Universidad Nacional de Educación a Distancia
Departamento de Lenguajes y Sistemas Informáticos

Práctica de Procesadores de lenguajes

Especificación del lenguaje PasQal

Dpto. de Lenguajes y Sistemas Informáticos
ETSI Informática, UNED

Felisa Verdejo (coordinadora)
Emilio Julio Lorenzo

Curso 2009 - 2010

Contenido

1 Introducción

En este documento se define la práctica de la asignatura de Procesadores de lenguajes correspondiente al curso 2009/2010. El objetivo de la práctica es realizar un compilador del lenguaje PasQal, variación del lenguaje de programación Pascal.

Primero se presenta una descripción del lenguaje elegido y las características especiales que tiene. A continuación se indicará el trabajo a realizar por los alumnos en diferentes fases, junto con las herramientas a utilizar para su realización.

A lo largo de este documento se intentará clarificar todo lo posible la sintaxis y el comportamiento del compilador de PasQal, por lo que es importante que el estudiante lo lea detenidamente por completo.

2 Descripción del lenguaje

Este apartado es una descripción técnica del lenguaje PasQal, una versión del lenguaje Pascal. En los siguientes apartados presentaremos la estructura general de los programas escritos en dicho lenguaje describiendo primero sus componentes léxicos y discutiendo después cómo éstos se organizan sintácticamente para formar construcciones del lenguaje semánticamente válidas.

2.1 Aspectos Léxicos

Desde el punto de vista léxico, un programa es una secuencia ordenada de TOKENS. Un TOKEN es una entidad léxica indivisible que tiene un sentido único dentro del lenguaje. En términos generales es posible distinguir diferentes tipos de TOKENS: Los operadores aritméticos, relacionales y lógicos, los delimitadores como los paréntesis o los corchetes, los identificadores utilizados para nombrar variables, constantes o nombres de procedimientos, o las palabras reservadas del lenguaje son algunos ejemplos significativos. A lo largo de esta sección describiremos en detalle cada uno de estos tipos junto con otros elementos que deben ser tratados por la fase de análisis léxico de un compilador.

2.1.1 Comentarios

Un comentario es una secuencia de caracteres que se encuentra encerrada entre los delimitadores de principio de comentario y final de comentario: “{” y “}”, respectivamente. Todos los caracteres encerrados dentro de un comentario deben ser ignorados por el analizador léxico. En este sentido su procesamiento *no debe generar TOKENS* que se comuniquen a las fases posteriores del compilador. En nuestra versión de Pascal **no** es posible realizar *anidamiento* de comentarios. De esta manera, dentro de un comentario no pueden aparecer los delimitadores de comentario “{” y “}” para acotar el comentario anidado.

Otra posibilidad de nuestro lenguaje es declarar un comentario de una línea con el delimitador “//”. En este caso todos los caracteres comprendidos entre “//” y un salto de línea serán ignorados, con la excepción de “{“ y “}“ que de encontrarse deberá generar un error léxico

Algunos ejemplos de comentarios correctos e incorrectos son los siguientes:

Listado 1. Ejemplo de comentarios

```
{ Este es un comentario correcto }

{ Es comentario contiene varias líneas

     Esta es la primera línea

     Esta es la segunda línea   }

{ Este es un comentario erróneo con comentarios anidados

    { Comentario Anidado 1 }

}

{ Este es un comentario erróneo } }

{ Este es un comentario { erróneo }

{ Este es un comentario erróneo no cerrado

// Este es un comentario de línea correcto

// Este es un comentario de línea erróneo {

// Este es un comentario de } línea erróneo
```

2.1.2 Constantes literales

Estas constantes no deben confundirse con la declaración de constantes simbólicas, que permiten asignar nombres a ciertas constantes literales, para ser referenciadas por nombre dentro del programa fuente, tal como se verá más adelante. En concreto, se distinguen los siguientes tipos de constantes literales:

- **Enteras**. Las constantes enteras permiten representan valores enteros no negativos. Por ejemplo: `0`, `32`, `127`, etc. En este sentido, no es posible escribir expresiones como `-2`, ya que el operador unario "–", no existe en este lenguaje. Si se pretende representar una cantidad negativa será necesario hacerlo mediante una expresión cuyo resultado será el valor deseado. Por ejemplo para representar – 2 podría escribirse `0 - 2`

- **Lógicas**. Las constantes lógicas representan valores de verdad (cierto o falso) que son utilizados dentro de expresiones lógicas como se verá más adelante. Únicamente existen 2 que quedan representadas por las palabras reservadas *true* y *false* e indican el valor cierto y falso respectivamente

- **Cadenas de caracteres**. Las constantes literales de tipo cadena consisten en una secuencia ordenada de caracteres ASCII. Están delimitadas por las comillas dobles, por ejemplo: "ejemplo de cadena". Las cadenas de caracteres se incluyen en la práctica únicamente para poder escribir mensajes de texto por pantalla mediante las

instrucciones `write()` y `writeln()` (ver más adelante), pero no es necesario tratarlas en ningún otro contexto. Es decir, *no se crearán variables de este tipo*. No se tendrán en cuenta el tratamiento de caracteres especiales dentro de la cadena ni tampoco secuencias de escape.

2.1.3 Identificadores

Un identificador consiste, desde el punto de vista léxico, en una secuencia ordenada de caracteres y dígitos que comienzan obligatoriamente por una letra. Los identificadores se usan para nombrar entidades del programa tales como las variables o los subprogramas definidos por el programador. El lenguaje **no** es sensible a las mayúsculas (case sensitive), lo que significa que dos identificadores compuestos de los mismos caracteres y que difieran únicamente en el uso de mayúsculas o minúsculas se consideran iguales. Por ejemplo, `Abc` y `ABC` son identificadores iguales. La longitud de los identificadores no está restringida.

2.1.4 Palabras reservadas

Las palabras reservadas son tokens del lenguaje que, a nivel léxico, tienen un significado especial de manera que no pueden ser utilizadas para nombrar otras entidades como variables, constantes, funciones o procedimientos.

A continuación se muestra una tabla con las palabras reservadas del lenguaje así como una breve descripción aclarativa de las mismas. Su uso se verá en más profundidad en los siguientes apartados.

PALABRA CLAVE	DESCRIPCIÓN
`and`	Y lógica
`begin`	Comienzo de bloque de sentencias
`boolean`	Tipo lógico
`const`	Comienzo de bloque de declaración de constantes
`else`	Comienzo del cuerpo de alternativa de una condicional if
`end`	Final de bloque de sentencias
`false`	Constante lógica que representa falso
`for`	Comienzo de un bucle for

`function`	Comienzo de una función
`if`	Comienzo de una sentencia condicional if
`in`	Operador de pertenencia a un conjunto
`integer`	Tipo entero
`of`	Declaración de conjuntos
`or`	O lógica
`procedure`	Comienzo de un procedimiento
`program`	Comienzo del programa
`record`	Declaración de una estructura tipo registro
`repeat`	Comienzo sentencia repeat
`set`	Declaración de una estructura tipo conjunto
`then`	Comienzo del cuerpo de una sentencia if
`to`	Asignación de rango en un bucle for
`true`	Constante lógica que representa verdadero
`type`	Comienzo de una declaración de tipos
`until`	Declaración de expresión de fin en una sentencia repeat
`var`	Comienzo de bloque de declaración de variables e indicador de paso por referencia
`write`	Procedimiento predefinido que muestra por pantalla un entero o cadena de texto (Sin salto de línea)
`Writeln`	Procedimiento predefinido que muestra un salto de línea

2.1.5 Delimitadores

El lenguaje define una colección de delimitadores que se utilizan en diferentes contextos. A continuación ofrecemos una relación de cada uno de ellos:

DELIMITADOR	DESCRIPCIÓN
"	Delimitador de constante literal de cadena
( )	Delimitadores de expresiones y de parámetros
[ ]	Delimitadores de rango
{ }	Delimitadores de comentario
// (salto línea)	Delimitadores de comentario en línea
,	Delimitador en listas de identificadores
..	Delimitador entre valores de un rango
;	Delimitador en secuencias de sentencia
:	Delimitador de tipo en una declaración de variable
=	Delimitador de tipo en una declaración de tipo y delimitador en una declaración de constante

2.1.6 Operadores

Existen diferentes tipos de operadores que son utilizados para construir expresiones por combinación de otras más sencillas como se discutirá más adelante. En concreto podemos distinguir los siguientes tipos:

Operadores aritméticos	+ (suma aritmética, unión de conjuntos) − (resta aritmética)
Operadores relacionales	< (menor) > (mayor) = (igual) <> (distinto)
Operadores lógicos	`and` (conjunción lógica) or (disyunción lógica)
Operadores de asignación	:= (asignación) = (definición de constantes simbólicas)
Operadores de acceso	. (acceso al campo de un registro)
Operadores especiales	IN (pertenencia a un conjunto) ^ (operador de indirección. Acceso al contenido apuntado por un puntero) @ (operador de dirección. Dirección en memoria de una variable)

Obsérvese que, como se advirtió con anterioridad, no se consideran los operadores unarios + y −, de forma que los literales numéricos que aparezcan en los programas serán siempre sin signo (positivos). Así, los números negativos no aparecerán en el lenguaje fuente, pero sí pueden surgir en tiempo de ejecución como resultado de evaluar una expresión aritmética de resultado negativo, como por ejemplo 1-4.

