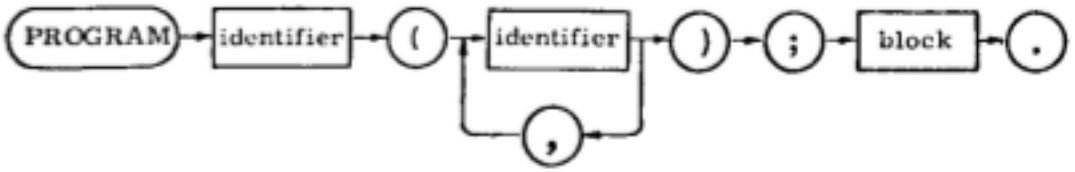
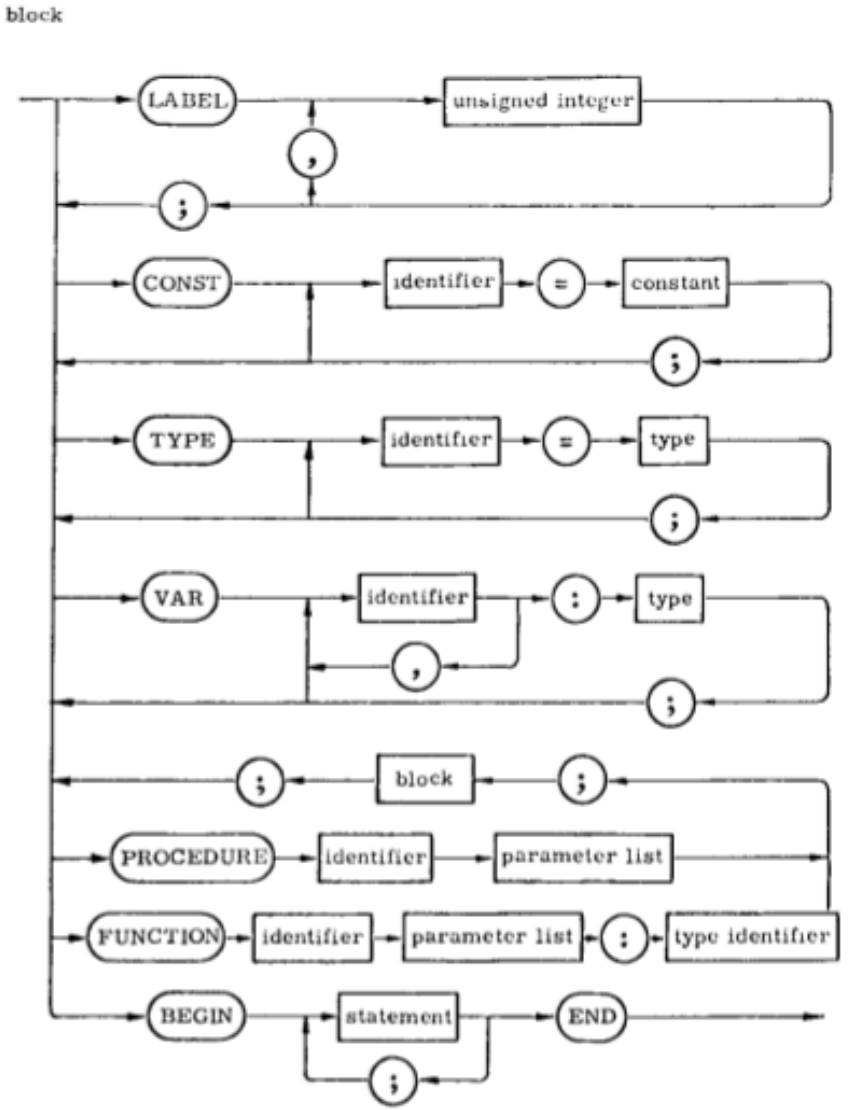
Programa en Pascal

program




```
<program> ::= <program heading> <block> .

<program heading> ::= program <identifier> ( <file identifier>
                                { , <file identifier> } );

<file identifier> ::= <identifier>

<identifier> ::= <letter> {<letter or digit>}

<letter or digit> ::= <letter> | <digit>
```

