Procesamiento

El lenguaje

- El lenguaje a considerar será un lenguaje imperativo del tipo utilizado para ilustrar las estrategias de generación de código P
- Nos centraremos en la sintaxis abstracta del lenguaje, dejando la provisión de una sintaxis concreta adecuada como un ejercicio
- La sintaxis abstracta viene dada por las siguientes constructoras (y géneros implicados en las mismas)
    - Representación de programas

Constructora	Descripción	Representación azucarada
`prog: List(Dec) × Inst → Prog`	Construye un programa a partir de una lista de declaraciones y una instrucción	*Ds* && *I*

    - Representación de declaraciones

Constructora	Descripción	Representación azucarada
`decVar: Tipo × String → Dec`	Construye una declaración de variable	**var** *id*:*T*
`decTipo: Tipo × String → Dec`	Construye una declaración de tipo	**type** *id*:*T*
`decProc: String × List(Param) × List(Dec) × Inst → Dec`	Construye la declaración de un procedimiento a partir de: (i) su nombre, (ii) su lista de parámetros formales, , (iii) una lista de declaraciones locales, (iv) una instrucción	**proc** *Id Ps Ds* && *I*
`param: (var|val) × String × Tipo → Param`	Construye un parámetro formal (cada parámetro formal viene descrito por (a) un modo –variable o valor-, (b) un nombre, (c) un tipo)	**param** (**var**|**val**) *id*:*T*

    - Representación de expresiones de tipo

Constructora	Descripción	Representación azucarada
`bool: Tipo`	Construye un tipo booleano	**bool**
`int: Tipo`	Construye un tipo entero	**int**
`ref: String → Tipo`	Construye un tipo sinónimo a otro dado como parámetro	**ref** *id*
`array: Integer × Tipo → Tipo`	Construye un tipo array (el primer argumento es la dimensión, el segundo el tipo base)	*T* [*num*]
`reg: List(String × Tipo) → Tipo`	Construye un tipo registro a partir de una lista de campos con sus respectivos tipos	**record** *Cs* **end** Cada definición de campo en *Cs* se denotará por *id*:*T*
`pointer: Tipo → Tipo`	Construye un tipo puntero a partir de otro dado	**pointer** *T*

    - Representación de Instrucciones

Constructora	Descripción	Representación azucarada
`asig: Desig × Exp → Inst`	Construye una instrucción de asignación	*D* = *E*
`write: Exp → Inst`	Instrucción de escritura	**write** *E*

`read: Desig → Inst`	Instrucción de lectura	**read** *Desig*
`new: Desig → Inst`	Instrucción de reserva de memoria	**new** *Desig*
`del: Desig → Inst`	Instrucción de liberación de memoria	**delete** *Desig*
`if: Exp × Inst → Inst`	Instrucción *if*	**if** *E I*
`ifElse: Exp × Inst × Inst → Inst`	Instrucción *if-Else*	**if** *E* I_0 I_1
`while: Exp × Inst → Inst`	Instrucción *while*	**while** *E I*
`bloque: List (Inst) → Inst`	Instrucción *bloque*	{*Is*}
`call: String × List(Exp) → Inst`	Instrucción *llamada a procedimiento*	*id* (*Ps*)

- Representación de Designadores

Constructora	Descripción	Representación azucarada
`var: String → Desig`	Construye un designador a partir de un nombre de variable	*id*
`selCampo: Desig × String → Desig`	Construye el designador que resulta de seleccionar un campo	*D.id*
`indxElem: Desig × Exp → Desig`	Construye el designador que resulta de indexar un elemento	*D*[*E*]
`deref: Desig → Desig`	Construye el designador que resulta de dereferir otro dado	*D->*

