term1.tsh = Term0.tsh
    Fact.tsh = Term0.tsh
```

```
    Term0.desig = false
    Term0.cod = Term1.cod || Fact.cod || Op1.op
    Term1.etqh = Term0.etqh
    Fact.etqh = Term1.etq
    Term0.etq = Fact.etq + 1
    Term1.refh = Term0.refh
    Fact.refh = Term0.refh

    return Term0
}

funcion term_R2 (Term1, Fact) {
    Term0.tipo = tipoFunc(Term1.tipo, or, Fact.tipo)
    Term1.tsh = Term0.tsh
    Fact.tsh = Term0.tsh
    Term0.desig = false
    Term0.cod → Term1.cod || copia || ir-v(Fact.etq ) || desapila || Fact.cod
    Term1.etqh = Term0.etqh
    Fact.etqh = Term1.etq + 3
    Term0.etq = Fact.etq
    Expr.refh = false
    Term1.refh = Term0.refh
    Fact.refh = Term0.refh

    return Term0
}

funcion term_R3 (Fact) {
    Term.tipo = Fact.tipo
    Fact.tsh = Term.tsh
    Term.desig = Fact.desig
    Term.cod = Fact.cod
    Fact.etqh = Term.etqh
    Term.etq = Fact.etq
    Fact.refh = Term.refh

    return Term
}

funcion fact_R1 (Fact1, Op2, Shft) {
    Fact0.tipo = tipoFunc(Fact1.tipo, Op2.op, Shft.tipo)
    Fact1.tsh = Fact0.tsh
    Shft.tsh = Fact0.tsh
    Fact0.desig = false
    Fact0.cod = Fact1.cod || Shft.cod || Op2.op
    Fact1.etqh = Fact0.etqh
    Shft.etqh = Fact1.etq
    Term0.etq = Shft.etq + 1
    Fact1.refh = Fact0.refh
    Shft.refh = Fact0.refh

    return Fact0
}

funcion fact_R2 (Fact1, Shft) {
    Fact0.tipo = tipoFunc(Fact1.tipo, and, Shft.tipo)
    Fact1.tsh = Fact0.tsh
    Shft.tsh = Fact0.tsh
    Fact0.desig = false
    Fact0.cod = Fact1.cod || copia || ir-f(Shft.etq ) || desapila || Shft.cod
    Fact1.etqh = = Fact0.etqh
    Shft.etqh = Fact1.etq + 3
    Fact0.etq = Shft.etq
    Fact1.refh = Fact0.refh
    Shft.refh = Fact0.refh

    return Fact0
}
```

```
funcion fact_R3 (Shft) {
    Fact.tipo = Shft.tipo
    Shft.tsh = Fact.tsh
    Fact.desig = Shft.desig
    Fact.cod = Shft.cod
    Shft.etqh = Fact.etqh
    Fact.etq = Shft.etq
    Shft.refh = Fact.refh

    return Fact
}

funcion shft_R1 (Unary, Op3, Shft1) {
    Shft0.tipo = tipoFunc(Unary.tipo, Op3.op, Shft.tipo)
    Unary.tsh = Shft0.tsh
    Shft1.tsh = Shft0.tsh
    Shft0.desig = false
    Shft0.cod = Unary.cod || Shft1.cod || Op3.op
    Unary.etqh = Shft0.etqh
    Shft1.etqh = Unary.etq
    Shft0.etq = Shft1.etq + 1
    Unary.refh = Shft0.refh
    Shft1.refh = Shft0.refh

    return Shft0
}

funcion shft_R2 (Unary) {
    Shft.tipo = Unary.tipo
    Unary.tsh = Shft.tsh
    Shft.desig = Unary.desig
    Shft.cod = Unary.cod
    Unary.etqh = Shft.eqth
    Shft.etq = Unary.etq
    Unary.refh = Shft.refh

