Universidad de La Laguna

E. T. S. DE INGENIERÍA INFORMÁTICA
Departamento de E.I.O. y Computación
Compiladores

PRACTICA 10: Generación de código intermedio para el compilador de *Pascal-*

Factor de ponderación [0-10]: 10

10.1. Objetivos

Implementar en el compilador de *Pascal-* la generación de código para una máquina orientada a pila. La generación de código se divide en dos partes:

1. Una extensión del analizador sintáctico que emite código para una P-Máquina.
2. Un ensamblador que define los valores de las etiquetas.

10.2. Implementación

El juego de instrucciones de la P-Máquina es el que se presenta en el Código 10.1.

`DEFADDR` y `DEFARG` son pseudoinstrucciones que se utilizan sólo durante la generación de código, no durante la ejecución del programa.

En la descripción de la práctica, con frecuencia se hace referencia de forma indistinta al analizador sintáctico y al generador de código. Si bien estos dos módulos, conceptualmente son diferentes, vamos a implementar el generador de código incrustado en el análisis sintáctico. De ahí que a veces no se diferencie entre ambos. El analizador sintáctico se ha de ampliar con una serie de funciones que permitirán la emisión de código. Por ejemplo, el analizador sintántico emitirá una instrucción de la forma:

```
VARIABLE level displ
```

por medio de una llamada a la función:

```
emit3(VARIABLE2, level, displ)
```

que tiene una forma similar a la que se muestra en el Código 10.2. Las restantes funciones que se utilizan para emitir instrucciones son similares a las que se muestran en el Código 10.3. El generador de código intermedio (incrustado en en analizador sintáctico) emite código hacia un fichero temporal (temp). Posteriormente, el ensamblador realizará dos pasadas sobre este fichero: en la primera realizará el cálculo de las etiquetas, y en la segunda generará el código definitivo. Estos estados de la emisión de código vienen determinados por los valores de las variables `AsmBool` y `Emitting`.

```
procedure emit3(op: OperationPart; arg1, arg2: integer);
begin
if AsmBool then
  begin
  if Emitting then
    begin
    writeln(code, ord(op));
    writeln(code, arg1);
    writeln(code, arg2);
    writeln(list, Address: 6, ':', tableNames[op]:12, arg1:6, arg2:6);
    end;
  Address := Address + 3
  end
else
  begin
  writeln(temp, ord(op));
  writeln(temp, arg1);
  writeln(temp, arg2);
  writeln(list, tableNames[op]:12, arg1:6, arg2:6);
  end
end; { emit3 }
```

Código 10.2: El pseudocódigo de la función emit3

`AsmBool` controla si se está emitiendo código hacia el fichero temporal (temp) o bien hacia el definitivo (code). En caso que se esté emitiendo código hacia el fichero definitivo (`AsmBool = true`), la variable global `Emitting` controla si se está en la primera (false) o segunda (true) pasada del ensamblador. En la primera pasada del ensamblador, simplemente se incrementa `Address` en el número de palabras que ocupará esta instrucción.

```
procedure emit1(op: OperationPart);
procedure emit2(op: OperationPart; Arg: integer);
procedure emit5(op: OperationPart; arg1, arg2, arg3, arg4: integer);
```

Código 10.3: Funciones para insertar instrucciones en el código intermedio

10.2.1. Direccionamiento de variables

Uno de los aspectos más delicados de la generación de código es el direccionamiento de variables, que se lleva a cabo utilizando un nivel relativo y un desplazamiento que identifican unívocamente a cada variable.

Consideremos el ejemplo del Código 10.4. Cuando se activa el procedimiento *QuickSort*, se reserva espacio para sus parámetros y variables locales en el registro de activación. Una referencia dentro de *Partition* a la variable global `j` genera la instrucción:

```
VARIABLE 1 4
```

El primer argumento de la instrucción `VARIABLE` indica que la variable `j` está almacenada en un nivel (1) por encima de las variables de Partition. El segundo argumento muestra que `j` está almacenada cuatro palabras por debajo del registro base del registro de activación en que se encuentra.

```
procedure QuickSort(m, n: integer);
var i, j: integer;

   procedure Partition;
   begin      ... j := n ...    end;
```

Código 10.4: Ejemplo de código en *Pascal-*

Para generar esta instrucción, debemos extender la estructura de datos que almacena las variables en la TS con dos nuevos campos enteros, uno que represente el nivel absoluto (`var_level`) y otro el desplazamiento (`var_displ`), como muestra el Código 10.5.

```
type  objectrecord =  record
                      name: integer;
                      Case kind: class of
                          ...
                      variable,
                      value_parameter,
                      var_parameter: (var_level, var_displ: integer;
                      var_type: ptr);
                        ...
                      end;
```

