UNIVERSIDAD DE CÓRDOBA
Escuela Politécnica Superior
Ingeniería informática
Procesadores de lenguajes
Víctor Monserrat Villatoro (i32moviv@uco.es)
Francisco José Rodríguez Ramírez (i32rodrf@uco.es)

INTÉRPRETE DE PSEUDOCÓDIGO EN ESPAÑOL

Especialidad: Computación
Tercer curso
Segundo cuatrimestre
Curso académico 2015-2016
Córdoba, agosto de 2016

1. Índice.
2. Introducción.
3. Lenguaje de pseudocódigo.
    3.1. Componentes léxicos.
        3.1.1. Palabras reservadas.
        3.1.2. Identificadores.
        3.1.3. Números.
        3.1.4. Cadenas.
        3.1.5. Operadores.
            3.1.5.1. Operador de asignación.
            3.1.5.2. Aritméticos.
            3.1.5.3. Alfanuméricos.
            3.1.5.4. Relacionales de números y cadenas.
            3.1.5.5. Lógicos.
        3.1.6. Comentarios.
        3.1.7. Fin de sentencia.
    3.2. Sentencias.
        3.2.1. Asignación.
        3.2.2. Lectura.
        3.2.3. Escritura.
        3.2.4. Sentencias de control.
        3.2.5. Comandos especiales.
4. Tabla de símbolos.
5. Análisis léxico.
6. Análisis sintáctico.
    6.1. Símbolos de la gramática.
        6.1.1. Símbolos terminales.
        6.1.2. Símbolos no terminales.
    6.2. Reglas de producción de la gramática.
    6.3. Acciones semánticas.
7. Funciones auxiliares.
8. Modo de obtención del intérprete.
9. Modos de ejecución del intérprete.
10. Ejemplos.
11. Conclusiones.

2. Introducción.

La práctica final de la asignatura consiste en la elaboración de un intérprete de pseudocódigo en español de forma individual o colectiva, como en nuestro caso. Para facilitarnos su elaboración se nos ha proporcionado por parte del profesor una serie de ejemplos. Aunque haya sido elaborado para reconocer lenguaje C, el ejemplo número nueve ha sido el de mayor ayuda y nos ha servido como base tras su adaptación para pseudocódigo. Hemos comenzado el desarrollo a partir de esta adaptación, a la que se le ha ido añadiendo funcionalidad según lo requerido.

El desarrollo del intérprete se ha realizado a través de las herramientas flex y bison ya usadas en la asignatura durante el curso.

En el presente documento se detalla su estructura, funcionamiento, algunos ejemplos y conclusiones sobre su elaboración. Además, el documento posee la estructura especificada por el profesor.

3. Lenguaje de pseudocódigo.

3.1. Componentes léxicos.

3.1.1. Palabras reservadas.

Las palabras reservadas son palabras que no podrán ser utilizadas como identificadores porque significan un comportamiento para el intérprete.

La lista de palabras reservadas en nuestro intérprete consta de las siguientes palabras: _mod, _div, _o, _y, _no, leer, leer_cadena, escribir, escribir_cadena, si, entonces, si_no, fin_si, mientras, hacer, fin_mientras, repetir, hasta, para, desde, paso, fin_para, _borrar.

El intérprete no distingue entre mayúsculas y minúsculas por lo que cualquier palabra de las anteriores podrá estar escrita de la

forma que sea, que el intérprete la reconocerá como palabra reservada.

3.1.2. Identificadores.

Los identificadores serán variables que use el usuario en la elaboración de un programa para almacenar datos.

Deben estar compuestos por una serie de letras, dígitos y subrayados pero deben comenzar por una letra y no pueden tener dos subrayados seguidos ni acabar con el símbolo de subrayado.

Además, no se distinguirá entre mayúsculas y minúsculas por lo que no existirán distintos identificadores con el mismo nombre pero diferente capitalización.

Ejemplos válidos de identificadores son: “dato, dato_1, dato_1_a”.

Ejemplos no válidos de identificadores son: “_dato, dato_, dato__1”.

3.1.3. Números.

Los números que pueden ser utilizados serán de tipo entero, real de punto fijo o real con notación científica pero todos son tratados conjuntamente como iguales.

3.1.4. Cadenas.

Las cadenas son una serie de caracteres delimitados por comillas simples que permiten la inclusión de otras comillas simples incluidas en la propia cadena a través del carácter de escape “\”. Las comillas simples delimitadoras de la cadena no serán incluidas en la propia cadena, su funcionamiento es simplemente delimitarla.

Ejemplos de cadenas son: ‘Ejemplo de cadena’, ‘Ejemplo de cadena con \’comillas\’ simples’.

3.1.5. Operadores.

3.1.5.1. Operador de asignación.

- El operador de asignación será “:=”.

3.1.5.2. Aritméticos.

- El operador de suma será “+” (unario y binario).
- El operador de resta será “-” (unario y binario).
- El operador de producto será “*”.
- El operador de división será “/”.
- El operador de división entera será “_div”.
- El operador de módulo será “_mod”.
- El operador de potencia será “**”.

3.1.5.3. Alfanuméricos.

- El operador de concatenación será “| |”.