    return Shft
}

funcion unary_R1 (Op4, Unary1) {
    Unary0.tipo = opUnario(Op4.op, Unary1.tipo)
    Unary1.tsh = Unary0.tsh
    Unary0.desig = false
    Unary0.cod = Unary1.cod || Op4.op
    Unary1.etqh = Unary0.eqth
    Unary0.eqt = Unary1.etq + 1
    Unary1.refh = Unary0.refh

    return Unary0
}

funcion unary_R2 (Cast, Paren) {
    Unary.tipo = casting(Cast.tipo, Paren.tipo)
    Paren.tsh = Unary.tsh
    Unary.desig = false
    Unary.cod = Paren.cod || Cast.type
    Paren.etqh = Unary.eqth
    Unary.etq = Paren.eqt + 1
    Paren.refh = Unary.refh

    return Unary
}

funcion unary_R3 (Paren) {
    Unary.tipo = Paren.tipo
    Paren.tsh = Unary.tsh
    Unary.desig = Paren.desig
    Unary.cod = Paren.cod
    Paren.eqth = Unary.etqh
```

```
    Unary.etq = Paren.etq
    Paren.refh = Unary.refh

    return Unary
}

funcion paren_R1 (Expr) {
    Paren.tipo = Expr.tipo
    Expr.tsh = Paren.tsh
    Paren.desig = false
    Paren.cod = Expr.cod
    Expr.etqh = Paren.eqth
    Paren.etq = Expr.etq
    Expr.tsh = Paren.tsh

    return Expr
}

funcion paren_R2 (Lit) {
    Parent.tipo = Lit.tipo
    Lit.tsh = Paren.tsh
    Paren.desig = false
    Paren.err = false
    Paren.cod = apila(Lit.valor)
    Paren.etq = Paren.etqh + 1

    return Paren
}

funcion paren_R3 (Desig) {
    Paren.desig = true
    Paren.err = Desig.err
    Paren.cod = Desig.cod ||
                si (esPrimitivo(Desig.tipo) && Desig.tsh[Desig.lex].clase == constante)
                    apila(Desig.tsh[Desig.lex].valor)
                    Desig.etq = Desig.etq + 1
                fsi
                si (esPrimitivo(Desig.tipo) && !Paren.refh)
                    apila-ind
                    Desig.etq = Desig.etq + 1
                fsi
    Desig.etqh = Paren.etqh
    Paren.etq = Desig.etq + 1

    return Paren
}

funcion op0_R1 () {
    Op0.op = igual

    return Op0
}

funcion op0_R2 () {
    Op0.op = noigual

    return Op0
}

funcion op0_R3 () {
    Op0.op = men

    return Op0
}

funcion op0_R4 () {
    Op0.op = may

    return Op0
```

```
}

funcion op0_R5 () {
    Op0.op = menoig

    return Op0
}

funcion op0_R6 () {
    Op0.op = mayoig

    return Op0
}

funcion op1_R1 () {
    Op1.op = menos

    return Op1
}

funcion op1_R2 () {
    Op1.op = mas

    return Op1
}

funcion op2_R1 () {
    Op2.op = mod

    return Op2
}

funcion op2_R2 () {
    Op2.op = div

    return Op2
}

funcion op2_R3 () {
    Op2.op = mul

    return Op2
}

funcion op3_R1 () {
    Op3.op = lsh

    return Op3
}

funcion op3_R2 () {
    Op3.op = rsh

    return Op3
}

funcion op4_R1 () {
    Op4.op = not

    return Op4
}

funcion op4_R2 () {
    Op4.op = menos

    return Op4
}
```

```
funcion lit_R1 (LitBool) {
    Lit.valor = LitBool.valor
    Lit.tipo = LitBool.tipo

    return Lit
}

funcion lit_R2 (LitNum) {
    Lit.valor = LitNum.valor
    Lit.tipo = LitNum.tipo

    return Lit
}

funcion lit_R3 (litchar) {
    Lit.valor = stringToChar(litchar)
    Lit.tipo = character