2.2 Aspectos Sintácticos

A lo largo de esta sección describiremos detalladamente las especificaciones sintácticas que permiten escribir programas correctos. Comenzaremos presentado la estructura general de un programa y, posteriormente, iremos describiendo cada uno de las construcciones que aparecen en detalle.

2.2.1 Estructura de un programa y ámbitos de visibilidad

Un programa en PasQal es un fichero de código fuente con extensión '.p' que tiene la siguiente estructura sintáctica:

```
program <nombre_del_programa>;
    [ const
        <declaraciones_de_constantes globales> ]
    [ type
        <declaraciones_de_tipos_compuestos globales> ]
    [ var
        <declaraciones_de_variables globales> ]
    [ <declaraciones_de_subprogramas> ]

begin
      <bloque_de_instrucciones>
end.
```

Las partes encerradas entre corchetes son opcionales. Es decir, pueden no estar presentes en el código.

Todo programa en PasQal empieza por la palabra reservada `program` y a continuación el nombre del programa acabado en punto y coma. Las siguientes partes son opcionales, pero en caso de aparecer **han de guardar el orden preestablecido en la estructura anterior**. Primero se declaran constantes (*const*) tipos (*type*) y variables (*var*) globales, en ese orden. Las palabras clave que definen cada sección sólo han de aparecer una vez. Por ejemplo, no es correcto escribir:

```
var a:integer;
var b:integer;
```

La forma correcta es:

```
var
      a:integer;
      b:integer;
```

A continuación se declaran los subprogramas en caso de necesitarse. Cómo se verá más adelante un subprograma puede ser un procedimiento o una función y tiene una estructura similar a la de *program*. Por último se define el cuerpo del programa delimitado con las palabras reservadas *begin* y *end*. Nótese que un programa ha de terminar con punto "." Y no con punto y coma ";" como los subprogramas.

Cada subprograma constituye un bloque sintáctico que determina un ámbito de visibilidad. En cada ámbito sólo están accesibles un subconjunto de todos los identificadores (constantes, tipos, variables o subprogramas) definidos en el programa. Es decir, las reglas de ámbito de un

lenguaje definen el alcance que tienen los identificadores de un programa dentro de cada punto del mismo. En este sentido, en PasQal distinguimos 3 posibilidades:

- **Referencias globales.** Estos identificadores son los que se declaran directamente en la primera sección del programa, antes de los subprogramas. Se dice que pertenecen al ámbito global del mismo y, por tanto, están accesibles desde cualquier otro ámbito, ya sea éste el programa principal o cualquiera la de los subprogramas definidos.

- **Referencias locales**. Los identificadores son locales a un ámbito cuando son solamente accesibles desde dentro de dicho ámbito. Por ejemplo, todas las variables y tipos definidos dentro de un subprograma (más los parámetros en el caso de las funciones) son locales al mismo y sólo pueden ser accedidos por el código del bloque.

- **Referencias no locales.** Dado que en esta práctica está *permitido la declaración de subprogramas anidados* y que éstos pueden incluir declaración de variables, tiene sentido hablar de anidamiento de ámbitos. En este punto la regla de alcance que se aplica prescribe que cuando se referencia un identificador dentro de un bloque, el ámbito inmediato que se consulta es el de dicho bloque. Si éste no contiene la referencia buscada, la búsqueda continua por el ámbito padre de acuerdo a la jerarquía de anidamiento de la declaración. Así se procede sucesivamente hasta llegar al ámbito base que es el ámbito global. A estas referencias, que se alcanzan atravesando varios ámbitos de declaración anidados se las conoce con el nombre de referencias no locales.

El siguiente listado muestra un ejemplo de la estructura de un programa en PasQal y contiene casos que ilustran cada uno de los 3 tipos de referencias reconocidos.

Listado 2. Ejemplo de un programa en PasQal

```
program ejemploPasQal;
      const
            cierto = true;
            falso = false;
      type
            usuario = record
                  dni : integer;
                  edad : integer;
                  casado : boolean;
            end;
            mayorEdad = set of 18..35;
```

```
var
      usuario1: usuario; // variable global
      rangoEdad : mayorEdad;  // variable global
{Declaración de subprogramas }
procedure imprimeUsuario(dni, edad:integer; casado:boolean);
      var debug : boolean; // variable local
      procedure imprimeCasado(valor:boolean);
      begin
            if  (debug=true)  // referencia no local
               if (valor=true) then  // referencia local
                  write("casado");
               else
                  write("soltero");
      end;
begin
      debug := true;  // referencia local
      write(dni); write("-");  // referencia local (parámetro)
      write(edad); write("-"); // referencia local (parámetro)
      imprimeCasado(casado);   // referencia local (parámetro)
end;

function mayorDeEdad (e : integer): boolean;
var punt: ^integer; // variable local
begin
      punt := @e;  // referencia local
      if (punt^ IN rangoEdad) then
            mayorDeEdad := true;  // referencia global
      else
            mayorDeEdad := false; // referencia global
end;
```

```
begin
// Comienzo del programa principal
usuario1.dni:=1234567; // referencia global
usuario1.edad:=15;     // referencia global
usuario1.casado:=false; // referencia global
if (mayorDeEdad(usuario1.edad))then
begin
      write("usuario:");
      imprimeUsuario(usuario1.dni, usuario1.edad, usuario1.casado);
      write(" es mayor de edad");
end;
// Fin del programa
end;
```

2.2.2 Declaraciones de constantes simbólicas

Las constantes simbólicas, cómo se ha comentado antes, constituyen una representación nombrada de datos constantes, cuyo valor va a permanecer inalterado a lo largo de la ejecución del programa. Han de declararse al inicio de un programa o subprograma obligatoriamente (ver declaración de subprogramas más adelante).

Algunos lenguajes resuelven el procesamiento de las constantes en la fase de análisis léxico haciendo una pasada sobre el fichero fuente y sustituyendo cada aparición de una constante simbólica por su correspondiente valor literal. No obstante en esta práctica se pretende que esta tarea sea llevada a cabo por las fases posteriores (análisis sintáctico, semántico y generación de código).

En esta práctica las constantes simbólicas pueden ser de tipo entero positivo o lógico (definidas en el siguiente apartado). La sintaxis para la declaración de constantes simbólicas enteras es la siguiente:

```
nombre = constanteLiteral;
```

Donde `nombre` es el nombre simbólico que recibe la constante definida dentro del programa y *`constanteLiteral`* un valor constante literal de tipo entero positivo (número sin signo) o lógico (*true* o *false*) de manera que cada vez que este aparece referenciado dentro del código se sustituye por su valor establecido en la declaración. La palabra reservada `const` marca el inicio de la sección de declaraciones de constantes y sólo puede aparecer una vez dentro de cada ámbito.

A continuación se muestran algunos ejemplos de declaración de constantes.

Listado 3. Ejemplos de declaración de constantes simbólicas enteras

```
const

uno = 1;

dos = 2;

cierto = true;

tres = 2+1; {error, ha de ser una cte. literal no una expresión}

cuatro = +4; {error, el entero no ha de llevar signo}
```

2.2.3 Declaración de tipos

La familia de tipos de un lenguaje de programación estructurado puede dividirse en 2 grandes familias: tipos primitivos del lenguaje y tipos estructurados o definidos por el usuario. A continuación describimos la definición y uso de cada uno de ellos.

2.2.3.1 Tipos primitivos

Los tipos primitivos del lenguaje (también llamados tipos predefinidos) son todos aquellos que se encuentran disponibles directamente para que el programador tipifique las variables de su programa. En concreto el lenguaje PasQal reconoce tres tipos primitivos: el tipo entero, el tipo lógico (boolean o booleano) y el tipo puntero (a entero). En los apartados subsiguientes abordamos cada uno de ellos en profundidad.

Tipo entero (integer)

El tipo entero representa valores enteros positivos y negativos. Por tanto, a las variables de este tipo se les puede asignar el resultado de evaluar una expresión aritmética, ya que dicho resultado puede tomar valores enteros positivos y negativos. El rango de valores admitido para variables de este tipo oscila entre -32728 y +32727 ya que el espacio en memoria que es reservado para cada variable de este tipo es de 1 palabra (16 bits).

Desde el punto de vista sintáctico, el tipo entero se representa con la palabra reservada `integer`. La aplicación de este tipo aparece en distintos contextos de uso como en la declaración de variables, la declaración de parámetros de funciones o los tipos de retorno de las mismas (véase más adelante).

Listado 4. Ejemplos de uso del tipo entero

```
var

      a, b : integer;

procedure escribenum (num: integer);

function suma (a: integer, b: integer):integer;
```

Tipo lógico (boolean)

El tipo lógico representa valores de verdad, representados por las constantes literales *true* y *false*. Para referirse en PasQal a este tipo de datos se utiliza la palabra reservada *boolean*. A continuación se muestran unos ejemplos de declaración de tipo lógico.

Listado 5. Ejemplos de uso del tipo lógico

```
var

      esCierto:boolean

procedure escribeValorDeVerdad (valor:boolean);
```

Tanto en el caso de tipos enteros o lógicos los datos tendrán un *tamaño de memoria* de una palabra (16 bits) dentro del entorno de ejecución ENS2001.