- Representación de Expresiones

Constructora	Descripción	Representación azucarada
`true: → Exp`	Construye la expresión *true*	**true**
`false: → Exp`	Construye la expresión *false*	**false**
`num: Integer → Exp`	Construye la expresión básica dada por un número	**num**
`mem: Desig → Exp`	Construye una expresión a partir de un designador	**mem** *D*
`igual: Exp × Exp → Exp`	Expresión *igual*	E_0 == E_1
`distinto: Exp × Exp → Exp`	Expresión *distinto*	E_0 != E_1
`menor: Exp × Exp → Exp`	Expresión *menor que*	E_0 < E_1
`mayor: Exp × Exp → Exp`	Expresión *mayor que*	E_0 > E_1
`menorOIgual: Exp × Exp → Exp`	Expresión *menor o igual que*	E_0 <= E_1
`mayorOIgual: Exp × Exp → Exp`	Expresión *mayor o igual que*	E_0 >= E_1
`suma: Exp × Exp → Exp`	Expresión *suma*	E_0 + E_1
`resta: Exp × Exp → Exp`	Expresión *resta*	E_0 - E_1
`mul: Exp × Exp → Exp`	Expresión *multiplicación*	E_0 * E_1
`div: Exp × Exp → Exp`	Expresión *división*	E_0 / E_1
`mod: Exp × Exp → Exp`	Expresión *módulo*	E_0 % E_1
`neg: Exp → Exp`	Expresión *negar*	- *E*
`and: Exp × Exp → Exp`	Expresión *y lógico*	E_0 **and** E_1
`or: Exp × Exp → Exp`	Expresión *o lógico*	E_0 **or** E_1
`not: Exp → Exp`	Expresión *no lógico*	**not** *E*

Procesamiento semántico

Vinculación de identificadores

- El proceso de vinculación de identificadores permitirá fijar el vínculo de los identificadores en: (i) expresiones de tipo, (ii) designadores, y (iii) llamadas a procedimientos.
- Para ello, se equipará a los respectivos nodos (los construidos por `ref`, `var` y `call`) de un atributo `vinculo`, que, tras el proceso de vinculación, apuntará a la correspondiente declaración.

- El proceso utilizará una tabla de símbolos global, operada por las siguientes operaciones:
    - iniciaTS(): Inicia la tabla de símbolos
    - abreBloque(): Abre un nuevo bloque de declaraciones
    - cierraBloque(): Cierra el bloque de declaraciones actual
    - insertaId(*id*, *Dec*): *resul*. Asocia una declaración con un identificador. El resultado es *cierto* si el identificador no existe en el bloque actual, y *falso* en otro caso.
    - declaracionDe(*id*): (*Dec* | **null**). Recupera la declaración de un identificador (o **null**, en caso de que el identificador no esté declarado)
- Así mismo, con los nodos creados por `reg` (representación de expresiones de tipo *registro*) se asociará una tabla `campos` que permitirá indexar de modo eficiente sus campos (en este proceso, de forma colateral, se asegurará que en las definiciones de registros no haya campos repetidos).
- El proceso podrá informar de errores relativos a identificadores duplicados en el mismo bloque, o sobre identificadores no declarados (en la práctica, será necesario almacenar en los nodos de tipo `ref`, `var` y `call` el número de fila y columna del identificador asociado en el texto fuente, a fin de proporcionar mensajes de error significativos; este aspecto se abstrae en el diseño aquí esbozado)
- En caso de que el proceso de vinculación haya anunciado errores, se asume que se interrumpirán las subsecuentes fases del procesamiento.
- Notación: mediante N`/Estructura/` se hará referencia a un nodo en el árbol de sintaxis abstracta, junto con su estructura (e.g., D`/var id:T/`)
- Como regla general, cuando un identificador sea usado, ha debido ser declarado previamente.
- La excepción es en la definición de tipos puntero: se permite hacer referencia a identificadores de tipos aún no declarados en el mismo bloque => la vinculación en las definiciones de tipo requiere dos pasadas:
    - Primera pasada: se vinculan todos los identificadores referidos en las definiciones de tipo, *excepto* los que constituyen el tipo base de definiciones de punteros.
    - Segunda pasada: se vinculan aquellos identificadores que constituyen el tipo base de definiciones de punteros.
- A continuación se detalla, mediante pseudocódigo, el proceso de vinculación (el procedimiento **vinculaDefPunteros** implementa la vinculación de identificadores que aparecen como tipos base de definiciones de punteros):
    - Programa

```
vincula(Ds && I) {
   iniciaTS();
   abreBloque();
   foreach D in Ds vincula(D);
   foreach D in Ds vinculaDefPunteros(D);
   vincula(I)
}
```

