

DISEÑO DEL LENGUAJE

Procesadores de Lenguajes

Práctica 1. Diseño del Lenguaje

Tabla Descriptiva del Tipo de Lenguaje		
1ª Componente Lenguaje Base	A	Lenguaje PASCAL
	B	Lenguaje C
2ª Componente Idioma	A	Palabras reservadas en Castellano
	B	Palabras reservadas en Inglés
3ª Componente Estructura de Datos	A	Listas
	B	Arrays multidimensionales
	C	Conjuntos
	D	Pilas
4ª Componente Subprogramas	A	Funciones
	B	Procedimientos
5ª Componente Estructuras de Control	A	case o switch
	B	for (formato de PASCAL)
	C	repeat-until
	D	do-until

David Medina Godoy
asce@correo.ugr.es
Ana María Ruiz Jiménez
anamrj@correo.ugr.es
22/03/2013

ÍNDICE

<i>1 DESCRIPCIÓN DEL LENGUAJE.....</i>	<i>2</i>
<i>2 DESCRIPCIÓN FORMAL DE LA SINTAXIS DEL LENGUAJE EN BNF.....</i>	<i>6</i>
<i>3 DEFINICIÓN DE LA SEMÁNTICA EN LENGUAJE NATURAL.....</i>	<i>10</i>
<i>3.1 ESTRUCTURA DEL PROGRAMA.....</i>	<i>10</i>
<i>3.2 DECLARACIÓN DE VARIABLES.....</i>	<i>10</i>
<i>3.3 DECLARACIÓN DE SUBPROGRAMAS : PROCEDIMIENTOS.....</i>	<i>11</i>
<i>3.4 SENTENCIAS.....</i>	<i>12</i>
<i>3.5 EXPRESIONES.....</i>	<i>13</i>
<i>4 IDENTIFICACIÓN DE LOS TOKENS CON EL MÁXIMO NIVEL DE ABSTRACCIÓN.....</i>	<i>14</i>

1 DESCRIPCIÓN DEL LENGUAJE

A continuación se procede a describir el lenguaje asignado BBBBA.

1 Sintaxis inspirada en un lenguaje de programación C.

Esto significa que tomaremos las reglas sintácticas usadas por el lenguaje C como referencia para las instrucciones del lenguaje nuevo, respetando en todo momento los requerimientos impuestos al lenguaje. Sin embargo, el lenguaje diseñado requiere que las declaraciones estén enmarcadas, mientras que en los lenguajes de referencia no se permite enmarcar las declaraciones.

2 Palabras reservadas en Inglés.

3 Estructuras de datos consideradas como tipo elementales.

Además de tener en cuenta los tipos de datos *entero*, *real*, *carácter* y *booleano* consideramos el uso del tipo de dato estructurado *arrays 1D* y *2D* no permitiendo la declaración de manera recursiva mezclada para este tipo de dato.

Operaciones sobre arrays 1D y 2D :

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
suma	binario	+	dos arrays	$a1+a2$	Suma elemento a elemento
			array y valor	$a+x$	Suma de x con cada elemento
			valor y array	$x+a$	Suma de cada elemento con x
resta	binario	-	dos arrays	$a1-a2$	Resta elemento a elemento
			array y valor	$a-x$	Resta de cada elemento con x
producto	binario	*	dos arrays	$a1*a2$	Producto elemento a elemento
			array y valor	$a1*x$	Producto de cada elemento con x
			valor y array	$x*a1$	Producto de x con cada elemento

Práctica 1. Diseño del lenguaje

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
división	binario	/	dos arrays	a1/a2	División elemento a elemento
			array y valor	a1/x	División de cada elemento con x
multipl.	binario	**	dos arrays	a1**a2	Multiplicación de matrices

- Los identificadores deben ser declarados antes de ser usados.
- Para la suma, resta, producto y división entre arrays el tamaño de ambos debe ser el mismo.
- Para la multiplicación de arrays el número de columnas del primer operando debe coincidir con el número de filas del segundo.