Tipo puntero

Una variable de tipo puntero (también llamada simplemente puntero) almacenará la dirección en memoria de otra variable. Otra forma de describirlo es diciendo que un puntero "apunta" a la posición de memoria de una variable. En general, los punteros se tipifican para indicar el tipo de datos de la variable a las que apuntan. Sin embargo, en PasQal los punteros pueden apuntar solo a variables de tipo entero. Esta es la razón por la cual consideraremos a los punteros a enteros, tipos primitivos de este lenguaje. Lo que significa que **NO** pueden aparecer definidos en la sección type (ver apartado de tipos compuestos). Para representar el tipo puntero se usa la construcción sintáctica ^integer. La sintaxis para declarar variables de tipo puntero es:

```
nombreVariable : ^integer;
```

Donde `nombreVariable` es el nombre de la variable tipo puntero.

Para asignar valor a un puntero la sintaxis (ver apartado sentencias de asignación) es la siguiente:

```
p:=@varEntero;

p:=q;
```

Siendo `p` un puntero y `varEntero` una variable de tipo entero. La primera sentencia almacenará en p la dirección de memoria dónde está la variable varEntero. *No* es correcto asignar a p otro valor, por ejemplo entero o booleano. En ese caso deberá generarse un error de tipos. Si se pueden asignar punteros a otros punteros. En la sintaxis anterior p y q son punteros. En este caso p apuntaría a la misma variable a la que apunta q, es decir, almacenaría la dirección de memoria de la variable a la que apunta q.

Para asignar un nuevo valor a la variable a la que apunta el puntero se utiliza la siguiente sintaxis:

```
p^:= expEntera;
```

Dónde expEntera es una expresión de tipo entero. Hay que destacar que esta expresión puede ser la dirección de memoria de una variable, ya que no deja de ser un valor entero. Sería válida por ejemplo la siguiente sentencia: p^:=@var;. La expresión *p^* devuelve el valor de la variable a la que apunta el puntero, es por tanto un valor entero. **NO** se debe considerar la posibilidad de una doble indirección del tipo p^^. En ese caso se debe generar un error.

No es necesario controlar que un puntero ha sido correctamente inicializado a la hora de acceder a él. Por tanto, **No** es necesario implementar el concepto de *null*.

En el siguiente listado se muestra un ejemplo de uso de punteros.

Listado 6. Ejemplos de asignación entre punteros

```
var
      p, q :^integer;
      a, b: integer;
begin
      a:=3;
      p:=@a;
      write (p^); {imprime 3}
      write(p); {imprime la dirección de memoria de a}
      q:=p;
      a:=a+1;
      write(q^); {imprime 4, valor actual de a}
      write(q); {imprime la dirección de memoria de a}
      b:=p+1; {p devuelve la dirección de memoria de a}
      write(b); {imprime la dirección de memoria mas 1 de a}
      p:=a;  {genera error por asignar una variable a un puntero}
      p:=q+1; {genera error por asignar una expresión a un
puntero}
end;
```

2.2.3.2 Tipos Estructurados

Los tipos estructurados (también llamados tipos *definidos por el usuario* o tipos *compuestos*) permiten al programador definir estructuras de datos complejas y establecerlas como un tipo más del lenguaje. Estas estructuras reciben un nombre (identificador) que sirve para referenciarlas posteriormente en el código del programa (declaración de variables, parámetros, campos de registros, etc.). Así en PasQal no existen tipos anónimos. Es decir, no es posible definir una estructura de datos para tipificar una variable directamente en la sección

de declaración de variables. En su lugar hay que crear previamente un tipo estructurado (con nombre) y usar dicho nombre después en la declaración de variables. En lo que respecta al sistema de tipos, la equivalencia de éstos es nominal, no estructural. Es decir dos tipos serán equivalentes únicamente si tienen el mismo nombre. Los tipos compuestos los define el usuario siempre en la sección `type` dentro de program o de un subprograma.

La sintaxis para la declaración de un tipo compuesto debe incluirse dentro de la sección de declaración de tipos y sigue la siguiente estructura sintáctica:

```
IdentificadorTipo = <<definición de tipo>>
```

Donde <<definición de tipo>> es una expresión de tipo de las definidas a continuación. En este sentido hay que precisar que: 1) no existen tipos sinónimos (por ejemplo, es un error TEntero = integer), 2) cada tipo puede tener un único nombre (es un error Ta, Tb = <<expresión de tipos>>) y 3) que no es posible definir tipos puntero (ya que como dijimos se consideran en PasQal tipos primitivos). En concreto se distinguen dos tipos de datos compuestos en PasQal: los conjuntos y los registros.

Tipo Conjunto

Las variables de tipo conjunto almacenan una colección de valores no repetidos que se encuentran comprendidos dentro de un rango de valores enteros positivos. Por ejemplo, si definimos un tipo conjunto con rango 1..3, los valores posible que pueden tomar las variables de ese tipo son: el 1, el 2, el 3, el 1 y el 2, el 1 y el 3, el 2 y el 3, el 1 el 2 y el 3, o el conjunto vacio (sin elementos). La sintaxis para declarar un tipo conjunto es:

```
nombreTipo = set of valorInicial..valorFinal;
```

Donde nombreTipo es el identificador que se da al nuevo tipo conjunto. El rango de posibles valores viene definido por las *constantes enteras* valorIncial y valorFinal. Estos valores han de ser constantes enteras, ya sean literales o simbólicas. Por tanto no pueden ser ni expresiones ni variables. Se debe comprobar que el valor inicial es menor o igual al del valor final, para en caso contario generar un error. En el listado siguiente se muestran unos ejemplos de definiciones de tipos conjunto.

Listado 7. Ejemplos de tipo conjunto

```
type
      unoAdiez = set of 1..10;
      tresAcinco = set of 3..5;
{ Las siguientes definiciones serian erróneas }
      error1 = set of 1..a;  // a no es entero
      error2 = set of 1..2+3; // 2+3 es una expresión
      error3 = set of 1..f(1); // f(1) es una expresión
      error4 = set of 3..1  // 3 es mayor que 1
```

Tras declarar una variable de tipo conjunto (ver apartado de declaraciones de variables) se le pueden asignar valores con la siguiente sintaxis:

```
varSet := [valorIncial .. valorFinal];

varSet := [];
```

En esta definición los corchetes no significan que se trata de una parte opcional en el código. Son delimitadores para declarar un *rango* (Importante: no aparecen en la definición del tipo). En este caso valorIncial y valorFinal son de tipo entero, pudiendo ser expresiones, resultado de llamadas a función, etc. Por ejemplo, suponiendo una variable s de tipo unoAdiez del listado anterior, la sentencia s:= [2..5]; almacenaría en la variable s el rango de números comprendidos entre 2 y 5, es decir: 2,3,4 y 5. En caso de intentar asignar un rango que no esté en los valores del tipo definido se ha de generar un error y terminar el programa. Esta comprobación debería realizarse en tiempo de ejecución, por lo tanto ha de implementarse en el código final. La segunda forma indicada se corresponde con la asignación del conjunto vacío. En este caso varSet no contendría ningún valor.

No se pueden asignar otros tipos que no sea una expresión de rango. Por ejemplo, intentar asignar una variable de tipo entero o booleano a una variable de tipo conjunto debe generar un error de tipos. Tampoco se puede asignar un conjunto a otro conjunto (excepto con la unión). Serían errores también asignaciones de la forma: s:=[1,2,3] s:=[1]. En caso de querer asignar un solo valor sería de la forma [1..1].

A continuación se van a explicar en este apartado varias operaciones que se pueden realizar sobre variables de tipo conjunto, como son la pertenencia y la unión.

Pertenencia. Esta operación permite saber si un determinado valor entero se encuentra entre los valores asignados a una variable de tipo conjunto. La sintaxis es:

```
valorEntero IN variableTipoConjunto
```

`valorEntero` representa un valor de tipo entero, pudiendo ser una expresión, constante literal o simbólica, etc; variableTipoConjunto representa a una variable de tipo conjunto.

Esta expresión devuelve un valor booleano que es true en caso de que el valorEntero esté dentro del conjunto. Siguiendo con el ejemplo anterior de la variable s, en caso de tener: "*3 IN s"*, devolvería true, mientras que "*8 IN s"* devolvería false. Debido a este comportamiento lo más habitual es que este tipo de expresiones aparezcan en sentencias de tipo IF o DO UNTIL, que se explicarán más adelante.

Unión. Esta operación realiza la unión de dos o más conjuntos. Es decir, el conjunto resultado contiene todos los elementos de los conjuntos que se "unen". La sintaxis es:

```
var1TipoConjunto := var2TipoConjunto + var3TipoConjunto [+
varNTipoConjunto];
```

Los corchetes indican partes opcionales en el código. En el caso de esta práctica, todos los operandos han de ser variables de tipo conjunto. No es posible realizar la unión de rangos de

valores ni la unión de una variable con un rango. En concreto *todas las variables tanto de la expresión como de la asignación han de ser del mismo tipo de conjunto*.

En la unión de conjuntos hay que recordar que en caso de existir dos elementos iguales sólo se incluye una vez en el conjunto resultado. En el siguiente listado se puede observar un ejemplo.

Listado 8. Ejemplos de unión de conjuntos

```
type
      unoAdiez = set of 1..10;
      dosAcinco = set of 2..5;
var
      c1, c2, c3, c4 : unoAdiez;
      c5, c6: dosAcinco;
begin
      c2:=[2..4];
      c3:=[3..5];
      c1:= c2+c3;
      c2:=[7..10];
      c4:= c2+c3;
      c6:=[2..3];
      {la siguiente sentencia daría error ya que se unen tipos
diferentes}
      C5:=c2+c6;
end;
```

En este ejemplo la variable c1 contendría los valores: 2,3,4,5. El 3 y 4 no se repiten. La variable c4 contendrá: 7,8,9,10,3,4,5. La asignación a c5 generaría un error de tipos en compilación al intentar unir conjuntos de diferente tipo.