    - Declaraciones

```
vincula(var id:T) {
    vincula(T);
    if (not insertaId(id,T)) error identificador duplicado;
}
```

```
vincula(type id:T) {
    vincula(T);
    if (not insertaId(id,T)) error identificador duplicado;
}
vincula(Proc/proc Id Ps Ds && I/) {
    if (not insertaId(id,Proc)) error identificador duplicado;
    abreBloque();
    insertaId(Id,Proc); // El procedimiento existe en su propio ámbito
    foreach P/param Modo p: T/ in Ps
      insertaId(p,P);  // Los parámetros formales existen en el ámbito del procedimiento
    foreach D in Ds vincula(D);
    foreach D in Ds vinculaDefPunteros(D);
    vincula(I);
    cierraBloque();
}

vinculaDefPunteros(var id:T) {
    vinculaDefPunteros(T);
}
vinculaDefPunteros(type id:T) {
    vinculaDefPunteros(T);
}
vinculaDefPunteros(proc Id Ps Ds && I) {
    foreach param Modo p: T in Ps
      vinculaDefPunteros(T);
}
```

- Definiciones de tipo. Nótese que en **vincula** se evita vincular identificadores en declaraciones de la forma *pointer id* (dicha vinculación se lleva a cabo en **vinculaDefPunteros**)

```
vincula(bool) {}
vincula(int)  {}
vincula(ID/ref id/)   {
   ID.vinculo = declaracionDe(id);
   if (ID.vinculo == null) error identificador no declarado;
}
vincula(T[n])   {
   vincula(T);
}
vincula(R/record Cs end/)   {
  R.campos = new Map;
  foreach C/id: T/ in Cs {
      if (R.campos.contiene(id) ) error campo duplicado
      else R.campos.añade(id,C);
      vincula(T);
   }
}
vincula(pointer T) {
  unless (T == ref id)
      vincula(T)
}

vinculaDefPunteros(bool) {}
vinculaDefPunteros(int)  {}
vinculaDefPunteros(id)   {}
vinculaDefPunteros(T[n])   {
  vinculaDefPunteros(T);
}
vinculaDefPunteros(record Cs end)   {
  foreach id: T in Cs {
       vinculaDefPunteros(T);
  }
}
vinculaDefPunteros(pointer T) {
  if (T == ref id) vincula(T)
```

```
  else vinculaDefPunteros(T)
}
```

- Procesamiento de las instrucciones

```
vincula(D=E) {vincula(D); vincula(E);}
vincula(write E) {vincula(E);}
vincula(read D) {vincula(D);}
vincula(new D) {vincula(D);}
vincula(delete D) {vincula(D);}
vincula(if E I) {vincula(E); vincula(I);}
vincula(if E I0 I1) {vincula(E); vincula(I0); vincula(I1);}
vincula(while E I) {vincula(E); vincula(I);}
vincula({Is}) {
  foreach I in Is
   vincula(I);
}
vincula(Proc/id(Args)/) {
  Proc.vinculo = declaracionDe(id);
  if (Proc.vinculo == null) error identificador no declarado;
  foreach E in Args do
   vincula(E);
}
```

- Procesamiento de los designadores

```
vincula(Id/id/) {
   Id.vinculo = declaracionDe(id);
   if (Id.vinculo == null) error identificador no declarado;
}
vincula(D.id) { vincula(D);}
vincula(D[E]) { vincula(D); vincula(E);}
vincula(D→) { vincula(D); }
```