3.1.5.4. Relacionales de números y cadenas.

- El operador menor que será “<”.
- El operador menor o igual que será “<=”.
- El operador mayor que será “>”.
- El operador mayor o igual que será “>=”.
- El operador igual que será “=”.
- El operador distinto que será “<>”.

Todos estos operadores pueden usarse tanto para variables numéricas como para variables alfanuméricas.

Ejemplos de expresiones relacionales son: “(A >= 0), (control <> ‘stop’)”.

3.1.5.5. Lógicos.

- El operador de disyunción lógica será “_o”.
- El operador de conjunción lógica será “_y”.
- El operador de negación lógica será “_no”.

Ejemplo de expresión lógica es: “(A >= 0) _y _no (control <> ‘stop’).

3.1.6. Comentarios.

- De varias líneas.

  Delimitados por el símbolo “#”.

  Ejemplo: “# ejemplo maravilloso
  de comentario
  de tres líneas #”.

- De una línea.

  Todo lo que sigue al carácter “@” hasta el final de línea.

  Ejemplo: “@ ejemplo espectacular de comentario de una línea”.

3.1.7. Fin de sentencia.

Para indicar el fin de una sentencia nuestro intérprete usará el símbolo de punto y coma (“;”).

3.2. Sentencias.

3.2.1. Asignación.

- identificador := expresión numérica.

  El intérprete declara identificador como una variable numérica y le asigna el valor de la expresión numérica.

  Las expresiones numéricas están formadas con números, variables numéricas y operadores numéricos.

- identificador := expresión alfanumérica.

  El intérprete declara identificador como una variable alfanumérica y le asigna el valor de la expresión alfanumérica.

  Las expresiones alfanuméricas están formadas con cadenas, variables alfanuméricas y operadores alfanuméricos.

3.2.2. Lectura.

- Leer (identificador).

  El intérprete declara a identificador como variable numérica y le asigna el número leído.

- Leer_cadena (identificador).

  El intérprete declara a identificador como variable alfanumérica y le asigna la cadena leída.

3.2.3. Escritura.

- Escribir (expresión numérica).

  El intérprete escribe en la pantalla el valor de la expresión.

- Escribir_cadena (expresión alfanumérica).

  El intérprete escribe en la pantalla el valor de la expresión alfanumérica teniendo en cuenta los caracteres especiales como “\t” y “\n”.

3.2.4. Sentencias de control.

- Condicional simple.

  si condición
  entonces sentencias
  fin_si

- Condicional compuesta.

  si condición
  entonces sentencias
  si_no sentencias
  fin_si

- Bucle “mientras”.

  mientras condición **hacer**
  sentencias
  fin_mientras

- Bucle “repetir”.

  repetir

sentencias
hasta condición

- Bucle “para”.

para identificador
desde expresión numérica 1
hasta expresión numérica 2
paso expresión numérica 3
hacer
sentencias
fin_para

En esta sentencia de control, el intérprete comprueba que no se produce un bucle infinito antes de ejecutarlo. Si el valor de la expresión numérica 3 es positivo, el valor de la expresión numérica 1 debe ser menor al de la expresión numérica 2. Por el contrario, si el valor de la expresión numérica 3 es negativo, el valor de la expresión numérica 1 debe ser mayor al de la expresión numérica 2. El paso siempre deberá ser o positivo o negativo pero nunca 0.

3.2.5. Comandos especiales.

- _borrar: borra la pantalla.
- _lugar(expresión numérica 1, expresión numérica 2):

Coloca el cursor de la pantalla en las coordenadas indicadas por los valores de las expresiones numéricas.

4. Tabla de símbolos.

La tabla de símbolos está declarada en symbol.c pero la estructura de la tabla de símbolos la tenemos en ipe.h.

```
typedef struct Symbol
          { /* elementos de la tabla de simbolos */
        char *nombre;
        short tipo; /* NUMBER,VAR,FUNCION,INDEFINIDA,CONSTANTE */
        short subtipo; /* NUMBER, CADENA */
        struct {
         double val;      /* VAR, NUMBER, INDEFINIDA, CONSTANTE */
         char *cad;         /* CADENA */
         double (*ptr)(); /* FUNCION  */
        } u;
                struct Symbol *siguiente;
      } Symbol;
```

Como podemos ver, la tabla de símbolos es una estructura que tiene 4 campos y otra estructura.

Los 4 campos que tiene son:

- char *nombre: almacena el nombre de la variable.
- short tipo: almacena el tipo de dato que tiene nuestra variable. Las opciones que vamos a tener son: NUMBER, VAR, FUNCION, INDEFINIDA y CONSTANTE.
- short subtipo: almacena el subtipo que tiene nuestro símbolo. Puede almacenar NUMBER o CADENA.
- struct Symbol *siguiente: es un puntero al siguiente símbolo.

Además, tiene una estructura dentro con 3 campos:

- double val: almacena el tipo de dato de nuestra variable: VAR, NUMBER, INDEFINIDA, CONSTANTE.
- char *cad: almacena la cadena de nuestra variable.
- double (*ptr)( ): almacena el puntero a una función.