Programa en Pascal




```
<block> ::= <label declaration part> <constant definition part>
          <type definition part> <variable declaration part>
          <procedure and function declaration part>
          <statement part>

<label declaration part> ::= <empty> |
          label <label> {, <label>} ;

<label> ::= <unsigned integer>

<constant definition part> ::= <empty> |
          const <constant definition> {; <constant definition>} ;

<constant definition> ::= <identifier> = <constant>
```

⋮

Pascal Expresiones

<expresión> ::= <expresiónSimple> |
<expresiónSimple> <opRel> <expresiónSimple>
<opRel> ::= = | <> | < | <= | > | >= | in
<expresiónSimple> ::= <término> |
<signo> <término> |
<expresiónSimple> <opAditivo> <término>
<opAditivo> ::= + | - | or
<término> ::= <factor> |
<término> <opMult> <factor>
<opMult> ::= * | / | div | mod | and
<factor> ::= <variable> | <nro> | (<expresión>) | not <factor>

Operator precedence: The operator not (applied to a Boolean operand) has the highest precedence, followed by the multiplying operators (*. /, div, mod, and), then the adding operators (+, -, or), and of lowest precedence, the relational operators (=, <>, <, <=. >=, >, in). Any expression enclosed within parentheses is evaluated independent of preceding or succeeding operators.

Pascal Expresiones

<Expresión>
<ExpresiónSimple>
<Término>
<Término> <OpMult> <Factor> <--- **and es operador multiplicativo**
<Factor> <OpMult> <Factor> <--- **Factor no puede poner OpRel**
(<Expresión>) <OpMult> <Factor> <--- **A menos que use (Exp) que permite volver a ExpSimple**
(<ExpresiónSimple> <OpRel> <ExpresiónSimple>) <OpMult> <Factor>
(<Término> <OpRel> <ExpresiónSimple>) <OpMult> <Factor>
(<Factor> <OpRel> <ExpresiónSimple>) <OpMult> <Factor>
(<Variable> <OpRel> <ExpresiónSimple>) <OpMult> <Factor>
(A <OpRel> <ExpresiónSimple>) <OpMult> <Factor>
(A > <ExpresiónSimple>) <OpMult> <Factor>
(A > <Término>) <OpMult> <Factor>
(A > <Factor>) <OpMult> <Factor>
(A > <Nro>) <OpMult> <Factor>
(A > 24) <OpMult> <Factor>
(A > 24) and <Factor> <--- **Otra vez debo "escalar" a Exp para poder obtener OpRel**
(A > 24) and (<Expresión>)
(A > 24) and (<ExpresiónSimple> <OpRel> <ExpresiónSimple>)
(A > 24) and (<Término> <OpRel> <ExpresiónSimple>)
(A > 24) and (<Factor> <OpRel> <ExpresiónSimple>)
(A > 24) and (<Variable> <OpRel> <ExpresiónSimple>)
(A > 24) and (B <OpRel> <ExpresiónSimple>)
(A > 24) and (B < <ExpresiónSimple>)
(A > 24) and (B < <Término>)
(A > 24) and (B < <Factor>)
(A > 24) and (B < <Nro>)
(A > 24) and (B < 5)

Pascal: sentencias con booleanos

<sentencia if> ::= if <expresión> then <sentencia> |
if <expresión> then <sentencia> else <sentencia>

<sentencia while> ::= while <expresión> do <sentencia>

La <expresión> luego del if o el while debería ser booleanas, pero nada en la BNF lo indica

Restricciones agregadas

- Como vimos recién la BNF no siempre especifica completamente el lenguaje, ya sea para no aumentar innecesariamente la complejidad de la sintaxis o bien porque hay ciertos aspectos que son sensibles al contexto.
- La expresión “not 423” o “x or 15” son derivables, pero no son correctas porque 423 y 15 no son valores booleanos.
- Por eso se agregan restricciones, en lenguaje natural, para complementar los constructos de la BNF.
- Si una cadena es **derivable** de la BNF decimos que es **sintácticamente correcta**. Si no cumple con las restricciones adicionales decimos que no cumple con las restricciones sintácticas o más comunmente que no cumple con la semántica [estática] del lenguaje.