- Procesamiento de las Expresiones

```
vincula(true) { }
vincula(false) { }
vincula(num) { }
vincula(mem D) { vincula(D); }
vincula(E0 == E1) {vincula(E0); vincula(E1);}
vincula(E0 != E1) {vincula(E0); vincula(E1);}
... los casos omitidos son análogos
vincula(not E) {vincula(E); }
```

Chequeo de tipos y del resto de restricciones contextuales

- Una vez finalizado el proceso de vinculación, si no se han detectado errores, cada nodo correspondiente al uso de un identificador hará referencia a su vínculo (nodo en el que el identificador es declarado)
- El árbol de sintaxis abstracta así decorado es suficiente para llevar a cabo la comprobación del resto de las restricciones contextuales (incluidas el chequeo de tipos)
- Para facilitar el chequeo de tipos se realizará, no obstante, un preproceso previo de los subárboles asociados con las definiciones de tipo, a fin de eliminar de ellos el uso de identificadores (como resultado, los tipos de variables y parámetros formales estarán expresados como árboles –grafos, en realidad- que involucran únicamente tipos básicos, y constructores de tipos: **representaciones canónicas de los tipos**)
- Sobre programa, declaraciones e instrucciones, se aplicará un procedimiento **chequea** que realiza las comprobaciones de las restricciones contextuales sobre dichas estructuras

- Sobre las declaraciones se empleará, así mismo, un procedimiento **simplificaDefTipos** que eliminará de las mismas los identificadores de tipo, substituyéndolos por sus definiciones simplificadas.
- Sobre designadores y expresiones, el procedimiento **chequea** fijará, además, el valor de la propiedad `tipo` de los nodos correspondientes a las definiciones de tipo que representan el tipo inferido para dichos nodos.
- A continuación se proporcionan los detalles.
    - Programa

```
chequea(Ds && I) {
  chequea(Ds);
  foreach D in Ds do
     simplificaDefTipos(D);
  chequea(I);
}
```

    - Declaraciones

```
chequea(var id:T) {
    chequea(T);
}
chequea(type id:T) {
    chequea(T);
}
chequea(proc Id Ps Ds && I) {
    foreach param Modo id: T in Ps
       chequea(T);
    foreach D in Ds {
       chequea(D);
       simplificaDefTipos(D);
    }
    chequea(I)
}
simplificaDefTipos(V/var id:T/) {
    replace T in V by tipoSimplificado(T)
}
simplificaDefTipos(Type/type id:T/) {
    replace T in Type by tipoSimplificado(T)
}
simplificaDefTipos(proc Id Ps Ds && I) {
    foreach param P/Modo id:T/ in Ps do
      replace T in P by tipoSimplificado(T)
}
```

    - Definiciones de tipo:
        - Chequeo: los identificadores definidos deben corresponder con declaraciones de tipos

```
chequea(bool) {}
chequea(int)  {}
chequea(ID/ref id/)   {
   if (ID.vinculo != type id:T) error el identificador debería ser uno de tipo;
}
chequea(A/T[n]/)   {chequea(T);}
chequea(R/record Cs end/)   {
  foreach C/id: T/ in Cs {
      chequea(T);
   }
}
chequea(pointer T) {chequea(T);}
```

- Simplificación: elimina de las definiciones de tipo los nodos de tipo **ref** *id*, substituyéndolos por las correspondientes referencias (en dicho proceso, se siguen cadenas de nodos **ref** id, por lo que la referencia de substitución final nunca es a otro nodo **ref** id: esto permite asegurar que, una vez procesadas todas las declaraciones, las definiciones de tipos estén en forma canónica)

```
tipoSimplificado(N/bool/) {return N;}
tipoSimpliciado(N/int/)   {return N;}
tipoSimplificado(N/ref id/)   {
   while N=ref id' {
      let N.vinculo = type id:N' in
          N = N'
   }
   return N;
}
tipoSimplificado(A/T[n]/) {
   replace T in A by tipoSimplificado(T);
   return A;
}
tipoSimplificado(R/record Cs end/) {
   foreach C/id: T/ in Cs {
   replace T in C by tipoSimplificado(T);
   return R;
}
tipoSimplificado(P/pointer T/) {
  replace T in P by tipoSimplificado(T);
  return P;
}
```