Operaciones para los tipos de datos entero, real, carácter y booleano :

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
suma	binario	+	entero y entero	2+4	Suma los dos operandos
			real y real	2.3+3.5	Suma los dos operandos

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
resta	binario	-	entero y entero	2-4	Resta los dos operandos
			real y real	2.3-3.5	Resta los dos operandos
producto	binario	*	entero y entero	2*4	Multiplica los dos operandos
			real y real	2.3*3.5	Multiplica los dos operandos
división	binario	/	entero y entero	4/2	Divide los dos operandos
			real y real	2.3/3.5	Divide los dos operandos
and	binario	&&	booleano y booleano	a&&b	Realiza la operación and entre operandos
or	binario		booleano y booleano	a b	Realiza la operación or entre operandos
not	unario	!	booleano	!a	Aplica la operación not a un operando
xor	binario	¬	booleano y booleano	¬ a	Realiza la operación xor entre operandos

Sentencia de asignación para todos los tipos de expresiones :

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
Asignación	binario	=	Variable y valor x	a=2	Asigna a 'a' el valor 2, debe ser del mismo tipo
			Variable y variable	a=b	Asigna a 'a' el valor de 'b', debe ser del mismo tipo
			Variable y expresión	a=b+2	Asigna a 'a' el resultado de evaluar la expresión 'b+2'

- Estos operadores no permiten datos de distinto tipo por lo que comprueban los tipos.
- El lenguaje no realiza un typecasting automático.

Permitirá expresiones aritméticas y lógicas :

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
Menor que	binario	<	entero y entero	2<4	Devuelve el booleano true si el primer operando es menor que el segundo, y en otro caso devuelve false
			real y real	2.3<3.6	

Nombre	Tipo	Lexema	Operandos	Ejemplo	Resultado
Menor o igual que	binario	<=	entero y entero	2<=4	Devuelve el booleano true si el primer operando es menor o igual que el segundo, y en otro caso devuelve false
			real y real	2.3<=3.6	
Mayor que	binario	>	entero y entero	2>4	Devuelve el booleano true si el primer operando es mayor que el segundo, y en otro caso devuelve false
			real y real	2.3>3.6	
Mayor igual que	binario	>=	entero y entero	2>=4	Devuelve el booleano true si el primer operando es mayor o igual que el segundo, y en otro caso devuelve false
			real y real	2.3>=3.6	

Práctica 1. Diseño del lenguaje

Igual a	binario	==	entero y entero	2==4	Devuelve el booleano true si el primer operando es igual que el segundo, y en otro caso devuelve false
			real y real	2.3==3.6	
Distinto de	binario	!=	entero y entero	2!=4	Devuelve el booleano true si el primer operando es distinto del segundo, y en otro caso devuelve false
			real y real	2.3!=3.6	

El orden de precedencia de los operadores considerando de mayor a menor es :

- Operador “!”
- Operadores “**”
- Operadores “*”, “/”
- Operadores “+”, “-”
- Operadores “<”, “<=”, “>”, “>=”, “==”, “!=”
- Operadores “&&”, “||”, “¬”
- Operador “=”

Para los casos en los que se quiera aumentar la prioridad de un operador respecto a otro se utilizaran paréntesis.

Todos los operadores binarios tienen asociatividad por la izquierda, es decir, si encontramos una operación como $1 + 2 + 3$ se resuelve como $(1 + 2) + 3$.

4 Subprogramas.

Se usan *procedimientos* las cuales no devuelven ningún tipo de dato.

5 Estructuras de control.

Dispone de las estructuras de control siguientes :

- IF-THEN-ELSE
- WHILE
- SWITCH

2 DESCRIPCIÓN FORMAL DE LA SINTAXIS DEL LENGUAJE EN BNF

La estructura BNF que un programa debe cumplir se muestra a continuación.

El armazón principal sintáctica del programa es :

```
<Programa> ::= <Cabecera_programa> <bloque>

<Cabecera_programa> ::= main
```

Los bloques seguirán esta estructura, permitiendo anidamiento :

```
<bloque> ::= <Inicio_de_bloque>
            <Declar_de_variables_locales>
            <Declar_de_subprogs>
            <Sentencias>
            <Fin_de_bloque>

<Inicio_de_bloque> ::= '{'

<Fin_de_bloque> ::= '}'
```

Las declaraciones previas de variables :

```
<Declar_de_variables_locales> ::= <Marca_ini_declar_variables>
                                <Variables_locales>
                                <Marca_fin_declar_variables>
                                |

<Marca_ini_declar_variables> ::= begin_declare

<Marca_fin_declar_variables> ::= end_declare

<Variables_locales> ::= <Variables_locales> <Cuerpo_declar_variables>
                       | <Cuerpo_declar_variables>

<lista_variables> ::= <lista_variables> ',' <iden> | <iden>

<Cuerpo_declar_variables> ::= <Tipo_simple> <lista_variables> ;

<identificador> ::= <letra> {<caracter>}

<caracter> ::= <letra> | _ | <digito>

<Tipo_simple> ::= int | double | bool | char
```