BNF lenguaje C

- Cambia la notación, según MORC (Manual de Referencia Oficial ANSI C)
  - Los no terminales van en cursiva (italic)
  - El metasímbolo de producción es :
  - Para indicar opciones pone una por renglón (en lugar de usar |)
  - Puede poner varias opciones en un renglón si se precede del metasímbolo *uno de* , usado generalmente para conjuntos de caracteres
  - Los terminales van en negrita (en K&R van en tipografía monoespaciada)
  - Si un término es opcional se lo indica con el subíndice *op*
  - No hay equivalente al cero o más repeticiones de la BNF usada en Pascal

Ejemplo

identificador :
nodígito
i*dentificador nodígito*
identificador dígito

nodígito: uno de
_ a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

dígito: uno de
0 1 2 3 4 5 6 7 8 9

Según el estándar las implementaciones deben distinguir los identificadores al menos hasta el carácter 31

Ejemplo de Código

ORIGINAL

```
 1  /* Basado en ejercicio 30 */
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #define MAX 10
 5  int main (int argc, char *argv[])
 6  {
 7      /* argv lo puedo ver como char **argv */
 8      char vec [MAX+1];
 9      int i;
10      if (argc > MAX) {
11          printf ("No se puede\n");
12      } else {
13          for (i=0; i < argc; i++)
14              vec[i] = argv[i][0];
15          vec[i] = '\0';
16          printf ("La cadena creada es %s\n", vec);
17      }
18      return EXIT_SUCCESS;
19  }
```

DESPUES DEL PREPROCESADOR

```
 492
 493  extern int printf (__const char *__restrict
__format, ...);
1853
1854  int main (int argc, char *argv[])
1855  {
1856
1857   char vec [10 +1];
1858   int i;
1859   if (argc > 10) {
1860    printf ("No se puede\n");
1861   } else {
1862    for (i=0; i < argc; i++)
1863     vec[i] = argv[i][0];
1864    vec[i] = '\0';
1865    printf ("La cadena creada es %s\n", vec);
1866   }
1867   return 0;
1868  }
```

Categorías Léxicas

- Lenguaje C usa varios LR a los cuales llama categorías léxicas o tokens
- Cada palabra de alguno de estos lenguajes se conoce como lexema
- En lenguaje C se reconocen los siguientes tokens
  - palabraReservada
  - identificador
  - constante
  - literalCadena
  - operador
  - caracterPuntuación

Palabras Reservadas
(Ansi C)

auto	double	int	short
break	else	struct	unsigned
case	enum	long	signed
char	extern	switch	void
const	float	register	sizeof
continue	for	typedef	volatile
default	goto	return	static
do	if	union	while

Palabras Reservadas (C 11)

Nuevo en estándar			
Ansi C	C99	C11	
auto	extern	short	while
break	float	signed	_Alignas
case	for	sizeof	_Alignof
char	goto	static	_Atomic
const	if	struct	_Bool
continue	inline	switch	_Complex
default	int	typedef	_Generic
do	long	union	_Imaginary
double	register	unsigned	_Noreturn
else	restrict	void	_Static_assert
enum	return	volatile	_Thread_local

Constantes

- Numéricas
  - Enteras
    - Decimales
    - Octales: comienza con 0
    - Hexadecimales: comienzan con 0x o 0X
    - Sufijos: `l L u U ll LL`
    - Sin sufijo son de tipo `int` (en principio)
  - Reales
    - El . decimal puede no tener dígitos adelante o atrás
    - Debo poner . decimal o exponente, uno de los dos es obligatorio
    - Sufijos: `l L f F`
    - Sin sufijo son de tipo `double`

Constantes

- De carácter
  - Delimitadas entre comillas simples `''`
  - Secuencias de escape
  - Son de tipo `int`
- De enumeración
  - Las que se definen con enum
  - Son de tipo `int`

Literales de cadena

- Se encierran entre comillas dobles
- Se puede aplicar \ para poder incluir una comilla doble
- El tipo de dato es `char[]` (generalmente usado como `char*`)
- Cadenas adyacentes se concatenan

  - Nota: tanto las constantes de carácter como los literales de cadenas pueden estar precedidos por el prefijo L (p.ej: L"cadena") que indica tipo wchar_t en lugar de char (es dependiente de la implementación, pero definido en **stddef.h** o wchar.h)