No es necesario controlar que un conjunto ha sido *inicializado* a la hora de hacer operaciones sobre él. En todo caso sería recomendable que el compilador asignase un valor por defecto al declararse la variable. En este caso, debería ser el conjunto vacío.

Tipo Registro

Un registro es una estructura de datos compuesta que permite agrupar elementos de diferentes tipos. Para definir un registro se utiliza la palabra reservada *record* seguida de una lista de declaraciones de campos y termina con la palabra reservada *end;*. Cada declaración de campos consiste en un nombre (identificador) seguido del delimitador de tipo seguido de un tipo primitivo. Concretamente la declaración de un registro sigue la siguiente sintaxis:

```
nombreTipoRegistro = record
      campo1 : TipoPrimitivo;
      campo2 : TipoPrimitivo;
      ...
      campoN : TipoPrimitivo;
end;
```

Donde `nombreTipoRegistro` será el identificador del tipo, y `campo1`, `campo2`, … `campoN`, representan los identificadores que reciben los distintos campos del registro, que *sólo pueden ser de tipos primitivos del lenguaje*. Para acceder a los campos de un registro se utiliza el operador de acceso a registro "." (punto). Una vez declarado un registro se pueden crear variables de ese tipo. El siguiente listado muestra un ejemplo de uso.

Listado 9. Ejemplo de uso de registros

```
type
      tipoPersona = record
        dni : integer;
        casado: boolean;
end;
var
      persona : tipopersona; //correcto, no case sentitive
begin
      persona.dni := integer;
      persona.casado := true;
      write(persona.dni);
end;
```

2.2.4 Declaraciones de variables

En PasQal es necesario que las variables estén declaradas antes de utilizarlas. Estas declaraciones pueden ser globales y locales. Como dijimos, las variables globales se declaran en su sección al inicio del programa, las variables locales en la sección correspondiente dentro de un subprograma.

El propósito de la declaración de variables es el de registrar identificadores dentro de los ámbitos de visibilidad establecidos por la estructura sintáctica del código fuente y tipificarlos adecuadamente para que sean utilizados dentro de éstos. Para declarar variables se utiliza la siguiente sintaxis, dentro de las áreas de declaración de variables de un programa:

```
nombre1 [, nombre2, nombre3...] : tipo
```

Donde `tipo` es el nombre de un tipo primitivo del lenguaje o definido por el usuario, `nombre1, nombre2,...` son identificadores para las variables creadas de ese tipo. Como se ve en la sintaxis, si se desean declarar varias variables de un mismo tipo en la misma línea ha de hacerse utilizando el delimitador ",".

A continuación se muestran algunos ejemplos de declaraciones de variables.

Listado 9. Ejemplos de declaración de variables

```
type
      tipoConjunto = set of 1..5;
var
      a:integer;
      a,b:integer;
      x,y:boolean;
      p,q:^integer;
      conjunto: tipoConjunto;
```

2.2.5 Declaración de subprogramas

En el lenguaje PasQal se pueden declarar subprogramas para organizar modularmente el código. Un subprograma es una secuencia de instrucciones encapsuladas bajo un nombre y, opcionalmente declarada con unos parámetros. En concreto, existen 2 tipos de subprogramas, procedimientos y funciones. A continuación describimos cada uno de ellos.

2.2.5.1 Procedimientos

Los procedimientos son rutinas encapsuladas bajo un nombre que realizan una determinada operación para que el programador las invoque, convenientemente parametrizadas, desde distintos puntos del programa. La sintaxis de un procedimiento es:

```
procedure nombre ([param11 , param12,..., param1N : tipo;
      param21, param22,..., param2N : tipo2;
      ...
      paramM1, paramM2,..., paramMN : TipoM)];
   [ const
        <declaraciones_de_constantes globales> ]
   [ type
        <declaraciones_de_tipos_de_datos globales> ]
```

```
[ var
     <declaraciones_de_variables globales> ]
[ <declaraciones_de_subprogramas> ]

begin
      <bloque_de_instrucciones>
end;
```

En esta definición los corchetes indican código opcional. Donde nombre es el nombre del procedimiento, paramIJ el nombre de cada uno de los parámetros y tipo1, tipo2,... TipoM los tipos de dichos parámetros. Como puede verse, la declaración de los parámetros formales es una lista de declaraciones de parámetros separadas por un delimitador de punto y coma. Cada declaración de parámetros tiene la forma de una lista de identificadores separados por comas, seguido del delimitador de tipo, dos puntos (:) y seguido de un nombre de tipo primitivo o un identificador de tipo compuesto. La forma de pasar parámetros es posicional y será explicada en este apartado más adelante. La declaración de parámetros es opcional. La cabecera de un procedimiento sin parámetros tiene esta estructura sintáctica:

```
procedure nombre();
```

Por su parte, La cabecera va seguida de un cuerpo con una estructura potencialmente idéntica a la del *program* (sección *const*, *type y var*, en este orden, subprogramas, y sentencias encerradas entre los delimitadores *begin* y *end*;). En el siguiente listado presentamos un ejemplo de procedimiento.

Listado 10 . Ejemplo de declaración de procedimientos

```
procedure incrementa (x:integer);
const uno = 1;
var  y:integer;
      procedure escribe (z:integer);
      begin
            write(z);
      end;
begin
      x := x + uno;
      escribe(x);
end;
```

2.2.5.2 Funciones

Las funciones son rutinas encapsuladas bajo un nombre que realizan un determinado cómputo y cuyo resultado devuelven al contexto de invocación. Como discutiremos más adelante dicho contexto suele ser una expresión. Desde el punto de vista sintáctico, una función sólo se diferencia de un procedimiento en dos aspectos: 1) después de la declaración de los parámetros le sigue un delimitador de tipo, dos puntos (:), seguido de un tipo primitivo (*incluidos punteros*) (las funciones NO pueden devolver estructuras de datos como resultado de su invocación) y 2) El *valor de retorno* es una variable con el nombre de la función. Esta variable no ha de ser declarada en la sección de declaración de variables. Si se declarara una con el mismo nombre debería darse un error. Es obligatorio que exista, al menos una, una sentencia de asignación a la variable de retorno en el cuerpo de una función. En caso de no existir debe generarse un error. La sintaxis de la *cabecera* en la declaración de una función es la siguiente:

```
function nombre (param11, param12,..., param1N : tipo1;
                 param21, param22,..., param2N : tipo2;
                 ...
                 paramM1, paramM2,..., paramMN : tipoM): tipo;
```

Donde nombre es el nombre de la función, paramIJ el nombre de cada uno de los parámetros, tipo1, tipo2,... tipoM y tipo, el tipo de retorno. Los tipos de dichos parámetros. Como en el caso de los procedimientos, el uso de parámetros en una función es opcional, así como la declaración de parámetros.

En el siguiente listado se presentan ejemplos de funciones.

Listado 11. Ejemplos de declaración de funciones

```
function suma (x,y:integer):integer;
begin
      suma:=x+y;
end;
function uno ():integer;
const numUno = 1;
begin
      uno:= numUno;
end;
function cierto:boolean;
begin
      cierto:=true;
```

```
end;

function resta (x,y:integer):integer;

{función errónea: no hay retorno}

begin

      x:=x-y;

end;

function resta2 (x,y:integer):integer;

{función errónea: variable local coincide con función}

var resta2;

begin

      resta2:=x-y;

end;
```

En el ejemplo anterior se puede ver que las últimas dos funciones son erróneas. En el caso de *resta*, no existe instrucción de retorno y en el caso de *resta2*, declara una variable (resta2) que colisiona con el nombre que la función.

2.2.5.3 Paso de parámetros a subprogramas

En PasQal es posible llamar a los subprogramas (procedimientos y funciones) pasando expresiones o referencias de alguno de los tipos definidos en la sección 2.2.3 como parámetros a un subprograma. El compilador debe asegurar en cualquier caso que el tipo, orden y número de parámetros actuales de la invocación de un subprograma coincidan con el tipo, orden y número de los parámetros formales definidos en la cabecera de la declaración de dicho subprograma. En caso de no ser así deberá emitirse un mensaje de error.

En concreto el paso de parámetros actuales a un subprograma puede llevarse a cabo de dos formas diferentes:

- **Paso por valor**. En este caso el compilador realiza una copia del argumento a otra zona de memoria para que el subprograma pueda trabajar con él sin modificar el valor del argumento tras la ejecución de la invocación. Los parámetros actuales pasados **no** pueden ser conjuntos ni registros completos.

- **Paso por referencia**. En este caso, el parámetro sólo podrá ser una referencia (no una expresión) y el compilador transmite al subprograma la dirección de memoria donde está almacenado el parámetro actual, de forma que las modificaciones que se hagan dentro de éste tendrán efecto sobre el argumento una vez terminada la ejecución del mismo. El paso de parámetros por referencia es utilizado para pasar argumentos de salida o de entrada / salida. Para indicar que un parámetro se pasa por referencia debe precederse su nombre con el la palabra reservada *var*. Se considera un error de

compilación pasar expresiones como argumentos actuales a una función donde se ha declarado un parámetro formal de salida o entrada/salida.