La declaraciones de subprogramas :

```

<Declar_de_subprogs> ::= <Declar_de_subprogs> <Declar_subprog>
                        |
<Declar_subprog> ::= <Cabecera_subprograma> <bloque>
<Cabecera_subprograma> ::= void <identificador> '(' <Declar_parametros>
',)'
<Declar_parametros> ::= <mas_parametros> <Tipo_simple> <identificador> |
<mas_parametros> ::= <mas_parametros> <Tipo_simple> <identificador>, |

```

Las sentencias del programa :

```

<Sentencias> ::= <Sentencias> <Sentencia>
                | <Sentencia>
<Sentencia> ::= <bloque>
                | <sentencia_asignacion>
                | <sentencia_if>
                | <sentencia_while>
                | <sentencia_entrada>
                | <sentencia_salida>
                | <Procedimiento>
                | <sentencia_case>

<sentencia_asignacion> ::= <iden> '=' <Expresion>;'
<iden> ::= <identificador>
          | <identificador> '[' <expresion> ']'
          | <identificador> '[' <expresion> ',' <expresion> ']'
<Expresion> ::= ( Expresion )
                | <Op_unario> <Expresion>
                | <Expresion> <Op_binario> <Expresion>
                | <iden>
                | <Constante>
                | <Procedimiento>
                | <Agregados>

<Procedimiento> ::= <identificador> '(' <Lista_expresiones> ')'
<Agregados> ::= '{' <Lista_expresiones> '}'
<Lista_expresiones> ::= <Expresion>
                       | <Lista_expresiones> ',' <Expresion>

<sentencia_if> ::= <Alternativa_doble>
                 | <Alternativa_simple>
<Alternativa_simple> ::= if <Expresion> <Sentencia_o_bloque>

```

Práctica 1. Diseño del lenguaje

```
<Alternativa_doble> ::= if <Expresion> <Sentencia_o_bloque> else
<Sentencia_o_bloque>

<Sentencia_o_bloque> ::= <sentencia> | <bloque>

<sentencia_while> ::= while '(' <Expresion> ')' <Sentencia>

<sentencia_entrada> ::= read <lista_variables>

<sentencia_salida> ::= write <lista_expresiones_o_cadena>

<letra> ::= a | ... | z | A | ... | Z

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Constante> ::= <constante_entera>
                | <constante_real>
                | <constante_booleana>
                | \"<caracter>\"
                | <cadena>

<signo> ::= + | - |

<constante_entera> ::= <constante_entera> <digito>
                    | <digito>

<constante_real> ::= <constante_entera> . <constante_entera>
                    | <constante_entera> E <signo> <constante_entera>
                    | <constante_entera> . <constante_entera> E
                    | <signo> <constante_entera>

<constante_booleana> ::= true | false

<cadena> ::= \"<mas_caracteres> <caracter>\"

<mas_caracteres> ::= <mas_caracteres> <caracter> |

<Op_unario> ::= !

<Op_binario> ::= ** | + | - | * | / | && | '||' | '¬|' | !=
                | < | <= | > | >= | ==

<sentencia_case> ::= switch '(' <identificador> ')' '{' <casos>
<caso_por_defecto>}'
<casos> ::= <casos> case <opcion> : <Sentencias> break ';' |
<opcion> ::= <constante_entera> | <caracter>
<caso_por_defecto> ::= default ':' <Sentencias> ';' |
```


3 DEFINICIÓN DE LA SEMÁNTICA EN LENGUAJE NATURAL

3.1 ESTRUCTURA DEL PROGRAMA

La estructura seguida en nuestro lenguaje es la misma que en C. La estructura general será una cabecera seguida de un bloque.

La **cabecera** comienza con la palabra reservada **main**.

En el **bloque** estará definido el cuerpo del programa. Éste estará delimitado por el inicio del bloque “{” y fin del bloque “}”. El cuerpo del programa estará constituido por la declaración de variables, declaración de subprogramas y sentencias. A continuación se muestra un ejemplo de un programa.

Debido a que este lenguaje ofrece sensibilidad, el ejemplo de la derecha estará bien formado mientras que el de la izquierda no.