Código 10.5: Los nuevos registros de la tabla de símbolos

La función que analiza un grupo de variables como

```
  var i, j: integer;
```

almacena los identificadores y los tipos de las variables `i` y `j` en dos registros distintos. La función también cuenta el número de variables del grupo, devuelve un puntero al registro que describe la última variable y un puntero al tipo de las variables (Un pseudocódigo para su cabecera se muestra en el Código 10.6).

```
{ variable_group -> VariableName { "," VariableName } ":" type_name }
procedure variable_group(kind: class; var last_var, typex: ptr; var
   Number: integer; stop: symbols);
```

Código 10.6: La cabecera de `VariableGroup`

La función `variable_definition` usa esta información para calcular el tamaño del grupo de variables, como se muestra en el Código 10.7.

```
{ variable_definition -> variable_group ";" }
procedure variable_definition(var last_var: ptr; var length: integer;
   stop: symbols);
var typex: ptr;
```

```
    Number: integer;
begin
variable_group(variable, last_var, typex, Number, [SemiColon1] + stop);
length := Number * type_length(typex);
match(Semicolon1, stop);
end; { variable_definition }
```

Código 10.7: Variable_definition

El tamaño de una variable se calcula con la función `type_length` cuyo pseudocódigo es similar al que se presenta en el Código 10.8.

```
function type_length(typex: ptr): integer;
begin
if typex^.kind = standard_type then
  type_length := 1
else
  if typex^.kind = array_type then
    type_length := (typex^.upper_bound - typex^.lower_bound + 1) *
       type_length(typex^.element_type)
  else {typex^.kind = record_type }
    type_length := typex^.record_length
end;
```

Código 10.8: El código de `type_length`

Una variable estándar (integer, boolean) tiene tamaño 1. El tamaño de un objeto de tipo array se calcula multiplicando el número de elementos del array por el tamaño de uno de sus elementos. El tamaño de un registro (`record_length`) se almacena en la estructura de datos con la que se representa al objeto (Ver Código 10.9).

```
type object_record =  record
                        name: integer;
                        Case kind: class of
                          ...
                          record_type: (record_length: integer;
                             last_field: ptr);
                          field: (FieldDispl: integer; field_type: ptr)
                             ;
                          ...
                          end;
```

Código 10.9: `record_length` en la tabla de símbolos

La función que analiza una definición de variables

```
var i, j: integer;
```

está definida como se muestra en la figura 10.10.

```
{ variable_definition_part -> var variable_definition {
   variable_definition}}
```

```
procedure variable_definition_part(var length: integer; stop: symbols);
var stop_aux: symbols;
    last_var: ptr;
    more: integer;
begin
stop_aux := [Id1] + stop;
match(Var1, stop_aux);
variable_definition(last_var, length, stop_aux);
while lookahead = Id1 do
  begin
  variable_definition(last_var, more, stop_aux);
  length := length + more
  end;
variable_addressing(length, last_var);
end; { variable_definition_part }
```

Código 10.10: La definición de variables

```
procedure variable_addressing(var_length: integer; last_var: ptr);
var displ: integer;
begin
displ := 3 + var_length;
while displ > 3 do
  begin
  displ := displ - type_length(last_var^.var_type);
  last_var^.var_level := block_level;
  last_var^.var_displ := displ;
  last_var := last_var^.previous
  end
end;
```

Código 10.11: La función `Variable_addressing`

Al final de la parte de definición de variables, el analizador sintáctico invocará a la función `variable_addressing` (ver el Código 10.11) que se mueve a través de la lista de variables en orden inverso y les asigna a cada una un desplazamiento. Las variables tienen desplazamientos (offset) con respecto a la parte de contexto del registro de activación que ocupa tres palabras. El direccionamiento de los parámetros formales es ligeramente diferente. El procedimiento *QuickSort* tiene como lista de parámetros:

```
m, n: integer
```

Cuando el analizador sintáctico encuentra una referencia al parámetro `n` en el procedimiento `Partition`, se ha de emitir la instrucción

```
VARIABLE 1 −1
```

El nivel 1 indica que `n` está almacenada un nivel por encima de las variables de `Partition`. El desplazamiento -1 muestra que `n` está almacenada una palabra por encima de la dirección base del registro de activación.

```
procedure parameter_addressing(param_length: integer; last_param: ptr);
var displ: integer;
begin
displ := 0;
while displ > (- param_length) do
  begin
  if last_param^.kind = var_parameter then
    displ := displ - 1
  else { last_param^.kind = value_parameter }
    displ := displ - type_length(last_param^.var_type);
  last_param^.var_level := block_level;
  last_param^.var_displ := displ;
  last_param := last_param^.previous
  end
end;
```