En el siguiente listado se incluyen ejemplos de subprogramas que utilizan paso por valor y por referencia:

Listado 12. Ejemplos de pasos de parámetros por valor y por referencia

```
program principal ();

var a,b:integer;

{paso por valor}

function incrementa (x:integer):integer;  // paso por valor

begin

  incrementa:=x+1;

end;

{paso por referencia}

procedure incrementaVar (var y:integer) // paso por referencia

begin

  y:=y+1;

end;

begin

  a := 1;

  b := incrementa (a); { a = 1, b = 2)

  incrementaVar (a); {a = 2}

end;
```

Es muy importante destacar que se debe dar soporte a la *recursividad* directa. Es decir, dentro del código de un subprograma puede aparecer una llamada a sí mismo provocando una nueva ejecución de su código. Sin embargo, la recursividad indirecta (aquellos casos en que un subprograma llama a otro y ésta al primero) no es preciso contemplarla.

2.2.6 Sentencias y Expresiones

El cuerpo de un programa o subprograma está compuesto por sentencias que, opcionalmente, manejan internamente expresiones. En este apartado se describen detalladamente cada uno de estos elementos. Sintácticamente cada bloque de código se organiza como una secuencia ordenada de sentencias separadas por delimitadores de punto y coma (;).

2.2.6.1 Expresiones

Una expresión es una construcción del lenguaje que devuelve un valor de retorno al contexto del programa donde aparece la expresión. Las expresiones no deben aparecer de *forma aislada* en el código. Es decir, han de estar incluidas como parte de una sentencia allá donde se espere una expresión. Desde un punto de vista conceptual es posible clasificar las expresiones de un programa en varios grupos:

Expresiones aritméticas

Las expresiones aritméticas son aquellas cuyo cómputo devuelve un valor de tipo entero al programa. Puede afirmarse que son expresiones aritméticas las constantes literales de tipo entero, las constantes simbólicas de tipo entero, los identificadores (variables o parámetros) de tipo entero y las funciones que devuelven un valor de tipo entero. Asimismo también son expresiones aritméticas la suma y resta de dos expresiones aritméticas. El operador @ también definirá una expresión aritmética, ya que devuelve un entero que representa la dirección de memoria de la variable a la que es aplicado. De la misma forma lo es el operador ^, ya que devolverá el contenido de la variable entera a la que apunta el puntero.

Expresiones lógicas

Las expresiones lógicas son aquellas cuyo cómputo devuelve un valor de tipo lógico al programa. Sintácticamente puede afirmarse que son expresiones lógicas las constantes literales de tipo lógico (valores de verdad true y false), las constantes simbólicas de tipo lógico, los identificadores (variables o parámetros) de tipo lógico y las funciones que devuelven un valor de tipo lógico. Asimismo también son expresiones lógicas la conjunción (and) y disyunción (or) de dos expresiones aritméticas. En PasQal se incluyen una serie de operadores relacionales que permite comparar expresiones aritméticas entre sí. El resultado de esa comparación es también una expresión lógica. Es decir, la comparación con los operadores >, <, = o <> de dos expresiones aritméticas es también una expresión lógica. Asimismo, la pertenencia a conjuntos (*in*) se considera una expresión lógica ya que devuelve un valor lógico.

Precedencia y asociatividad de operadores

Cuando se trabaja con expresiones aritméticas (o lógicas) es muy importante identificar el orden de prioridad (precedencia) que tienen unos operadores con respecto a otros y su asociatividad. La siguiente tabla resume la relación de precedencia y asociatividad y operadores (la prioridad decrece según se avanza en la tabla. Los operadores en la misma fila tienen igual precedencia).

Precedencia	Asociatividad
`. ( )`	Izquierdas
`and`	Izquierdas
`+ - or`	Izquierdas
`< > = <>`	No aplica

Para alterar el orden de evaluación de las operaciones en una expresión aritmética prescrita por las reglas de prelación se puede hacer uso de los paréntesis. (Como puede apreciarse los paréntesis son los operadores de mayor precedencia). Esto implica que toda expresión aritmética encerrada entre paréntesis es también una expresión aritmética. En el siguiente listado se exponen algunos ejemplos sobre precedencia y asociatividad y cómo la parentización puede alterar la misma. En el caso de la pertenencia a conjuntos *IN* irá siempre entre paréntesis, separándose de otras expresiones lógicas. El siguiente listado muestra algunos ejemplos.

Listado 13. Ejemplos de precedencia y asociatividad en expresiones

```
true or false and true { Devuelve true }

true or (false and true) { Devuelve true }

2 > 3 > 1 { Error }

(a in conjunto) and 2>3
```

Expresiones de acceso a campos de registros

Para acceder a los campos de un registro se utiliza el operador de acceso a registro "**.**" (punto). Dado que los campos de un registro no pueden ser a su vez registros previamente declarados no es posible concatenar el uso del operador de punto. En el apartado tipo registro se puede ver un ejemplo de estas expresiones.

Expresiones relativas a conjuntos

Podemos diferenciar tres tipos de expresiones relativas a conjuntos. Los rangos ([a..b]), la unión (+) y la pertenencia (IN).

Como se ha comentado anteriormente la pertenencia se considera una expresión lógica ya que devuelve ese tipo de valor. Se consideran expresiones de rango a la definición de un rango para conjuntos (ver tipo conjunto), que tienen la forma [a..b]. Estas expresiones sólo pueden ser usadas para asignar valores a un conjunto. Por último para la unión de conjuntos se utiliza el símbolo +, pero a diferencia de las expresiones aritméticas, se devolverá un nuevo conjunto resultado de la unión de otros dos. Se recomienda realizar estas comprobaciones en el análisis semántico, ya que los posibles errores serían de tipos.

Expresiones relativas a punteros

Dentro de este epígrafe se incluyen las expresiones con los símbolos @ y ^. Ya se ha indicado que ambas expresiones se consideran aritméticas al devolver un entero. Por lo tanto pueden aparecer en cualquier parte del código en la que se espere ese tipo de expresiones.

Llamadas a función

Para llamar a una función ha de escribirse su identificador indicando entre paréntesis los parámetros actuales de la llamada, que deben coincidir en número y tipo con los definidos en la declaración del subprograma. Los parámetros pueden ser referencias a variables, campos de registros, constantes o expresiones (ver apartado 2.2.5.3).

El uso de los paréntesis es siempre obligado. Así, si una función no tiene argumentos, en su llamada es necesario colocar unos paréntesis vacios () tras su identificador.

Nótese que **los procedimientos no son expresiones**. Por tanto, si se utilizan en el lugar de una expresión debe generarse un error. Esta comprobación es propia del análisis semántico.

En el siguiente listado se muestran algunos ejemplos.

Listado 14. Ejemplos de llamadas a funciones

```
x:=suma(a, b);

y:=suma(a, resta(b, c));{siendo resta una función}

a := funcion1();

b := suma (r.campo1, r.campo2);

c := suma (p^, q^); {corresponden a enteros}

e := suma (puntero1, puntero2); {corresponden a punteros}
```

Evaluación de expresiones

El modelo de evaluación de expresiones lógicas de PasQal es en cortocircuito o modo de *evaluación perezosa*. Este modo de evaluación prescribe que 1) si en la evaluación de una conjunción (`and`) de dos expresiones la primera de las evaluadas es falsa la otra no debe evaluarse y 2) si en la evaluación de una disyunción (`or`) de dos expresiones la primera de las evaluadas es verdadera la otra no debe evaluarse.

2.2.6.2 Sentencias

PasQal dispone de una serie de sentencias que permiten realizar determinadas operaciones dentro del flujo de ejecución de un programa. En concreto nos estamos refiriendo a las sentencias de asignación, las sentencias de control de flujo condicional e iterativo, y las sentencias de entrada salida.

Asignaciones

Las instrucciones de asignación sirven para asignar un valor a una variable, puntero, conjunto o campo de un registro. Para ello se escribe primero una referencia a alguno de estos elementos seguido del operador de asignación ":=" y a su derecha una expresión. El compilador deberá comprobar en primer lugar la compatibilidad entre el tipo de la expresión a la derecha del operador de asignación y el de la referencia a la izquierda. El operador de asignación no es asociativo. Es decir no es posible escribir construcciones sintácticas del estilo a := b := c. La sintaxis corresponde a:

```
ref := expresion;
```

Según el tipo de la variable ref la expresión deberá ser compatible con ese tipo. Por ejemplo variables lógicas sólo admiten valores lógicos mientras que conjuntos sólo admiten rangos o uniones de conjuntos. El siguiente listado muestra algunos ejemplos de uso de sentencias de asignación:

Listado 15. Ejemplos de uso de la sentencia de asignación

```
i := 3 + 7;

distinto := 3<>4;

juan.casado = false;

a = 2 + suma(2,2);

conjunto:= [2..3];

conjunto1 : = conjunto2 + conjunto3;

p:=@a;

p^:=3+b;
```

En PasQal no es posible hacer asignaciones a estructuras de forma directa. Así son errores de compilación sentencias como `registro1 := registro2;` y `conjunto1:=conjunto2;`

Sentencia de control de flujo condicional if – then – else

Esta sentencia permite alterar el flujo normal de ejecución de un programa en virtud del resultado de la evaluación de una determinada expresión lógica. Sintácticamente esta sentencia puede presentarse de esta forma:

```
if (expresiónLogica) then
  <<sentencias>>

[else
  <<sentencias>>]
```

Donde `expresionLogica` es una expresión lógica que se evalúa para comprobar si debe ejecutarse las sentencias o bloque de sentencias. La parte del *else* es opcional, es decir, puede o no aparecer al escribir esta sentencia. *Es necesario que `expresionLogica` esté siempre entre paréntesis*. <<sentencias>> puede corresponder a una sentencia o a varias. En caso de que sentencias esté compuesta por **varias sentencias** se ha de declarar un bloque con *begin y end;* esto no es necesario (aunque posible) si sólo existe una sentencia. El uso del punto y coma es obligatorio siempre tras la palabra reservada *end*.