Estructura

```
MAIN {
  <Declar_de_variables_locales>
  <Declar_de_subprogs>
  <Sentencias>
}
```

Estructura

```
main {
  <Declar_de_variables_locales>
  <Declar_de_subprogs>
  <Sentencias>
}
```

3.2 DECLARACIÓN DE VARIABLES

La declaración de variables está delimitada por las marcas “**begin_declare**” y “**end_declare**”. Cada declaración de variable estará constituida por la especificación de tipo de dato al principio de la declaración seguida del identificador de la variable, una “,” si queremos definir otra variable de ese mismo tipo seguida de otro identificador de variable (esto se puede realizar tantas veces como variables del mismo tipo queramos definir) y se delimitará por un punto y coma “;” al final de la declaración.

Los tipos de datos posibles son los siguientes : **int** , **double** , **bool** , **char** .

Práctica 1. Diseño del lenguaje

También se pueden definir arrays 1D y 2D de los tipos anteriormente citados : **int [n]**, **int [n,m]**, **double [n]**, **double [n,m]** , **bool [n]**, **bool [n,m]** , **char [n]**, **char [n,m]** .

Para que los **identificadores** sean válidos deben comenzar con una letra y pueden estar seguidos de letras, dígitos o un “_”.

Debido a que este lenguaje ofrece sensibilidad, el ejemplo de la derecha estará bien formado mientras que el de la izquierda no.

Ejemplo de declaración de variables

```
BEGIN_DECLARE  
INT a_1 , A_2 , tres ;  
CHAR id1 ;  
Bool array1[3] , array2[2,2] ;
```

```
END_DECLARE
```

Ejemplo de declaración de variables

```
begin_declare  
bool b1[2,2], b2[3] ;  
double id1 ;  
int id2 ;
```

```
end_declare
```

Las variables definidas en el bloque del programa principal tendrán su ámbito en todo el bloque del programa principal, incluidos los subprogramas definidos en dicho bloque.

3.3 DECLARACIÓN DE SUBPROGRAMAS : PROCEDIMIENTOS

Se pueden declarar varios subprogramas dentro del bloque del programa . En nuestro caso, los subprogramas serán procedimientos, por lo que no devolverán ningún dato.

La cabecera del subprograma se define con la palabra reservada **void** , el **identificador del subprograma** que es lo que le da nombre al subprograma, **entre paréntesis** se puede definir opcionalmente una **lista de argumentos** siguiendo la estructura de la declaración del tipo de dato, a continuación el identificador que le da nombre, pudiendo repetirse la estructura separados entre un argumento y otro por una coma “,”. Tras la cabecera habrá un bloque, en el cual se define el cuerpo del subprograma.

Ejemplo de declaración de función

```
void un_procedimiento ( int arg1,int arg2,char arg_c, double [2,2])
{

    <Declar_de_variables_locales>
    <Declar_de_subprogs>
    <Sentencias>

}
```

El bloque del subprograma tendrá la misma estructura que el bloque del programa principal. Las variables definidas en el bloque de un subprograma tendrán su ámbito sólo dentro de éste subprograma.

3.4

SENTENCIAS

El conjunto de sentencias que soporta el lenguaje está formado por sentencias de asignación, if , while , de entrada (read) , de salida (write) , case. A continuación se comenta con más detalle cada una de ellas.

Las **sentencias de asignación** vienen dadas por un identificador de una variable, el símbolo “=” y un valor dado por una expresión. El tipo de dato del valor asignado debe coincidir con el tipo de dato de la variable.

La **sentencia if** comienza por “if” continuada de una <Expresion>, de inmediato <Sentencia> y opcionalmente la palabra reservada “else” ,seguida de una estructura de tipo <Sentencia>. <Expresion> debe ser de **tipo bool**. <Sentencia> puede ser una sola sentencia o un bloque que a su vez contiene varias sentencias (encerradas entre llaves).

La **sentencia while** tiene como estructura : comienza por **while (** <Expresion> **)** y por último viene una estructura de tipo <Sentencia>. Al igual que en las sentencias if, <Expresion> debe de ser de **tipo bool** y lo mismo ocurre con lo dicho con anterioridad sobre <Sentencia>.

La **sentencia de entrada** viene dada por la palabra reservada **read** y una lista de identificadores de variables en la que se almacenará la información leída desde teclado.

La **sentencia de salida** viene dada por la palabra reservada **write** y entre paréntesis una expresión o cadena o lista de expresiones o cadenas cuyo valor se imprimirá por pantalla.