5. Análisis léxico.

Nuestro análisis léxico se ha realizado en el fichero ipe.l y está compuesto por las siguientes expresiones regulares:

- numero: en esta parte se selecciona todo lo que tenga un número entre 0 y 9.

```
numero          [0-9]
```

Para seleccionar un número completo. Es decir, con decimales si es necesario, utilizamos la siguiente expresión:

```
{numero}+(\.{numero}+)?([eE][+-]?{numero}+)?
```

Si encontrara una expresión así, debemos de instalar el número en la tabla de símbolos:

```
{numero}+(\.{numero}+)?([eE][+-]?{numero}+)? {
  double d;
  sscanf(yytext,"%lf",&d);
  yylval.sym=install("",NUMBER,d);
  return NUMBER;
}
```

- letra: aquí almacenamos cualquier letra del abecedario, ya sea mayúscula o minúscula.

```
letra           [a-zA-Z]
```

- identificador: aquí se seleccionan los identificadores que nuestro intérprete considera correcto, que son aquellos mencionados en el apartado 3.1.2.

```
identificador {letra}(_?({letra}|{numero}))*
```

Además, si encuentra algún identificador lo tenemos que insertar en la tabla de símbolos.

```
{identificador} {
  Symbol *s;
  if ((s=lookup(yytext)) == 0)
    s = install (yytext, INDEFINIDA, 0.0);
  yylval.sym = s;
  return s->tipo == INDEFINIDA ? VAR : s->tipo;
}
```

- Cadenas: para el reconocimiento de cadenas necesitaremos utilizar distintos estados:
  - Reconocer una comilla simple ‘: se inicia el estado de la cadena y se reserva memoria para un elemento.

```
"\'" {
  BEGIN ESTADO_CADENA;
  cadena = (char *) malloc(sizeof(char));
  size = 0;
}
```

  - Reconocer \’: se reasigna memoria para poder albergar un carácter más y se inserta el carácter ‘ en la cadena.

```
<ESTADO_CADENA>"\\\'" {
  cadena = (char *) realloc(cadena, (size+1)*sizeof(char));
  cadena[size] = '\'';
  size++;
}
```

  - Reconocer \\n: si se reconoce el un \n en la cadena, se inserta ese carácter en la cadena después de reasignarle memoria.

```
<ESTADO_CADENA>"\\n" {
  cadena = (char *) realloc(cadena, (size+1)*sizeof(char));
  cadena[size] = '\n';
  size++;
}
```

- Reconocer \\t: si se reconoce el un \t en la cadena, se inserta ese carácter en la cadena después de reasignarle memoria.

```
<ESTADO_CADENA>"\\t" {
  cadena = (char *) realloc(cadena, (size+1)*sizeof(char));
  cadena[size] = '\t';
  size++;
}
```

- Reconocer cualquier otra cosa o un \n: se reasigna memoria a la cadena para almacenar un carácter más y se introduce en la cadena.

```
<ESTADO_CADENA>. {
  cadena = (char *) realloc(cadena, (size+1)*sizeof(char));
  cadena[size] = *yytext;
  size++;
}
```

- Reconocer de nuevo una comilla simple ‘: se acaba el estado de la cadena pero primero tenemos que reasignar memoria para un carácter más (el \0). Luego instalamos la cadena en la tabla de símbolos y liberamos la memoria de la cadena utilizada.

```
<ESTADO_CADENA>"\'" {
  cadena = (char *) realloc(cadena, (size+1)*sizeof(char));
  cadena[size] = '\0';
  yylval.sym=installcadena("",CADENA,cadena);
  free(cadena);
  BEGIN 0;
  return CADENA;
}
```

- Comentarios:

  - Una línea: en esta parte detectaremos los comentarios de una línea con la expresión:

```
"@".*
```

- Varias líneas: en esta parte detectaremos los comentarios de varias líneas con una serie de estados, que son:

  - Detectar el carácter '#':

```
"#" {
  BEGIN ESTADO_COMENTARIO;
}
```

  - Si encuentra cualquier carácter o un salto de línea, seguiría el comentario activo:

```
<ESTADO_COMENTARIO>.|\n {;}
```

  - Sin embargo, si se vuelve a encontrar otro '#', el comentario acabaría.

```
<ESTADO_COMENTARIO>"#" {
  BEGIN 0;
}
```

6. Análisis sintáctico.

El análisis sintáctico se realiza en el fichero ipe.y.

6.1. Símbolos de la gramática.

6.1.1. Símbolos terminales.

- Símbolos terminales de tipo: NUMBER, CADENA, VAR, CONSTANTE, INDEFINIDA.

- Símbolos terminales de secuencias de control: SI, ENTONCES, SI_NO, FIN_SI, MIENTRAS, HACER, FIN_MIENTRAS, REPETIR, HASTA, PARA, DESDE, PASO, FIN_PARA.

- Símbolos terminales de funciones: FUNCION0_PREDEFINIDA, FUNCION1_PREDEFINIDA, FUNCION2_PREDEFINIDA, LEER, LEER_CADENA, ESCRIBIR, ESCRIBIR_CADENA, _BORRAR, _LUGAR, +, -, *, /, _MOD, _DIV, CONCATENACION, _NO, **.