El siguiente listado ilustra varios ejemplos de uso.

Listado 16. Ejemplos de sentencia if – then -else

```
{una sola sentencia}

if (a=b) then

      a:=a+1;


if (a=b) then
```

```
begin
      a:=a+1;
end;

{varias sentencias}
if (a=b) then
begin
   a:=a+1;
   b:=b+1;
end;

{uso de else}
if (esCierto=true) then
      valor:=true;
else
      valor:=false;

{varias sentencias en if y una sola en else}
if (esCierto=true)then
begin
      valor:=true;
      a:=a+1;
end;
else
      valor:=false;
if (a in set) then
      write ("a en conjunto set");
else
      write ("a no pertenece a set");
```

Este tipo de construcciones pueden anidarse con otras construcciones de tipo if-then-else o con otros tipos de sentencias de control de flujo que estudiaremos a continuación.

Sentencia de control de flujo condicional repeat-until

La sentencia `repeat` modifica el flujo normal de ejecución repitiendo un bloque de sentencias hasta que se cumpla cierta condición. La sintaxis de esta sentencia corresponde con:

```
repeat
      <<sentencias>>
until (condicionLogica);
```

Donde `condicionLogica` es la expresión lógica a evaluar al final de cada iteración y `sentencias` el bloque de sentencias a ejecutar en cada vuelta. Es decir, si la expresión lógica es cierta, se vuelve a ejecutar el bloque de sentencias. Este proceso se repite una y otra vez hasta que la condición sea falsa. La `condicionLogica` ha de aparecer siempre entre paréntesis y con un punto y coma al final. En el siguiente listado se muestra un ejemplo de este tipo de bucle.

Listado 17. Ejemplos de sentencia repeat

```
repeat
      a:=a+1;
until (a>10);


repeat
      a:=a+1;
      b:=b+2;
until (a>10);
```

La sentencia `repeat` permite que otras estructuras de control formen parte del bloque de sentencia a iterar de forma que se creen anidamientos. Hay que tener en cuenta también que *no se usan begin ni end para delimitar un bloque de sentencias*.

Sentencia de control de flujo iterativo for

La sentencia `for` es otra sentencia de control de flujo iterativo funcionalmente similar a la sentencia `repeat`. La estructura sintáctica de una sentencia `for` es:

```
for (variable := valor1 to valor2) do
      <<sentencias>>
```

Donde `variable` es el índice que regula la iteración, *valor1* indica el valor inicial del índice y *valor2* el final, es decir, cuando la variable alcance valor2 se saldrá del bucle. Ambos valores han de ser de tipo entero y pueden ser expresiones numéricas. *Variable* ha de ser de tipo entero y no puede ser una expresión. Los paréntesis entre las palabras reservadas for y do son

obligatorios. No es necesario que se realice la comprobación de que valor1 es menor que valor2.

Hay que hacer notar que no es posible declarar la variable indice dentro de la sentencia `for`. Como se ha repetido numerosas ocasiones la declaración de variables ha de ser anterior a la de las sentencias. En cada iteración del bucle *variable* incrementará su valor en uno de forma automática, no existe por tanto la necesidad de explicitarlo con una sentencia de asignación dentro del bucle. <<sentencias>> sigue las mismas reglas detalladas en el caso de if.

El siguiente listado muestra dos ejemplos de sentencia for

Listado 18. Ejemplos de sentencia for

```
for (a:=0 to 5) do

      b:=b+1;

x:= 2;

{en el ejemplo superior el for solo iterará sobre la sentencia
b:=b+1, cuando termine ejecutará la siguiente asignación x:=2}

for (a:=0 to x) do

begin

      b:=b+1;

      c:=c+1;

end;
```

La estructura `for` permite que otras estructuras de control formen parte del bloque de sentencia a iterar de forma que se creen anidamientos.

Debido a la estructura de esta sentencia es posible modificar el valor de la variable índice dentro del bloque de sentencias. Durante la ejecución de un programa se debería salir de este bucle en caso de que el valor de la variable índice sea superior o igual al de *valor2*. Es decir, **no** debe generarse un error (ni en compilación, ni en ejecución) en caso de sobrepasarse el valor final del bucle.

Sentencias de llamada a un procedimiento

Como ya se discutió con anterioridad, en PasQal es posible declarar subprogramas procedimientos para que realicen procedimientos de cómputo potencialmente reutilizables. La invocación de procedimientos es idéntica a la que se describió para el caso de las funciones (véase Llamadas a funciones).

Sentencias de Entrada / Salida

El lenguaje PasQal dispone de una serie de procedimientos predefinidos que pueden ser utilizados para emitir por la salida estándar (pantalla) resultados de diferentes tipos. Estas

funciones están implementadas dentro del código del propio compilador lo que implica 1) que son funciones especiales que están a disposición del programador y 2) constituyen palabras reservadas del lenguaje y por tanto debe considerarse un error la declaración de identificadores con dichos nombres. En concreto disponemos de 2 procedimientos. A continuación los detallamos.

- `write(parámetro)`. Este procedimiento puede mostrar por pantalla el valor de un parámetro, que puede ser un identificador o expresión, así como una constante literal de tipo cadena de texto. El ejemplo `write("Hola mundo");` mostrará el texto `Hola mundo.` `write(x);` mostrará el valor de la variable `x.` `write (a+2);` mostrará el resultado de sumar 2 al valor de a. Sólo se deben mostrar valores enteros o cadenas de texto. En caso de intentar mostrar un valor lógico, un conjunto, registro completo, etc., se debe generar un error.

  En caso de no recibir ningún parámetro no escribiría nada por pantalla, pero no sería un error. Este procedimiento sólo admite un único parámetro y por último, indicar que *no se debe generar un salto de línea* después de mostrar el resultado por pantalla (se considerará un error en la práctica) ya que de eso se encargará la siguiente instrucción.

- `writeln()`. Este es un procedimiento sin parámetros que provoca la impresión de la secuencia de escape de salto de línea por la salida estándar. Su único uso es de la forma `writeln();` Los paréntesis son obligatorios.

No han de considerarse el uso de caracteres especiales ni secuencias de escape.

2.3 Gestión de errores

Un aspecto importante en el compilador es la gestión de los errores. Se valorará la cantidad y calidad de información que se ofrezca al usuario cuando éste no respete la descripción del lenguaje propuesto.

Como mínimo se exige que el compilador indique el tipo de error: léxico, sintáctico o semántico. Debe indicarse obligatoriamente el número de línea en que ha ocurrido. Por lo demás, se valorarán intentos de aportar más información sobre la naturaleza del error, por ejemplo:

- Errores léxicos: Aunque algunos errores de naturaleza léxica no puedan ser detectados a este nivel y deban ser postergados al análisis sintáctico donde el contexto de análisis es mayor, en la medida de lo posible deben, en esta fase, detectarse el mayor número de errores. Son ejemplos de errores léxicos: literal mal construido, identificador mal construido, carácter no admitido, etc.

- Errores sintácticos: todo tipo de construcción sintáctica que no se ajuste a las especificaciones gramaticales del lenguaje constituye un error sintáctoco.

- Errores semánticos: Los errores motivados por la comprobación explícita de tipos de acuerdo al sistema de tipos de PasQal constituyen errores de carácter semántico. Algunos ejemplos de errores semánticos son: identificador duplicado, tipo erróneo de

variable, variable no declarada, subprograma no definido, campo de registro inexistente, campo de registro duplicado, demasiados parámetros en llamada a subprograma, etc.

El compilador generado deberá recuperarse de los errores sintácticos que se encuentre durante el proceso de análisis de un programa. Esto implica que deben utilizarse los mecanismos pertinentes para que cuando se encuentre un contexto sintáctico de error el compilador genere un mensaje y continúe con el análisis con el ánimo de emitir la mayor cantidad de mensajes de error posibles y no solamente el primero. Sin embargo, no se debe realizar una recuperación de errores a nivel léxico ni semántico. Así por ejemplo si el compilador encuentra un carácter extraño en el código fuente o un error de concordancia de tipos, éste emitirá un mensaje de error y abortará el proceso de compilación.

3 Descripción del trabajo

En esta sección se describe el trabajo que ha de realizar el alumno. La práctica es un trabajo amplio que exige tiempo y dedicación. De cara a cumplir los plazos de entrega, recomendamos empezar a trabajar lo antes posible y avanzar constantemente sin dejar todo el trabajo para el final. Se debe abordar etapa por etapa, pero hay que saber que todas las etapas están íntimamente ligadas entre sí, de forma que es complicado separar unas de otras. De hecho, es muy frecuente tener que revisar en un punto decisiones tomadas en partes anteriores, especialmente en lo que concierne a la gramática.

La práctica ha de desarrollarse en **Java** (se recomienda utilizar la última versión disponible)**.** Para su realización se usarán las herramientas JFlex, Cup y ENS2001 además de seguir la estructura de directorios y clases que se proporcionará. Más adelante se detallan estas herramientas.

En este documento no se abordarán las directrices de implementación de la práctica que serán tratadas en otro diferente. El alumno ha de ser consciente de que se le proporcionará una estructura de directorios y clases a implementar que ha de seguir fielmente.