    return Lit
}

funcion litBool_R1 () {
    LitBool.valor = true
    LitBool.tipo = boolean

    return LitBool
}

funcion litBool_R2 () {
    LitBool.valor = false
    Lit.tipo = boolean

    return LitBool
}

funcion litNum_R1 (litnat) {
    LitNum.valor = stringToNat(litnat)
    LitNum.tipo = natural

    return LitNum
}

funcion litNum_R2 (litfloat) {
    LitNum.valor = stringToFloat(litfloat)
    LitNum.tipo = float

    return LitNum
}
```

12 Formato de representación del código P

La máquina pila funciona mediante la carga de un fichero binario que define las instrucciones del código. Dicho código binario (*bytecode*) no contiene información de la tabla de símbolos o la memoria: únicamente instrucciones.

Las instrucciones vienen determinadas por un único byte, opcionalmente seguido de operandos. El tipo de un operando viene dado por la instrucción. Los operandos pueden ser:

Tipo	Valor
type	Un único byte que representa un tipo (ver tabla de tipos)
nat	Cuatro bytes que representan un entero de 31 bits sin signo
int	Cuatro bytes que representan un entero de 32 bits con signo

float	Cuatro bytes que representan un flotante en IEEE 754 binary single precision
char	Dos bytes que representan un caracter unicode UTF-16
bool	Un byte que representa un booleano (0=false, 1=true)

Las instrucciones que requieren un valor literal utilizan un sufijo dentro del propio código de operación en lugar de un argumento de tipo. tanto los argumentos de tipo como dichos sufijos siguen la siguiente tabla:

Código	Tipo
000	natural
001	integer
010	float
011	character
100	boolean

Con todo esto, podemos empezar a definir instrucciones:

Código	Operandos	Instrucción
0000 0000	-	suma (+)
0000 0001	-	resta (-)
0000 0010	-	mul (*)
0000 0011	-	div (/)
0000 0100	-	mod (%)
0000 0101	-	igual (==)
0000 0110	-	no-igual (!=)
0000 0111	-	menor (<)
0000 1000	-	menor-o-igual (<=)
0000 1001	-	mayor (>)
0000 1010	-	mayor-o-igual (>=)
0000 1011	-	and
0000 1100	-	or
0000 1101	-	despl izq (<<)
0000 1110	-	despl dcha (>>)
0000 1111	-	opuesto (- unario)
0001 0000	-	negación (not)
0010 0TTT	valor	apila(valor). *T* es el tipo de *valor*
0010 1TTT	-	in. *T* es el tipo pedido
0011 0TTT	-	cast(T). *T* es el tipo del casting
0011 1000	tipo, dir	apila-dir(tipo, dir). Tipo es un parámetro *type*, dir esun *nat*.
0011 1001	tipo, dir	desapila-dir(tipo, dir). Tipo es un parámetro *type*, dir esun *nat*.
0011 1010	tipo	apila-ind(tipo). Tipo es un *type*

0011 1011	tipo	desapila-ind(tipo). Tipo es un *tipo*
0011 1100	-	output.
0011 1101	-	stop.
0011 1110	-	swap1.
0011 1111	-	swap2.
0100 0000	-	ir-a
0100 0001	-	ir-f
0100 0010	-	ir-v
0100 0011	-	ir-ind
0100 0100	-	copia
0100 0101	tam	mover(tam). Tam es un *nat*
0100 0110	-	desapila.
0100 0111	tam	rango(tam). Tam es un *nat*

Los códigos no definidos en la tabla no corresponden a ninguna instrucción.

13 Notas sobre la implementación

13.1 Descripción de archivos

es.ucm.fdi.plg.evlib

Esta es la librería EvLib modificada para solventar algunos problemas que hemos tenido durante el desarrollo de la práctica.

plg.gr3

Contiene el main de la aplicación y una clase Util con ciertas utilidades para la aplicación.

plg.gr3.code

Contiene todas las clases de lectura y escritura de código. Su base son las clases abstractas CodeReader y CodeWriter, de las que existen implementaciones para leer y cargar de fichero, así como una implementación de CodeWriter que permite la escritura directa en una lista.

plg.gr3.data