6.1.2. Símbolos no terminales.

- list: termina en épsilon, stmt o un error.
- stmt: termina en épsilon, asgn, escribir, escribir_cadena, leer, leer_cadena, _borrar, _lugar, mientras, si, repetir o para.
- asgn: asigna la variable o muestra un error.
- variable: define a una variable.
- cond: espera una expresión entre paréntesis.
- mientras: define a un bucle mientras.
- si: define a un condicional si.
- repetir: define al bucle repetir.
- para: define el término de parada de un bucle.
- end: termina en épsilon.
- stmtlist: es una lista de stmt.
- exp: evalúa los tipos de dato, funciones y operadores.

6.2. Reglas de producción de la gramática.

```
list :    /* nada: epsilon produccion */
        | list stmt  ';' {code(STOP); return 1;}
        | list error ';'   {yyerrok;}
        ;

stmt :    /* nada: epsilon produccion */  {$$=progp;}
        | asgn          {code(pop2);}
        | ESCRIBIR '(' expr ')'    {code(escribir); $$ = $3;}
        | ESCRIBIR_CADENA '(' expr ')'    {code(escribircadena); $$ = $3;}
        | LEER '(' VAR ')'    {code2(leervariable,(Inst)$3);}
        | LEER_CADENA '(' VAR ')'    {code2(leercadena,(Inst)$3);}
        | _BORRAR {code(borrar);}
        | _LUGAR '(' expr ',' expr ')' {code(lugar); $$ = $3; $$ = $5;}
        | mientras cond HACER stmtlist FIN_MIENTRAS end
                  {
                   ($1)[1]=(Inst)$4; /* cuerpo del bucle */
                   ($1)[2]=(Inst)$6; /* siguiente instruccion al bucle */
                  }
        | si cond ENTONCES stmtlist FIN_SI end /* proposicion if sin parte else
        */
                  {
                   ($1)[1]=(Inst)$4; /* cuerpo del if */
                   ($1)[3]=(Inst)$6; /* siguiente instruccion al if */
                  }
        | si cond ENTONCES stmtlist end SI_NO stmtlist FIN_SI end /*
        proposicion if con parte else */
                  {
                   ($1)[1]=(Inst)$4; /* cuerpo del if */
                   ($1)[2]=(Inst)$7; /* cuerpo del else */
                   ($1)[3]=(Inst)$9; /* siguiente instruccion al if-else */
                  }
```

```
59          | repetir stmtlist HASTA cond end
60                      {
61                        ($1)[1]=(Inst)$4; /* condicion */
62                        ($1)[2]=(Inst)$5; /* siguiente instruccion al repetir */
63                      }
64          | para variable DESDE expr end HASTA expr end PASO expr end HACER
            stmtlist FIN_PARA end
65                      {
66                        ($1)[1]=(Inst)$4;   /* desde */
67                        ($1)[2]=(Inst)$7;   /* hasta */
68                        ($1)[3]=(Inst)$10;  /* paso */
69                        ($1)[4]=(Inst)$13;  /* cuerpo del for */
70                        ($1)[5]=(Inst)$15;  /* siguiente instruccion al para */
71                      }
72          ;
73
74
75  asgn :    VAR ASIGNACION expr { $$=$3; code3(varpush,(Inst)$1,assign);}
76          | CONSTANTE ASIGNACION expr
77            {execerror(" NO se pueden asignar datos a constantes ",$1->nombre);}
78    ;
79
80  variable :    VAR {code((Inst)$1); $$ = progp;}
81          ;
82
83  cond :    '(' expr ')' {code(STOP); $$ =$2;}
84          ;
85
86  mientras:    MIENTRAS      {$$= code3(whilecode,STOP,STOP);}
87          ;
88
89  si:       SI          {$$= code(ifcode); code3(STOP,STOP,STOP);}
```

```
92  repetir:       REPETIR         {$$= code3(repetircode,STOP,STOP);}
93          ;
94
95  para:       PARA         {$$= code3(paracode,STOP,STOP); code3(STOP,STOP,STOP);}
96          ;
97
98  end :    /* nada: produccion epsilon */  {code(STOP); $$ = progp;}
99          ;
100
101 stmtlist:  /* nada: prodcuccion epsilon */ {$$=progp;}
102         | stmtlist stmt ';'
103         ;
```

```
105  expr :    NUMBER          {$$=code2(constpush,(Inst)$1);}
106          | CADENA          {$$=code2(cadenapush,(Inst)$1);}
107          | VAR             {$$=code3(varpush,(Inst)$1,eval);}
108          | CONSTANTE         {$$=code3(varpush,(Inst)$1,eval);}
109          | asgn
110          | FUNCION0_PREDEFINIDA '(' ')'      {code2(funcion0,(Inst)$1->u.ptr);}
111          | FUNCION1_PREDEFINIDA '(' expr ')'
             {$$=$3;code2(funcion1,(Inst)$1->u.ptr);}
112          | FUNCION2_PREDEFINIDA '(' expr ',' expr ')'
113                                              {$$=$3;code2(funcion2,(Inst)$1->u.pt
                                                 r);}
114          | '(' expr ')'    {$$ = $2;}
115          | expr '+' expr   {code(sumar);}
116          | expr '-' expr   {code(restar);}
117          | expr '*' expr   {code(multiplicar);}
118          | expr '/' expr   {code(dividir);}
119          | expr _DIV expr  {code(dividirenteros);}
120          | expr _MOD expr  {code(modulo);}
121          | expr POTENCIA expr  {code(potencia);}
122          | expr CONCATENACION expr   {code(concatenacion);}
123         |'-' expr %prec UNARIO  {$$=$2; code(negativo);}
124         |'+' expr %prec UNARIO  {$$=$2; code(positivo);}
125          | expr MAYOR_QUE expr   {code(mayor_que);}
126          | expr MAYOR_IGUAL expr {code(mayor_igual);}
127          | expr MENOR_QUE expr   {code(menor_que);}
128          | expr MENOR_IGUAL expr {code(menor_igual);}
129          | expr IGUAL expr   {code(igual);}
130          | expr DISTINTO expr  {code(distinto);}
131          | expr _Y expr  {code(y_logico);}
132          | expr _O expr  {code(o_logico);}
133          | _NO expr  {$$=$2; code(negacion);}
134          ;
```