Operadores

Un operador especifica una operación a ser realizada: una evaluación para obtener un valor, un lvalue, un efecto lateral o una combinación de ellos

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Caracteres de Puntuación

- De acuerdo al estándar (Pto 6.1.6 y Anexo B.1.7) los posibles caracteres de puntuación son:
  [ ] ( ) { } * , : = ; ... #
- [ ] se utilizan en la declaración o definición de un arreglo
  Nota: al acceder a un elemento del arreglo es operador
- ( ) se utilizan para alterar precedencias o en la definición o declaración de funciones. También en es puntación después del while, for, etc.
  Nota: al invocar una función es operador.
- { } delimitan bloques
- ; transforma una expresión en sentencia
- , se utiliza para separar variables en su declaración o definición y para separar argumentos de una función.
  Nota: si concateno expresiones es operador

Caracteres de Puntuación

- * se usa en la declaración o definición de variables de tipo puntero.
  Nota: se usa como operador de desreferencia.
- : se usa en etiquetas, ya sea para goto o para los distintos casos dentro de un switch
- = se utiliza para inicializar variables al definirlas. También se usan en las enumeraciones si se quiere cambiar el valor de las constantes que se definen
- ... se utilizan para declarar o definir funciones con cantidad variable de argumentos
- # directiva del preprocesador

Detección de lexemas

```
/* fragmento de ejemplo*/
int mifunc(int a)
{
        int r,aux=1;
        r = aux+++a;  /*solo para mostrar +++ */
        return r;
}
```

int	palabraReservada	**,**	caracterPuntuación	**++**	operador
mifunc	identificador	**aux**	identificador	**+**	operador
(	caracterPuntuación	**=**	caracterPuntuación	**a**	identificador
int	palabraReservada	**1**	constante	**;**	caracterPuntuación
a	identificador	**;**	caracterPuntuación	**return**	palabraReservada
)	caracterPuntuación	**r**	identificador	**r**	identificador
{	caracterPuntuación	**=**	operador	**;**	caracterPuntuación
int	palabraReservada	**aux**	identificador	**}**	caracterPuntuación
r	identificador				

Categorías Gramaticales

- Son los LIC que se usan en la sintaxis del lenguaje
- Se dividen en
  - Expresiones
  - Declaraciones y Definiciones
  - Sentencias
- Todas se completan con restricciones agregadas.
  - Es más fácil agregar las restricciones al compilador que implementarlas como parte de la BNF

Expresiones

- Básicamente es un conjunto de operadores y operandos que:
  - Producen un valor, con el eventual uso de paréntesis para la precedencia
  - Designan un objeto o función
  - Generan un efecto lateral
  - O bien combinaciones de los anteriores
  - Las constantes, variables y llamados a funciones forman parte de las expresiones primarias o elementales

Expresiones - Booleanos

- Booleanos: en lenguaje C "*no hay*" valores booleanos, se considera, a efectos de evaluación, 0 como falso y distinto de cero como verdadero. Si una expresión es verdadera, devuelve un 1 como valor estándar de verdadero (tipo `int`).
- A partir de C99 se crea el tipo de datos `_Bool` que un tipo entero sin signo cuyo rango "puede ser menor al de otros tipos enteros"
- Para conveniencia de uso se definen en el encabezado `stdbool.h` `bool` como alias de `_Bool` y los valores `true` y `false`

Expresiones - ValorL

- Objeto: zona contigua de memoria donde se guarda un valor (de un determinado tipo)
- ValorL: designa un objeto, permite acceder a él
  - Nombre de una variable escalar: s
  - Elemento de un arreglo: v[i]
  - Vía un puntero: *p
- Las expresiones, su resultado, se los conoce también como valorR

Expresiones - evaluación

- Se evalúan siguiendo las precedencias y respetando los paréntesis, sin embargo no está definido un orden para:
  - Parámetros con los que se llama a una función
  - Efectos laterales, por ejemplo v[i++] = i
- Las precedencias se dan por la cercanía o lejanía con respecto al axioma
- La asociatividad está dada por la recursividad (a izquierda o a derecha) de la producción

Expresiones - operandos

- El orden de evaluación de los operandos **NO** está definido, salvo en los siguientes casos:
  - Operadores `&&` y `||` aseguran la evaluación de izquierda a derecha (para detener la evaluación en cuanto el resultado esté definido
  - Operador ternario `?:` asegura evaluar el primer operando y luego el segundo o tercero según corresponda
  - El operador coma garantiza que los expresiones (operandos de coma) se evalúan de izquierda a derecha, todas menos la última son evalúan como `void`, y el resultado es del tipo al que evalúe la última expresión.