La **sentencia case** sigue la siguiente estructura :

➤ La palabra reservada “**switch**” “(“ seguida de un identificador (el cual significa que el contenido de ese identificador será el que se compare para ver a que caso o bifurcación pertenece) , a continuación la palabra reservada “)” , una secuencia de “**case**” seguida de un identificador : a continuación <Sentencia> y las palabras reservadas “**break**” “;” , y de forma opcional podemos tener la secuencia “**default**” “:” <Sentencia> “**break**” “;”.

De la misma forma que en “if” y en “while”, <Sentencia> puede ser una sola sentencia o un bloque que contenga varias sentencias entre las palabras reservadas de principio y fin de bloque.

Este tipo de sentencia puede seguir la estructura de llevar una sola bifurcación explicada antes o una lista de estas bifurcaciones con la estructura dicha previamente.

Tenemos otra opción en la estructura de las sentencias de tipo case, esta es la mencionada antes pero añadiéndole una bifurcación llamada por defecto la cual se indica con la palabra reservada **default** (como es por defecto no aparecerá ningún identificador detrás de esta palabra) seguido de una <Sentencia> y concluyendo con la palabra “**break**” “;”.

3.5

EXPRESIONES

Las expresiones pueden ser identificadores de variables, constantes, llamadas a funciones y operaciones unarias, binarias sobre expresiones. Las expresiones se pueden anidar o agrupar usando paréntesis y se interpretarán con el orden de precedencia indicado en *La descripción del lenguaje*.

Las posibles **constantes** en el lenguaje serán **enteras**, **reales**, **lógicas** (true o false), **cadenas de caracteres** (secuencia de caracteres entre comillas dobles), **carácter** (carácter entre comillas simple) y **array constante** (array 1D y 2D de constante anteriores separadas por comas y entre llaves).

Las **llamadas a procedimientos** se identifican por medio del identificador del procedimiento entre paréntesis de forma opcional una expresión o lista de expresiones.

4 IDENTIFICACIÓN DE LOS TOKENS CON EL MÁXIMO NIVEL DE ABSTRACCIÓN

Identificaremos las palabras reservadas para después agrupar todas las que tengan un mismo significado sintáctico.

Token	Identificador	Atributos	Patrón / Expresión regular
MAIN	260		“main”
INICIO	261		“{”
FINBLO	262		“}”
INICIOV	263		“begin_declare”
FINV	264		“end_declare”
PYC	265		“;”
COMA	266		“,”
TIPOSIMPLE	267	1 : int 2 : double 3 : bool 4 : char	“int” “double” “bool” “char”
PROCED	268		“void”
CORIZ	269		“[”

Token	Identificador	Atributos	Patrón / Expresión regular
CORDER	270		“]”
ASIG	271		“=”
IF	272		“if”
ELSE	273		“else”
WHILE	274		“while”
READ	275		“read”

Práctica 1. Diseño del lenguaje

WRITE	276		"write"
CONSTANTE	277	1: [0-9]+.[0-9]* 2: [0-9]+ "E+" [0-9] + 3: [0-9]+ "E-" [0-9] + 4: [0-9]+.[0-9] "E+" [0-9] + 5: [0-9]+.[0-9] "E-" [0-9] + 6: " " [^" ' "] " " " 7: true 8: false	[0-9]+.[0-9]* [0-9]+ "E+" [0-9] + [0-9]+ "E-" [0-9] + [0-9]+.[0-9] "E+" [0-9] + [0-9]+.[0-9] "E-" [0-9] + " " [^" ' "] " " " "true" "false"
CADENA	278		\["^\"]+\\"
IDENTIFICA	279		[a-zA-Z] ([a-zA-Z] [0-9] "_")*
SWITCH	280		"switch"
CASE	281		"case"
DEFAULT	282		"default"
DOSP	283		","
OPU	284		"!"
OPB	285	1: ** 2: * 3: / 4: && 5: 6: ¬ 7: != 8: < 9: <= 10: > 11: >= 12: = 13: + 14: -	"**" "*" "/" "&&" " " "¬" "!=" "<" "<=" ">" ">=" "=" "+" "-"
PARDER	286)"
PARIZ	287		"("
LLAVEIZ	288		"{"
LLAVEDER	289		"}"
BREAK	290		"break"
SUMA_RESTA	291	1: + 2: -	+ -
CONSTANTE_E	292		[0-9]+
CHARACTER	293		' [^'] '