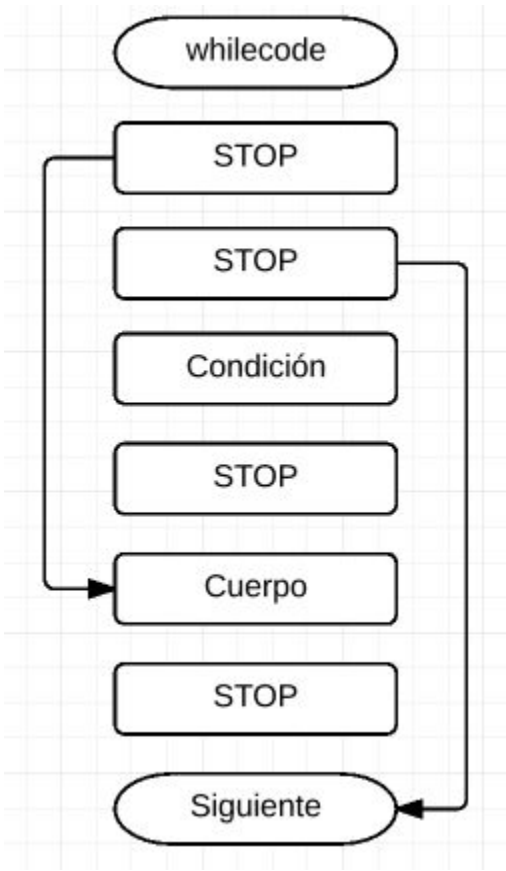
6.3. Acciones semánticas.

En las acciones semánticas tenemos dos secciones para cada acción. Lo primero que tenemos (la primera línea) son los elementos que conforman nuestra acción y lo siguiente que tenemos son los datos que se van almacenando en la pila.

Por cada línea (instrucción) se almacena un dato en la pila.

- Bucle MIENTRAS:

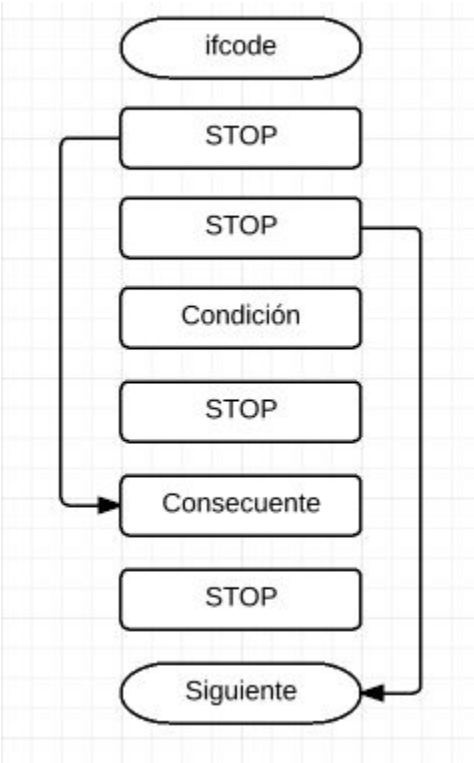
```
| mientras cond HACER stmtlist FIN_MIENTRAS end
        {
         ($1)[1]=(Inst)$4; /* cuerpo del bucle */
         ($1)[2]=(Inst)$6; /* siguiente instruccion al bucle */
        }
```



Aquí vemos que el primer STOP almacena un puntero al cuerpo del bucle y el segundo almacena un puntero a la siguiente instrucción.