Expresiones BNF

- En el libro dan una versión resumida de la BNF completa, que sirve para ilustrar los conceptos presentados (Ver pág 59 y siguientes)
- Notar como la primer producción menciona la asignación, es por tanto el de menor precedencia. El operador coma: **,** se eliminó para poder resumir.

Estándar

```
expression:
            assignment-expression
            expression , assignment-expression
```

Libro

```
expresión: expAsignación
```

Expresiones BNF

- Notar también que su recursión es a derecha, por eso los operadores de asignación asocian de derecha a izquierda

```
expAsignación: expCondicional
               expUnaria operAsignación expAsignación

operAsignación: uno de = +=
```

Expresiones BNF

- Ver que luego los operadores || y && tienen recursión a izquierda, por eso asocian de izquierda a derecha
  - Notar también que || viene antes que && por tener menor precedencia

```
expCondicional: expOr
                expOr ? expresión : expCondicional

expOr: expAnd
       expOr || expAnd

expAnd: expIgualdad
        ExpAnd && expIgualdad
```

Expresiones BNF

- Notar en la BNF que para invocar una función, la lista de argumentos es opcional, pero los paréntesis son obligatorios

```
expPostfijo: expPrimaria
             expPostfijo [ expresión ]
             expPostfijo ( listaArgumentos_op )

listaArgumentos: expAsignación
                 ListaArgumentos , expAsignación

expPrimaria: identificador
             constante
             literalCadena
             ( expresión )
```

Declaraciones y Definiciones

- Declarar implica dar a conocer, es decir especificar la interpretación y atributos de un identificador.
- Definir es declarar y además reservar almacenamiento.
- Ejemplo: `int mayor(int a, int b);` es una **declaración**. La **definición** sería, por ejemplo:

```
int mayor(int a, int b) {
    return a > b ? a : b;
}
```

- struct empleado `{int legajo, char nombre[30]};`
  Es una **declaración**
  `struct empleado e1;`
  Es una **definición**

Sentencias

- Lenguaje C divide las sentencias en
  - Sentencia compuesta (o bloque)
    - Entre { }
    - OJO: ANSI C solo permite declaraciones y definiciones al principio. Estándares posteriores si lo permiten.
  - Sentencia expresión
    - Expresión con ; al final. En general asignaciones
    - Puede ser nula (solo ;)
    - Si pones algo como 2*3; da warning unused value

Sentencias

- Lenguaje C divide las sentencias en
  - Sentencia de selección
    - **if**
      - El else es optativo, pero de estar empareja con el if más cercano
    - **switch**
      - Abajo de cada etiqueta solo puede haber una sentencia (no una declaración o definición de variables).

Sentencias

- Iteración
  - **while** itera mientras la expersión sea verdadera

```
while (exp)
   sentencia
```

  - **do while**: similar pero con la comprobación abajo

```
do
   sentencia
while (exp);
```

Sentencias

- Iteración
  - **for**: salvo que haya una sentencia continue

    `for (`exp_1 `;` exp_2`;` exp_3`)` sentencia

    es equivalente a

    ```
    exp1;
    while (exp2) {
       sentencia
       exp3;
    }
    ```

    - TODAS las exp_i son opcionales, 1 y 2 es evalúan como void si no están, en tanto que exp_3 se reemplaza por una constante verdadera si se lo omite.
    - for (;;) es loop infinito

Sentencias

- Salto
  - return exp_{op}
    - Sale de la función y retorna el control al código que invocó la función. Si la función devuelve un valor la exp_{op} es el valor retornado
  - goto etiqueta
    - Salto incondicional a la sentencia etiquetada con la etiqueta que acompaña al goto
  - break
    - Sale del ciclo iterativo o del switch que lo contiene
  - continue
    - Saltea lo que queda de esa iteración pero vuelve al control del ciclo y si corresponde sigue iterando

Licencia