Es responsabilidad del alumno visitar con asiduidad el *tablón de anuncios* del Curso Virtual, donde se publicarán posibles modificaciones a este y otros documentos y recursos.

3.1 División del trabajo

A la hora de desarrollar la práctica se distinguen **dos especificaciones** diferentes sobre la misma que denominaremos A y B. Cada una de ellas supone una carga de trabajo equivalente y prescribe la implementación de un subconjunto de la especificación descrita en los apartados anteriores de este documento

Para las entregas de febrero y junio, cada alumno deberá implementar *solamente* una de las dos especificaciones. La especificación que debe realizar depende de su número de DNI. Asi:

```
Si DNI es par → Especificación A

Si DNI es impar → Especificación B.
```

Sin embargo, si la práctica se entrega en la convocatoria de septiembre, el alumno deberá implementar las características correspondientes a ambas especificaciones. Es decir la entrega de septiembre consiste en el desarrollo de un compilador completo que responde a TODAS las características descritas en este documento.

El compilador debe respetar la descripción del lenguaje que se hace a lo largo de esta sección. Incorporar características de Pascal no contempladas en PasQal no sólo no se valorará, sino que se considerará un error y **puede suponer un suspenso en la práctica**.

A continuación se detallan las funcionalidades que incorporan cada una de las especificaciones A y B. Para cada funcionalidad, la "X" indica que esa característica debe implementarse mientras que el "-" indica que no debe implementarse.

Todas las funcionalidades que no se incluyan en esta tabla pertenecen a ambas especificaciones.

FUNCIONALIDAD		**A**	**B**
Tipos de datos	Conjuntos	X	-
	Registros	-	X
	Punteros	-	x
Paso de parámetros	Por valor	-	X
	Por referencia	X	-
Operadores aritméticos	+	X	-
	-	-	X
Operadores relacionales	<	X	-
	>	-	X
	=	-	X
	<>	X	-
Operadores lógicos	and	X	-
	or	-	X
Sentencias de control de flujo	repeat	X	-
	for	-	X

3.2 Entregas

Antes de empezar, nos remitimos al documento “Procesadores de Lenguajes – Normas, y evaluación” que podrá encontrar en el entorno virtual para más información sobre este tema. Es fundamental que el alumno conozca en todo momento las normas indicadas en dicho documento. Por tanto en este apartado se explicará únicamente el contenido que se espera en cada entrega.

3.2.1 Fechas y forma de entrega

Las fechas límite para las diferentes entregas son las siguientes:

Febrero	14 de febrero
Junio	13 de junio
Septiembre	12 de septiembre

En un plazo aproximado de dos días después de la entrega se publicará en el Tablón de Anuncios del Curso Virtual un listado con los alumnos cuyos trabajos se hayan entregado correctamente. Si algún alumno entregó la práctica y no aparece en ese listado, debe ponerse en contacto a lo largo del día y de forma urgente con el profesor responsable de la práctica (Emilio Lorenzo) para aclarar lo sucedido. No se admitirán reclamaciones sobre entregas pasado el día de publicación de la lista de entregas.

Para entregar su práctica el alumno debe acceder a la sección de la práctica del Curso Virtual. Si una vez entregada desea corregir algo y entregar una nueva versión, puede hacerlo las veces que sea necesario hasta la fecha límite. Los profesores no tendrán acceso a los trabajos hasta dicha fecha, y por tanto no realizarán correcciones o evaluaciones de la práctica antes de tener todos los trabajos. En ningún caso se enviarán las prácticas por correo electrónico a los profesores.

Puesto que la compilación y ejecución de las prácticas de los alumnos se realiza de forma automatizada, *el alumno debe respetar las normas de entrega indicadas en el enunciado de la práctica*.

Se recuerda que es necesario superar unas **sesiones de control obligatorias** a lo largo del curso para aprobar la práctica y la asignatura. Nos remitimos al documento de normas de la asignatura, dónde viene explicado las fechas y normativa a aplicar.

3.2.2 Formato de entrega

El material a entregar, mediante el curso virtual, consiste en un único archivo comprimido en formato **zip** cuyo nombre debe construirse de la siguiente forma:

Grupo de prácticas + "-" + Identificador de alumno + "." + extensión

(por ejemplo: a-cgonzalez21.zip)

Dicho archivo contendrá la estructura de directorios que se proporcionará en las directrices de implementación. Esta estructura debe estar en la raíz del fichero zip. **No** se debe de incluir dentro de otro directorio, del tipo, por ejemplo: “pdl”, “practica”, “arquitectura”, etc.

En cuanto a la memoria, será un breve documento llamado "memoria" con extensión .doc o .pdf y situado en el directorio correspondiente de la estructura dada.

El índice de la memoria para la entrega de febrero será:

Portada obligatoria (Modelo estará disponible en el curso virtual).

1. El analizador léxico

2. El analizador sintáctico

3. Conclusiones

4. Gramática

En este punto se ha de incluir un esquema con las producciones de la gramática generada

El índice para las entregas de junio y septiembre será:

Portada obligatoria

1. Cambios realizados en los analizadores léxico y sintáctico

Además se deben comentar decisiones de diseño que el alumno considere que deben ser aclaradas en su propuesta.

2. El analizador semántico y la comprobación de tipos

2.1. Descripción de la Tabla de Símbolos

Describir la solución tecnológica elegida para implementarla

3. Generación de código intermedio

3.1. Descripción de la estructura utilizada

4. Generación de código final

4.1. Descripción del registro de activación

5. Indicaciones especiales

En este punto se han de incluir aquellas apreciaciones que el alumno quiera hacer sobre su práctica. Por ejemplo partes incompletas, interpretaciones dudosas del enunciado y soluciones tomadas, apuntes sobre necesidades de la ejecución del compilador, etc.

6. Conclusiones

7. Gramática

   En este punto se ha de incluir un esquema con las producciones de la gramática generada, sin incluir acciones semánticas ni atributos.

En cada apartado habrá que incluir únicamente comentarios relevantes sobre cada parte y no texto "de relleno", de forma que la extensión de la memoria esté comprendida aproximadamente entre 2 y 5 hojas (sin incluir el esquema de las producciones de la gramática). En caso de que la memoria no concuerde con las decisiones tomadas en la implementación de cada alumno la práctica puede ser considerada suspensa.

3.2.3 Entrega de febrero

Habrá una primera entrega (obligatoria) en febrero, que cubrirá únicamente la parte de análisis léxico y sintáctico: sólo se pide que el compilador procese el archivo fuente e identifique y escriba por pantalla los errores léxicos y sintácticos encontrados en el archivo fuente.

3.2.3.1 Análisis léxico

Para realizar esta fase se usará la herramienta *JFLex*. El primer paso es familiarizarse con la herramienta a través de los ejemplos básicos proporcionados en la página de la asignatura y después realizar la especificación léxica del lenguaje, compilarla y probarla. En esta fase, es importante identificar el número de línea y columna en el que aparece un símbolo, de cara a proporcionar información de contexto, dentro del código fuente al analizador sintáctico. Al finalizar esta etapa, se debe obtener el código fuente en java de un scanner capaz de identificar todos los TOKENS de un programa fuente en el lenguaje pedido así como detectar los posibles errores léxicos que éste pudiera contener.

3.2.3.2 Análisis sintáctico

Para la etapa de análisis sintáctico se utilizará la herramienta *Cup*. En primer lugar hay que dedicar tiempo a escribir la gramática del lenguaje. Esta gramática es el eje de la práctica y debe estar cuidadosamente diseñada, abarcando todas las posibles sentencias que pueden aparecer en un programa fuente. Los errores en el diseño de esta gramática obligarán a volver sobre ella más adelante, lo que conlleva pérdidas de tiempo. Especial atención merece la precedencia de operadores, un problema que se puede solucionar utilizando la directiva "precedence" de Cup. El código Cup permite integrar el análisis léxico de JFlex, de forma que al finalizar esta etapa el compilador debe reconocer las sentencias del programa fuente y detectar posibles errores sintácticos, indicando por pantalla el número de línea en que ha ocurrido el error.

3.2.3.3 Comportamiento esperado del compilador en esta entrega

El compilador debe procesar archivos fuente. Si aparecen errores léxicos o sintácticos, debe notificarlos por pantalla. En caso contrario se emitirá un mensaje por pantalla indicando que el código fuente no tiene errores sintácticos y que el proceso de compilación ha terminado con éxito (pese a que aún no se haya generado un fichero de salida).

Debido a la complejidad de la segunda entrega se recomienda que se intente adelantar el trabajo todo lo posible, incluso empezar a implementar la tabla de símbolos, aunque esta parte no pertenezca a esta entrega.

3.2.4 Entrega de junio

Para la entrega de junio hay que completar el resto de las etapas comenzando por el análisis semántico, hasta llegar a la generación de código intermedio y código final. En ningún caso es necesario realizar optimización del código intermedio ni del código final generado. Es importante destacar que esta segunda parte de la práctica constituye una *carga de trabajo considerablemente mayor que la del primer cuatrimestre*.