- Condición SI:

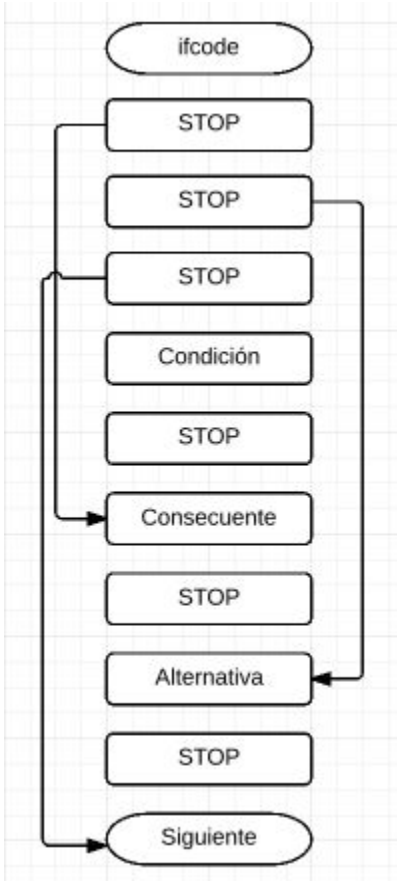
```
| si cond ENTONCES stmtlist FIN_SI end /* proposicion if sin parte else
*/
        {
         ($1)[1]=(Inst)$4; /* cuerpo del if */
         ($1)[3]=(Inst)$6; /* siguiente instruccion al if */
        }
```



Aquí vemos que el primer STOP almacena un puntero al consecuente de la condición y el segundo almacena un puntero a la siguiente instrucción.

- Condición SI SI_NO:

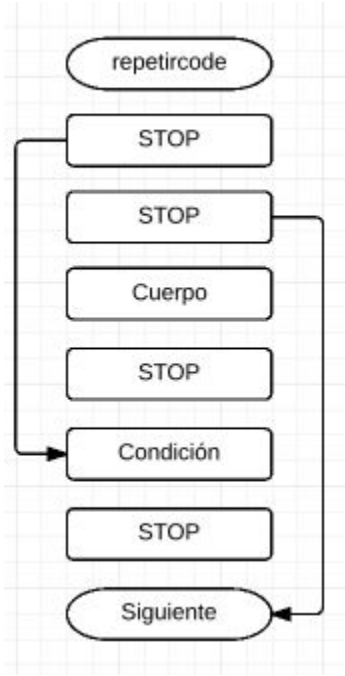
```
| si cond ENTONCES stmtlist end SI_NO stmtlist FIN_SI end /* proposicion
if con parte else */
            {
             ($1)[1]=(Inst)$4; /* cuerpo del if */
             ($1)[2]=(Inst)$7; /* cuerpo del else */
             ($1)[3]=(Inst)$9; /* siguiente instruccion al if-else */
            }
```



Aquí vemos que el primer STOP almacena un puntero al consecuente de la condición, el segundo almacena un puntero a la alternativa y el tercero a la instrucción siguiente instrucción.

- Bucle REPETIR:

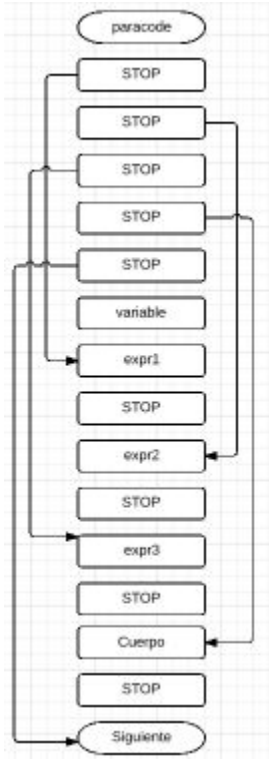
```
| repetir stmtlist HASTA cond end
          {
            ($1)[1]=(Inst)$4; /* condicion */
            ($1)[2]=(Inst)$5; /* siguiente instruccion al repetir */
          }
```



Aquí vemos que el primer STOP almacena un puntero a la condición del bucle y el segundo almacena un puntero a la siguiente instrucción.

- Bucle PARA:

```
| para variable DESDE expr end HASTA expr end PASO expr end HACER
stmtlist FIN_PARA end
          {
            ($1)[1]=(Inst)$4;   /* desde */
            ($1)[2]=(Inst)$7;   /* hasta */
            ($1)[3]=(Inst)$10;  /* paso */
            ($1)[4]=(Inst)$13;  /* cuerpo del for */
            ($1)[5]=(Inst)$15;  /* siguiente instruccion al para */
          }
```



Aquí vemos que el primer STOP almacena un puntero a la expr1, el segundo a la expr2, el tercero a la expr3, el cuarto al cuerpo del bucle y el quinto a la siguiente instrucción.

7. Funciones auxiliares.

Las funciones auxiliares tienen su declaración en el fichero ipe.h y la mayoría están programadas en el fichero code.c

- `void init();` Inserta en la tabla de símbolos las constantes y las funciones que se encuentran en las variables globales "consts" y "funciones".
- `void push(Datum d);` Mete el dato d en la pila.
- `extern Datum pop();` Saca un dato de la pila y lo devuelve.
- `extern void pop2();` Saca un dato de la pila y NO lo devuelve.