- Instrucciones

```
chequea(D=E) {
   chequea(D);
   chequea(E);
   if (D.tipo != null && E.tipo != null &&
       not compatibles(D.tipo,E.tipo)) error incompatibilidad de tipos en asignación;
}
chequea(write E) {
  chequea(E);
  if (D.tipo != null && not tipoPresentable(E.tipo))
     error no es posible escribir valores de este tipo;
}
chequea(read D) {
  chequea(D);
  if (D.tipo != null && not tipoLegible(D.tipo))
      error no es posible leer valores de este tipo;
}
chequea(new D) {
  chequea(D);
  if (D.tipo != null && D.tipo !=pointer T) error el tipo de D debe ser un tipo puntero;
}
chequea(delete D) {
  chequea(D);
  if (D.tipo != null && D.tipo != pointer T)
      error el tipo de D debe ser un tipo puntero;
}
chequea(if E I) {
   chequea(E);
   if (E.tipo != null && E.tipo != bool ) error el tipo de E debe ser booleano;
   chequea(I);
}
chequea(if E I₀ I₁) {
   chequea(E);
```

```
   if (E.tipo != null && E.tipo != bool ) error el tipo de E debe ser booleano;
   chequea(I0);
   chequea(I1);
}

chequea(while E I) {
   chequea(E);
   if (E.tipo != null && E.tipo != bool ) error el tipo de E debe ser booleano;
   chequea(I);
}
chequea({Is}) {
  foreach I in Is
   chequea(I);
}


chequea(Proc/id(Args)/) {
  if (Proc.vinculo == proc id (Ps) Ds I) {
     if Ps.length != Args.length error discordancia en número de parámetros;
     else
       for i=0 to Ps.length-1 do {
         chequea(Args[i]);
         let Ps[i] = param Modo id: T in
           if Modo=var and not esDesignador(Args[i])
             error el parámetro i-esimo debe ser un designador;
           else if not compatibles(T, Args[i].tipo)
               error tipos incompatibles en parámetro i-esimo;
       }
  }
  else error se está invocando a un objeto que no es un procedimiento;
 }
```

- Designadores

```
chequea(Id/id/) {
   if (not validoComoDesignador(id.vinculo)) {
      error id debe ser una variable o un parámetro;
      t=null;
   else
      t = tipoEn(Id.vinculo);
   Id.tipo = t;
}

chequea(N/D.id/) {
  chequea(D);
  if (D.tipo != null) {
    if (D.tipo == record Cs end) {
      C = D.tipo.consulta(id);
      if C == null {
       error campo inexistente;
       t=null;
      }
      else
       let C=id:T in
         t=T;
    }
    else {
     error el designador debería ser de tipo registro;
     t=null;
    }
  else t=null;
  N.tipo = t;
}

chequea(N/D[E]/) {
  chequea(D);
  chequea(E);
```

```
  if D==null or E==null then
     t = null
  else if D.tipo != T[n] then {
         error el designador debería ser de tipo array;
         t=null;
  }
  else if E.tipo != int then {
         error el índice debería ser de tipo entero;
         t=null;
  }
  else  {
    let D.tipo == T[n] in
        t=T;
  }
  N.tipo = t;
}

chequea(N/D→/) {
  chequea(D)
  if D.tipo == null then t= null;
  else if D.tipo == pointer T then t=T;
  else error {el designador debería ser de tipo puntero; t=null}
  N.tipo = t;
}
```