3.2.4.1 Análisis semántico

En la fase de análisis semántico se debe asignar un significado a cada construcción sintáctica del código fuente. Esto se consigue, en primer lugar gestionando los elementos declarados por el programador tales como constantes, tipos, variables, procedimientos y funciones. Para llevar a cabo esta labor es preciso hacer uso de una colección de estructuras entre las que destaca la tabla de símbolos por su especial relevancia. La *tabla de símbolos* (TS) es una estructura disponible en tiempo de compilación que almacena información sobre los nombres definidos por el usuario en el programa fuente, llamados símbolos en este contexto teórico. Dependiendo del tipo de símbolo encontrado por el compilador durante el procesamiento del código fuente (constante, variable, función), los datos que deben ser almacenados por la TS serán diferentes. Por ejemplo:

- Para las constantes: nombres, tipo, valor…
- Para las variables: tipo, ámbito, tamaño en memoria...
- Para las funciones : parámetros formales y sus tipos, tipo de retorno,…

Otra estructura importante es la *tabla de tipos* (TT) cuya responsabilidad es mantener una definición computacional de todos los tipos (primitivos y compuestos) que están accesibles por el programador dentro de un programa fuente. Las entradas de la TS mantienen referencias a esta tabla para tipificar sus elementos (variables, constantes funciones y procedimientos).

El trabajo de la fase de análisis semántico consiste, básicamente, en implementar, dentro de las acciones java que Cup permite insertar entre los elementos de la parte derecha de la gramática, el sistema de tipos para el lenguaje. Esto requiere añadir en la TT una entrada por tipo primitivo, tipo compuesto y subprograma definido; añadir una entrada en la TS por cada símbolo declarado (variables, constantes y subprogramas) y finalmente comprobar en las expresiones el uso de elementos de tipos compatibles entre sí. Asimismo deben comprobarse que no existen duplicaciones entre símbolos que pertenezcan al mismo ámbito o nombres de tipos. Cualquiera de estas violaciones o error de tipos constituye un error semántico que debe comunicarse al usuario.

3.2.4.2 Generación de código intermedio

Esta fase traduce la descripción de un programa fuente expresado en un árbol de análisis sintáctico (AST) decorado en una secuencia ordenada de instrucciones en un código cercano al

ensamblador pero aún no comprometido con ninguna arquitectura física. En este sentido, las instrucciones de código intermedio son CUADRUPLAS de datos formadas, a lo sumo, por 1) un operador de operación, 2) dos operandos y 3) un operando de resultado. En esta fase, los operandos son referencias simbólicas a los elementos del programa (entradas de la TS), nombres de etiquetas o variables temporales que referencian "lugares" donde se almacenan resultados de cómputo intermedio. EN NINGUN CASO estos operandos son direcciones físicas de memoria. El trabajo de esta fase consiste, básicamente en insertar en las acciones semánticas de Cup las instrucciones java pertinentes para realizar la traducción de un AST a una secuencia de CUADRUPLAS. Al final de esta etapa ya no necesitaremos más herramientas, ni tampoco el programa fuente: tendremos una TS llena y una lista de cuádruplas que describen todo el contenido del programa fuente.

3.2.4.3 Generación de código final

Para generar código final se parte del código intermedio generado y de la TS. Esta etapa es la única que no coincide en el tiempo con las anteriores, ya que se realiza posteriormente a realizar el proceso de compilación. Su objetivo es el de convertir la secuencia de CUADRUPLAS generada en la fase anterior en una secuencia de instrucciones para una arquitectura real (en nuestro caso ENS2001). Este proceso requiere convertir cada operador en su equivalente en ENS2001 y en convertir las referencias simbólicas de los operandos en direcciones físicas reales. Nótese que en función del ámbito de las mismas (variables globales, locales, parámetros, temporales, etc.) el modo de direccionamiento puede ser diferente.

Una vez obtenido el código final, éste puede probarse utilizando la herramienta ENS2001 que permite interpretar el código generado. Así podremos ejecutar un programa compilado con nuestro compilador y probar su funcionamiento.

3.2.4.4 Comportamiento esperado del compilador en esta entrega

El compilador debe procesar archivos fuente y generar archivos ensamblador (ENS2001). Si los programas fuente incluyen errores (léxicos, sintácticos o semánticos), el compilador debe indicarlos por pantalla; en ese caso no se generará ningún código. Los programas en ensamblador generados por el compilador deben ejecutar correctamente con la configuración predeterminada de ENS2001 (crecimiento de la pila descendente). Esta ejecución es manual, es decir, el compilador del alumno no debe ejecutar la aplicación ENS2001, sino únicamente generar el código ensamblador. Posteriormente, el usuario arrancará ENS2001 y cargará el archivo ensamblador generado ejecutándolo.

3.2.4.5 Dificultades de esta entrega

Al abordar esta segunda etapa es muy importante dedicar tiempo a entender el proceso completo de compilación y ejecución, especialmente a la hora de distinguir dos conceptos fundamentales: tiempo de compilación y tiempo de ejecución. La diferencia es que en tiempo de compilación tenemos a nuestra disposición la tabla de símbolos, que nos dice, por ejemplo, en qué dirección de memoria está una determinada variable. Sin embargo, al ejecutar el programa ya no tenemos ningún apoyo más que el propio código generado, de forma que el código debe contener toda la información necesaria para que el programa funcione correctamente. Esto resulta especialmente complicado a la hora de manejar llamadas a funciones, especialmente si son llamadas recursivas; para mantener esta información en

tiempo de ejecución se reserva un espacio en memoria llamado *registro de activación*, que almacena elementos como la dirección de retorno, el valor devuelto, los parámetros y las variables locales.

El registro de activación y la gestión de memoria son probablemente los puntos más delicados y difíciles para el alumno, por lo que recomendamos abordarlos con cuidado y apoyarse en la teoría.

4 Herramientas

Para el desarrollo del compilador se utilizan herramientas de apoyo que simplifican enormemente el trabajo. En concreto, se utilizarán las indicadas en los siguientes apartados. Para cada una de ellas se incluye su página web e información relacionada. En el curso virtual de la asignatura pueden encontrarse una versión de todas estas herramientas junto con manuales y ejemplos básicos.

4.1 JFlex

Se usa para especificar analizadores léxicos. Para ello se utilizan reglas que definen expresiones regulares como patrones en que encajar los caracteres que se van leyendo del archivo fuente, obteniendo tokens.

Web JFlex: http://jflex.de/

4.2 Cup

Esta herramienta permite especificar gramáticas formales facilitando el análisis sintáctico para obtener un analizador ascendente de tipo LALR. Además Cup también permite asociar acciones java entre los elementos de la parte derecha de la gramática de forma que éstas se vayan ejecutando a medida que se va construyendo el árbol de análisis sintáctico. Estas acciones permitirán, de cara a la segunda entrega de la práctica implementar las fases de análisis semántico y generación de código intermedio.

Web Cup: http://www2.cs.tum.edu/projects/cup/

4.3 Ensamblador ENS2001

ENS2001 es una máquina virtual que proporciona un sencillo entorno de ejecución basado en registros, memoria direccionada a nivel de byte y un sencillo juego de instrucciones. Además proporciona 2 entornos (uno en línea de comandos y otro con interfaz gráfica de usuario) que permite ejecutar los programas escritos en ENS2001 y depurarlos convenientemente.

4.4 Jaccie

Esta herramienta consiste en un entorno visual donde puede especificarse fácilmente un analizador léxico y un analizador sintáctico y someterlo a pruebas con diferentes cadenas de entrada. Su uso resulta muy conveniente para comprender como funciona el procesamiento sintáctico siguiendo un proceso ascendente. Desde aquí recomendamos el uso de esta

herramienta para comprobar el funcionamiento de una expresión gramatical. Además, puede ayudar también al estudio teórico de la asignatura, ya que permite calcular conjuntos de primeros y siguientes y comprobar conflictos gramaticales. Pero **No es necesaria** para la realización de la práctica

4.5 Ant

Ant es una herramienta muy útil para automatizar la compilación y ejecución de programas escritos en java. La generación de un compilador utilizando JFlex y Cup se realiza mediante una serie de llamadas a clases java y al compilador de java. Ant permite evitar situaciones habituales en las que, debido a configuraciones particulares del proceso de compilación, la práctica sólo funciona en el ordenador del alumno.

Web Ant: http://ant.apache.org/

5 Ayuda e información de contacto

Es **fundamental y obligado** que el alumno consulte regularmente el Tablón de Anuncios de la asignatura, accesible desde el Curso Virtual para los alumnos matriculados. En caso de producirse errores en el enunciado o cambios en las fechas siempre se avisará a través de este medio. El alumno es, por tanto, responsable de mantenerse informado.

También debe estudiarse bien el documento que contiene **las normas y el calendario** de la asignatura, incluido en el Curso Virtual.

Se recomienda también la utilización de los foros como medio de comunicación entre alumnos y de estos con el Tutor de Apoyo en Red (TAR). Se habilitarán diferentes foros para cada parte de la práctica. Se ruega elegir cuidadosamente a qué foro dirigir el mensaje. Esto facilitará que la respuesta bien por otros compañeros, por el TAR o por el equipo docente sea más eficiente.

Esto no significa que la práctica pueda hacerse en común, por tanto **no debe compartirse código**. Se utilizará un programa de detección de copias al corregir la práctica, comparando los trabajos con los de este año y con anteriores.

El alumno puede plantear sus dudas al tutor del Centro o a los profesores. El profesor encargado de la práctica es Emilio J. Lorenzo Galgo (emiliojulio@lsi.uned.es) *El alumno debe comprobar si su duda está resuelta en la sección de Preguntas Frecuentes (FAQ)o en los foros de la asignatura antes de contactar con el tutor o profesor. Por otra parte, si el alumno tiene problemas relativos a su tutor o a su Centro Asociado, debe contactar con el coordinador de la asignatura Felisa Verdejo* (felisa@lsi.uned.es).