- `void initcode();` Inicializa la generación de código.
- `Inst *code(Inst f);` Instala una instrucción u operando.
- `void execute(Inst *p);` Ejecuta una instrucción.
- `extern void assign();` Asigna el valor superior al siguiente valor.
- `extern void constpush();` Mete una constante en la pila.
- `extern void cadenapush();` Mete una cadena en la pila.
- `void dividir();` Divide los dos valores superiores de la pila.
- `void dividirenteros();` Divide los dos valores superiores de la pila pero como enteros.
- `void escribir();` Saca de la pila el valor superior y lo escribe.
- `void escribircadena();` Saca de la pila el valor superior (cadena) y lo escribe.
- `void eval();` Evalúa una variable de la pila.
- `void funcion0();` Evalúa una función predefinida sin parámetros.
- `void funcion1();` Evalúa una función predefinida con un parámetro.
- `void funcion2();` Evalúa una función predefinida con dos parámetros.
- `void modulo();` Calcula el resto de la división entera del segundo valor de la pila por el valor de la cima.

- `void multiplicar();` Multiplica los dos valores superiores de la pila.
- `void negativo();` Niega el valor superior de la pila.
- `void positivo();` Toma el valor positivo del elemento superior de la pila.
- `void potencia();` Realiza la potencia de los valores superiores de la pila. Un valor será la base y el otro el exponente.
- `void restar();` Resta los dos valores superiores de la pila.
- `void sumar();` Suma los dos valores superiores de la pila.
- `void concatenacion();` Concatena los dos valores superiores de la pila (tienen que ser cadenas).
- `void varpush();` Mete una variable en la pila.
- `void ifcode();` Realiza una sentencia if.
- `void whilecode();` Realiza una sentencia while.
- `void repetircode();` Realiza una sentencia do while not.
- `void paracode();` Es la función que dice cuándo para un bucle o alerta de bucle infinito.
-
```
void mayor_que();
void menor_que();
void mayor_igual();
void menor_igual();
void igual();
void distinto();
```
Compara los dos valores superiores de la pila dependiendo de la función que se llame.

- `void y_logico();` `void o_logico();` `void negacion();` Realiza la operación lógica AND, OR o NOT.
- `void leervariable();` Lee una variable numérica por teclado.
- `void leercadena();` Lee una cadena por teclado.
- `void borrar();` Llama a la macro BORRAR y borra la pantalla.
- `void lugar();` Llama a la macro LUGAR(x,y) y pone el texto en el lugar(x,y).

8. Modo de obtención del intérprete.

El intérprete se obtiene mediante la ejecución de un fichero Makefile que contiene las instrucciones necesarias para realizar la compilación completa del intérprete.

```
1  FUENTE = ipe
2  LEXICO = ipe
3
4  CC = gcc
5  CFLAGS = -c -g
6  YFLAGS = -d          # Generar el fichero $(FUENTE).tab.h
7  LFLAGS = -lm -lfl    # fl: biblioteca de flex; m: biblioteca matemáticalfl
8  OBJS= $(FUENTE).tab.o lex.yy.o  init.o   math.o   symbol.o  code.o
```

Podemos observar que en las 8 primeras líneas del Makefile se establecen una serie de macros.

```
$(FUENTE).exe: $(OBJS)
  $(CC) $(OBJS) $(LFLAGS) -o $(FUENTE).exe

code.o:  code.c $(FUENTE).h
  $(CC) -c code.c

init.o:  init.c $(FUENTE).h $(FUENTE).tab.h
  $(CC) -c init.c

symbol.o:  symbol.c $(FUENTE).h $(FUENTE).tab.h
  $(CC) -c symbol.c

math.o:  math.c $(FUENTE).h
  $(CC) -c math.c

lex.yy.o:  lex.yy.c $(FUENTE).tab.h $(FUENTE).h macros.h
  $(CC) -c lex.yy.c

lex.yy.c: $(LEXICO).l $(FUENTE).tab.h $(FUENTE).h macros.h
  flex $(LEXICO).l

$(FUENTE).tab.o:  $(FUENTE).tab.c $(FUENTE).tab.h $(FUENTE).h
  $(CC) -c $(FUENTE).tab.c

$(FUENTE).tab.c $(FUENTE).tab.h:  $(FUENTE).y $(FUENTE).h
  bison $(YFLAGS) $(FUENTE).y
```

En esta parte del Makefile es donde se van ejecutando los pasos de la compilación para poder generar el intérprete final ‘ipe.exe’.

```
#Opcion para borrar los ficheros objetos y auxiliares
clean:
  rm -f  $(OBJS) $(FUENTE).tab.[ch] lex.yy.c $(FUENTE).exe $(FUENTE).output
```

Finalmente tenemos una opción para borrar los archivos no fuente.

Los ficheros que se han ido utilizando a lo largo de la compilación son:

- init.c: contiene el prototipo de dos funciones matemáticas, estructuras donde se almacenan palabras clave y una función (void init( )) que instala en la tabla de símbolos las constantes y funciones que se encuentran en las variables globales “consts” y “funciones”.

- ipe.l: es el archivo que contiene el analizador léxico con las expresiones regulares correspondientes que se necesitan para ello.

- ipe.h: este archivo contiene el prototipo de una serie de funciones de error, la función init( ) del archivo 'init.c'. También contiene la estructura de los símbolos y los datos con todos sus componentes.
  Por último, vemos que contiene el prototipo de las funciones que se utilizarán a lo largo del intérprete para leer cadenas, escribirlas, sumar, restar, etc.