- Procesamiento de las Expresiones

```
chequea(N/true/) {N.tipo = bool; }
chequea(N/false/) {N.tipo = bool; }
chequea(N/num/) {N.tipo = int; }
chequea(N/mem D/) { chequea(D); N.tipo=D.tipo;}
chequea(N/E₀ == E₁/) {
  chequea(E₀);
  chequea(E₁);
  if E₀.tipo == null or E₀.tipo == null then t=null;
  else if tiposComparables(E₀.tipo,E₁.tipo) then {
    t=bool;
  }
  else {error tipos no comparables; t = null}
  N.tipo = t;
}
... los casos para !=, <, >, <=, >= son análogos
chequea(N/E₀ + E₁/) {
  chequea(E₀);
  chequea(E₁);
  if E₀.tipo == null or E₁.tipo == null then t=null;
  else if operacionAritmeticaValida(E₀.tipo,E₁.tipo) then {
    t=int;
  }
  else {error tipos no comparables; t = null}
  N.tipo = t;
}
... el resto de casos es análogo
```

Ejercicio: Definir las funciones y procedimientos que se han dejado sin detallar.

Generación de código

- La generación de código se llevará a cabo:
  - Asignando tamaños a los tipos, así como niveles y direcciones a los parámetros y variables (para los procedimientos, dichas direcciones serán relativas al comienzo

del registro de activación), niveles a los procedimientos, desplazamientos a los campos de registros, y otros atributos relativos a la disposición en memoria.

- Generando código mediante la aplicación de los patrones de generación de código ya analizados.

Asignación de tamaños y direcciones

- En esta fase se decorarán los nodos asociados a las definiciones de tipos, y a las declaraciones de parámetros y variables con la siguiente información:
  - Para toda definición de tipo, un atributo `tam` que indique el número de celdas requeridas para almacenar objetos de dicho tipo
  - Para los campos de las definiciones de registro, un atributo `desp` que indique el desplazamiento de dichos campos en el registro
  - Para cada declaración de variable y de parámetro en un procedimiento:
    - Un atributo `nivel` que indicará el nivel de anidamiento en el que se encuentra declarado el objeto.
    - Un atributo `dir` que indicará la dirección del objeto (en el caso de objetos declarados en procedimientos, relativa al comienzo de los datos del registro de activación)
  - Los nodos asociados a la declaración de procedimientos se decorarán también con un atributo `nivel` que indique el nivel de anidamiento del procedimiento.
  - El nodo raíz (el asociado al programa principal) se decorará con un atributo `finDatos` que contendrá la dirección en la que finaliza el segmento de datos estáticos.
- Para facilitar el procesamiento se utilizarán dos variables globales (`dir` y `nivel`)
- Detalles del procesamiento:
  - Programa

```
asignaEspacio(N/Ds && I/) {
   N.finDatos = anidamiento(Ds);
   dir=N.finDatos;
   nivel=0;
   foreach D in Ds do
     asignaEspacio(D);
}
anidamiento(Ds) {
   anidamiento=0;
   foreach D in Ds do
     anidamiento = max(anidamiento,anidamientoDe(D));
   return anidamiento;
}
anidamientoDe(var id:T) {return 0;}
anidamientoDe(type id:T) {return 0;}
anidamientoDe(proc Id Ps Ds && I) {return 1+anidamiento(Ds);}
```

  - Declaraciones

```
asignaEspacio(N/var id:T/) {
   N.nivel=nivel;
   N.dir=dir;
   asignaTamaño(T);
   dir = dir+T.tam;
}
asignaEspacio(type id:T) {}
asignaEspacio(Proc/proc Id Ps Ds && I/) {
```

```
   copiaDir = dir;
   copiaNivel = nivel;
   nivel = nivel+1;
   Proc.nivel = nivel;
   dir = 0;
   foreach P/ param Modo id:T/ in Ps {
     P.dir = dir;
     P.nivel = nivel;
     asignaTamaño(T);
     if Modo=var then {
       dir=dir+1;
     }
     else {
       dir = dir+T.tam;
    }
   }
   foreach D in Ds do
     asignaEspacio(D);
   nivel=copiaNivel;
   dir=copiaDir;
}
```