Contiene todo lo relacionado con la gestión de datos, es decir: los tipos, los valores del lenguaje y los operadores. La clase Type representa los tipos de nuestro lenguaje. Existe una instancia de esta clase para cada tipo primitivo y para el tipo error, así como dossubclases TupleType y ArrayType para representar arrays y tuplas, respectivamente.

Los operadores se representan mediante las clases BinaryOperator y UnaryOperator, que implementan una interfaz Operator por cuestiones de comodidad en su manejo.

Los valores de nuestro lenguaje vienen representados usando las subclases de la clase abstracta Value, los cuales envuelven los tipos primitivos de Java, añadiendo la restricción a los naturales de que sólo se pueden usar valores positivos.

plg.gr3.debug

Paquete de depuración que incluye utilidades para escribir por consola errores y mensajes, indicando en ellos línea y columna (para compilación) o número de instrucción (para ejecución).

plg.gr3.errors

Paquete base para la representación de errores. Sólo incluye una clase abstracta Error, superclase de los errores de ejecución y compilación.

plg.gr3.errors.runtime

Errores en tiempo de ejecución, con base en la clase abstracta RuntimeError. Los errores de ejecución se dan en una posición del programa e instrucción concretas, lo cual queda reflejado con los atributos. Las subclases de esta clase abstracta incluidas en este paquete son los tipos de errores que podemos tener en ejecución.

plg.gr3.errors.compile

Errores en tiempo de compilación, con base en la clase abstracta CompileError. Los errores de compila-ción se dan en una posición del fichero fuente, incluyendo línea y columna, lo cual queda reflejado con los atributos. Las subclases de esta clase abstracta incluidas en este paquete son los tipos de errores que podemos tener en compilación.

plg.gr3.parser

Contiene el analizador sintáctico y todas las clase que necesita. Parte de este paquete as autogerada por CUP y JFlex. Además, incluye la definición de la tabla de símbolos, así como del descriptor de las funciones de atribución (Attribution) y de algunas clases útiles.

plg.gr3.parser.semfun

Contiene algunas funciones semánticas que se han reutilizado en la clase Attribution.

plg.gr3.vm

Definición de la máquina virtual en la clase VirtualMachine, que mantiene el estado de la máquina virtual y define métodos para que pueda manipularse externamente.

plg.gr3.vm.instr

Contiene las definiciones de instrucciones, todas ellas descendientes de una clase abstracta Instruction. Este paquete es el que implementa la ejecución de código, mediante Instruction#execute(VirtualMachine), método abstracto que todas las instrucciones deben implementar.

13.2 Otras notas

Ejecución del programa

El programa principal es un único main, incluído en laclase `plg.gr3.Main`. Para su uso se implementan dos comandos, `compile` y `run`. Ambos comandos pueden modificarse usando los sufijos `.v` y `.vv`, lo que hará que se muestren mensajes dedepuración en mayor medida y, en el caso de `run.vv`, permitirá la ejecución en modo traza, parándose tras cada instrucción.

El comando `compile` tiene dos argumentos: El fichero fuente y el fichero destino. Este comando compilará el programa pasado como fuente y volcará el *bytecode* resultante en el fichero destino. En modo depuración (`compile.v`), imprimirá alguna información útil de depuración, así como el código generado. En modo traza (`compile.vv`), mostrará además la salida de EvLib.

El comando `run` tiene un único argumento: El fichero con el *bytecode* a ejecutar. Este comando ejecutará el programa, imprimiendo detalles como la pila y la memoria en el caso de modo depuración (`run.v`) y parándose tras cada instrucción en el modo traza (`run.vv`).