- ipe.y: es el archivo que compila bison. Este archivo tiene todas las reglas de producción necesarias además de los tokens que se deben de usar.
  También tiene las funciones de error.

- macros.h: se definen una serie de macros de pantalla en este fichero.

- math.c: este fichero contiene funciones matemáticas.

- symbol.c: este fichero tiene las funciones con las que se accede a la tabla de símbolos, además de la propia tabla de símbolos.

  Estas funciones son:

  - lookup: inserta una palabra en la tabla de símbolos, indicando el token que le corresponde y su valor inicial. Devuelve un puntero al nodo que contiene a la palabra.

  - install: Inserta una palabra (entero) en la tabla de símbolos, indicando el token que le corresponde y su valor inicial. Devuelve un puntero al nodo que contiene a la palabra.

  - installcadena: igual que 'install' pero esta instala un char*.

  - emalloc: reserva memoria dinámica y comprueba que no se producen errores. Devuelve un puntero a la memoria reservada.

- code.c: este fichero contiene las funciones de los prototipos del fichero ipe.h programadas en C.

9. Modos de ejecución del intérprete.

- Modo interactivo.

  El modo interactivo del intérprete permite introducir sentencias que se irán interpretando según lo establecido en el lenguaje, normalmente cuando aparezca el símbolo de fin de sentencia aunque no sea así por ejemplo en las sentencias de control.

  Para ejecutar el modo interactivo, ejecutamos el programa “ipe.exe” sin argumentos.

```
variable := 3;
si (variable > 0)
        entonces
                escribir_cadena('La \'variable\' es -->\t positiva\n');
si_no
        escribir_cadena('La \'variable\' es -->\t negativa\n');
fin_si;
```

  Al introducir este trozo de código, el intérprete nos muestra lo siguiente:

```
La 'variable' es -->     positiva
```

- A partir de un fichero.

  Para ejecutar el intérprete a partir de un fichero es tan simple como ejecutarlo con el nombre del fichero deseado como único argumento.

```
./ipe.exe ejemplo_1_saluda.e
```

  El fichero será leído e interpretado por nuestro intérprete.

10. Ejemplos.

- Secuencia de Fibonacci.

```
first := 0;
second := 1;
escribir_cadena ('Introduce el número de terminos para Fibonacci: ');
leer(n);
escribir_cadena ('Los primeros ');
escribir (n);
escribir_cadena (' números de la serie son: \n');
para c desde 0 hasta n-1 paso 1 hacer
    si (c <= 1)
        entonces
        next := c;
    si_no
        next := first + second;
        first := second;
        second := next;
    fin_si;
    escribir (next);
    escribir_cadena ('\n');
fin_para;
```

```
Introduce el número de terminos para Fibonacci: 10
Los primeros 10 números de la serie son:
0
1
1
2
3
5
8
13
21
34
```

- Concatenación.

```
escribir_cadena ('Introduce tu nombre: ');
leer_cadena (nombre);
escribir_cadena ('Introduce tu primer apellido: ');
leer_cadena (apellido1);
escribir_cadena ('Introduce tu segundo apellido: ');
leer_cadena (apellido2);
escribir_cadena ('Tu nombre completo es: ' || nombre || ' ' || apellido1 || ' '
|| apellido2 || '\n');
```

```
Introduce tu nombre: Víctor
Introduce tu primer apellido: Monserrat
Introduce tu segundo apellido: Villatoro
Tu nombre completo es: Víctor Monserrat Villatoro
```

- Media.

```
escribir_cadena ('Introduce el número de términos para hacer la media: ');
leer(n);
valorFinal := 0;
para i desde 1 hasta n paso 1 hacer
  escribir_cadena ('elemento ');
  escribir (i);
  escribir_cadena (': ');
  leer(valor);
  valorFinal := valorFinal + valor;
fin_para;
escribir_cadena ('La media de tus elementos es ');
escribir (valorFinal/n);
escribir_cadena ('\n');
```

```
Introduce el número de términos para hacer la media: 5
elemento 1: 3
elemento 2: 9
elemento 3: 6.5
elemento 4: 4.5
elemento 5: 7
La media de tus elementos es 6
```

- División, cociente y resto.

```
escribir_cadena ('Introduce un número: ');
leer (n1);
escribir_cadena ('Introduce otro número: ');
leer (n2);
escribir (n1);
escribir_cadena ('/');
escribir (n2);
escribir_cadena (' = ');
escribir (n1/n2);
escribir_cadena ('\nEl cociente de la división = ');
escribir (n1 _div n2);
escribir_cadena (' y el resto ');
escribir (n1 _mod n2);
escribir_cadena ('\n');
```

```
Introduce un número: 8
Introduce otro número: 3
8/3 = 2.6666667
El cociente de la división = 2 y el resto 2
```

11. Conclusiones.

Es un trabajo entretenido porque vas viendo como de la nada, tu programa va creciendo y desarrollándose hasta convertirse en un intérprete bastante completo. Es muy útil, tanto para esta asignatura porque ayuda a reforzar los conocimientos que se adquieren en ella, como para el resto porque te ayuda a comprender mejor cómo funciona la interpretación de un lenguaje por lo que te ayuda a entender la programación en sí.