- Definiciones de tipo:
  - Nótese que ya no tiene sentido considerar el caso **ref** *id*
  - Nótese también que la asignación de tamaño a las definiciones de tipos se realiza *bajo demanda*, por lo que, en las definiciones de tipo compuestas, antes de proceder se comprueba si se conoce ya el tamaño, para evitar recalcularlo

```
asignaEspacio(N/bool/) {N.tam=1;}
asignaEspacio(N/int/)  {N.tam=1;}
asignaEspacio(A/T[n]/)   {
   if A.tam == ? then {
     asignaEspacio(T);
     A.tam = T.tam * n;
  }
}
asignaEspacio(R/record Cs end/)   {
  if R.tam == ? then {
    R.tam = 0;
    foreach C/id: T/ in Cs {
      C.desp = R.tam;
      asignaEspacio(T);
      R.tam = R.tam + T.tam;
   }
  }
}
asignaEspacio(P/pointer T/) {
  asignaEspacio(T); // se necesita conocer el tamaño de T para generar
                    // código para las instrucciones de gestión de memoria dinámica
  P.tam = 1;
}
```

Generación de código

- En esta fase se aplican los patrones ya explicados para la generación de código.
- El atributo `cod` en cada nodo mantendrá el código para dicho nodo.
- Para permitir fijar adecuadamente las direcciones de salto, se asociará un par de atributos con los nodos para las instrucciones: `inicio` (dirección de la primera instrucción generada para el nodo), y `fin` (dirección de la última instrucción).

- Los nodos de definición de procedimiento tendrán asociados un atributo `dirComienzo`, que indicará la dirección de comienzo de los mismos.
- Se utiliza una variable global `cinst` para llevar cuenta del número de instrucciones generadas.
- Detalles del procesamiento:
    - Programa

```
codigo(P/Ds && I/) {
   cinst = numeroInstruccionesActivacionPrograma(P);
   foreach D in Ds do codigo(D);
   codigo(I);
   P.cod = codigoActivacionPrograma(P) ||
           codigo(Ds) || codigo(I);
}
```

    - Declaraciones

```
codigo(var id:T) {}
codigo(type id:T) {}

codigo(Proc/proc Id Ps Ds && I/) {
   Proc.cod = []
   foreach D in Ds do {
    codigo(D);
    Proc.cod = Proc.cod || D.cod;
   }
   Proc.dirComienzo = cinst;
   cinst = cinst + numeroInstrucccionesPrologo(Proc);
   codigo(I);
   cinst = cinst + numeroInstrucccionesEpilogo(Proc);
   Proc.cod = Proc.cod || codigoPrologo(P) || I.cod || codigoEpilogo(P)
}
```

    - Instrucciones

```
codigo(I/D=E/) {
   I.comienzo=cinst;
   codigo(D); codigo(E); cinst=cinst+numeroInstruccionesFinAsig(I);
   I.fin = cinst;
   I.cod = D.cod || E.cod || codigoFinAsig(I);
}
codigo(I/write E/) {
   I.comienzo=cinst;
   codigo(E); cinst=cinst+numeroInstruccionesFinWrite(I);
   I.fin = cinst;
   I.cod = E.cod || codigoFinWrite(I);
}
codigo(I/read D/) {
   I.comienzo=cinst;
   codigo(D); cinst=cinst+numeroInstruccionesFinRead(I);
   I.fin = cinst;
   I.cod = D.cod || codigoFinRead(I);
}
codigo(I/new D/) {
   I.comienzo = cinst;
   codigo(D); cinst=cinst+numeroInstruccionesFinNew(I);
   I.fin = cinst;
   I.cod = D.cod || codigoFinNew(I);
}
codigo(I/delete D/) {
   I.comienzo=cinst;
   codigo(D); cinst=cinst+numeroInstruccionesFinDelete(I);
   I.fin = cinst;
   I.cod = D.cod || codigoFinDelete(I);
}
```

```
codigo(I/if E I/) {
   I.comienzo=cinst;
   codigo(E); cinst=cinst+numeroInstruccionesAccesoValor(E); cinst=cinst+1; codigo(I);
   I.fin = cinst;
   I.cod = E.cod || codigoAccesoValor(E) || ir_f(I.fin) || I.cod;
}
codigo(I/if E I0 I1/) {
  I.comienzo=cinst;
  codigo(E); cinst=cinst+numeroInstruccionesAccesoValor(E); cinst=cinst+1;
  codigo(I0); cinst=cinst+1; codigo(I1);
  I.cod = E.cod || codigoAccesoValor(E) || ir_f(I1.comienzo) || I0.cod ||
          ir_a(I1.fin) || I1.cod;
  I.fin = cinst;
}


codigo(I/while E C/) {
  I.comienzo=cinst;
  codigo(E); cinst=cinst+numeroInstruccionesAccesoValor(E);
  cinst=cinst+1; codigo(C); cinst=cinst+1;
  I.cod = E.cod || codigoAccesoValor(E) || ir_f(C.fin) || C.cod || ir_a(I.comienzo)
  I.fin = cinst;
}
codigo(I/{Is}/) {
  I.comienzo=cinst;
  I.cod = []
  foreach I' in Is {
   codigo(I');
   I.cod = I.cod || I'.cod;
 }
 I.fin=cinst;
}
codigo(Proc/id(Ps)/) {
  Proc.comienzo=cinst;
  Proc.cod = codigoComienzoPaso(Proc);
  cinst = cinst + numeroInstruccionesComienzoPaso(Proc);
  for E in Ps do {
    Proc.cod = Proc.cod || dup; cinst=cinst+1;
    codigo(E);
    Proc.cod = Proc.cod || codigoPaso(Proc,E);
    cinst = cinst + numeroInstruccionesPaso(Proc,E);
  }
  Cinst = cinst + numeroInstruccionesFinLlamada(Proc);
  Proc.cod = Proc.cod || codigoFinLlamada(Proc);
  Proc.fin = cinst;
}
```

- Designadores

```
codigo(Id/id/) {
  Id.codigo = codigoAccesoId(Id);
  cinst = cinst + numeroInstruccionesAccesoId(Id);
}

codigo(N/D.id/) {
  codigo(D); cinst = cinst+numeroInstruccionesAccesoCampo(N);
  N.cod = D.cod || codigoAccesoCampo(N);
}

codigo(N/D[E]/) {
  codigo(D); codigo(E);
  cinst = cinst+numeroInstruccionesIndexacion(N);
  N.cod = D.cod || E.cod || codigoIndexacion(N);
}

codigo(N/D→/) {
```

```
  codigo(D);
  cinst = cinst+numeroInstruccionesDereferencia(N);
  N.cod = D.cod || codigoDereferencia(N);
}
```

- Procesamiento de las Expresiones

```
codigo(N/true/) {N.cod = codigoTrue(N); cinst=cinst+codigoTrue(N); }
codigo(N/false/) {N.cod = codigoFalse(N); cinst=cinst+codigoFalse(N);}
codigo(N/num/) {N.cod = codigoInt(N); cinst=cinst+codigoInt(N);}
codigo(N/mem D/) { codigo(D); N.cod = D.cod;}
codigo(N/E0 + E1/) {
  codigo(E0); cinst=cinst+numeroInstruccionesAccesoValor(E0);
  codigo(E1); cinst=cinst+numeroInstruccionesAccesoValor(E1);
  cinst=cinst+numeroInstruccionesCodigoSuma(N);
  N.cod = E0.cod || codigoAccesoValor(E0) || E1.cod || codigoAccesoValor(E1) ||
          codigoSuma(N);
}
... el resto de los casos son análogos
```

Ejercicio: Definir las funciones y procedimientos que se han dejado sin detallar.