Código 10.12: Direccionamiento de parámetros

Cuando el analizador sintáctico revisa la lista de parámetros formales de `QuickSort`, almacenará el identificador y el tipo de los parámetros formales `m` y `n` en las estructuras de datos correspondientes.

```
 { variable_access -> VariableName { Selector }
  Selector -> indexed_selector | field_selector }
procedure variable_access(var typex: ptr; stop: symbols);
var stop_aux: symbols;
    obj: ptr;
    level: integer;

begin
...
level := block_level - obj^.var_level;
if obj^.kind = var_parameter then
  emit3(Var_param2, level, obj^.var_displ)
else
  emit3(Variable2, level, obj^.var_displ);
...
end; { variable_access }
```

Código 10.13: Acceso a variables

El analizador sintáctico también calculará la longitud total de la lista de parámetros y mantendrá un puntero al último parámetro `n`. Al final del análisis de la lista de parámetros el analizador sintáctico se moverá en orden inverso por la lista de parámetros y les asignará un desplazamiento negativo (ver el código de `parameter_addressing` en el Código 10.12).

Un parámetro pasado por variable ocupa una palabra. El tamaño de un parámetro pasado por valor está determinado por su tipo. Cuando el analizador sintáctico encuentra

un identificador de variable en una sentencia invoca a la función `variable_access`. Esta función localiza un puntero a la estructura de datos que representa al objeto en el que está almacenado el nivel de la variable y su desplazamineto y emite una instrucción `VARIABLE` como indica el fragmento de código del Código 10.13.

Cuando el analizador sintáctico encuentra una referencia a un elemento de un array (de la forma `X[e]`), se ha de producir código para direccionar `X` y evaluar `e`. Después se usará la descripción del tipo de `X` para emitir la instrucción `INDEX`. (Ver el Código 10.14).

```
{ indexed_selector -> "[" expression "]" }
procedure indexed_selector(var typex: ptr; stop: symbols);
begin
...
Leng := type_length(typex^.element_type);
emit5(Index2, typex^.lower_bound, typex^.upper_bound, Leng, line_num);
...
end; { indexed_selector }
```

Código 10.14: Acceso a componentes de un vector

El direccionamiento de los campos de un registro es muy similar al direccionamiento de variables (los campos no tienen nivel). A los campos de un tipo registro se les asignan desplazamientos consecutivos empezando desde cero. El desplazamiento se almacena en el campo `field_displ` que tiene asociado. (Ver Código 10.15).

```
procedure field_addressing(record_length: integer; last_field: ptr);
var displ: integer;
begin
displ := record_length;
while displ > 0 do
  begin
  displ := displ - type_length(last_field^.field_type);
  last_field^.FieldDispl := displ;
  last_field := last_field^.previous
  end
end;
```

Código 10.15: Direccionamiento de los campos de un registro

Cuando el analizador sintáctico reconoce una referencia al campo de una variable (`X.f`) encuentra la estructura que representa al campo `f` y emite una instrucción `FIELD` como muestra el Código 10.16.

```
procedure field_selector(var typex: ptr; stop: symbols);
begin
...
typex := obj^.field_type;
emit2(Field2, obj^.FieldDispl);
...
end; { field_selector }
```

Código 10.16: Selector de campo en un registro

10.2.2. Código para las expresiones.

Consideremos un bloque con dos variables locales y una expresión como:

```
var y, z: integer;
.......
   y * z div 5
.......
```

Esta expresión se compila en las siguientes instrucciones:

```
VARIABLE 0  3
VALUE 1
VARIABLE 0  4
VALUE   1
MULTIPLY
CONSTANT  5
DIVIDE
```

Sigamos el proceso de compilación paso a paso:

1. Cuando el analizador sintáctico encuentra la expresión, invoca a cuatro funciones llamadas `expression`, `simple_expression`, `term` y `factor` en este orden.

```
{ factor -> constant | variable_access |
            "(" expression ")" | "not" factor }
procedure factor(var typex: ptr; stop: symbols);
begin
...
variable_access(typex, stop);
Leng := type_length(typex);
emit2(Value2, Leng);
push(Leng - 1);
...
end; { factor }
```

Código 10.17: El código de `Factor`

`Factor` reconoce un identificador de variable y ejecuta las sentencias que se muestran en el Código 10.17. La llamada a `variable_access` genera la primera instrucción `VARIABLE 0 3` y la función `factor` produce `VALUE 1`.

2. Cuando la función `term` reconoce el operador de multiplicación, entra en el siguiente bucle que almacena el operador temporalmente e invoca nuevamente a `factor`. Esta vez, `factor` reconoce la variable `z` y genera las dos instrucciones `VARIABLE 0 4` y `VALUE 1`.

3. Después la función `term` ejecuta las sentecias que se muestran en el Código 10.18. El efecto de estas sentencias es traducir el operador de multiplicación a una instrucción `MULTIPLY`.

```
 { term -> factor { MultiplyingOperator factor }
  MultiplyingOperator -> "*" | "div" | "mod" | "and"  }
procedure term(var typex: ptr;  stop: symbols);
begin
...
if typex = type_integer then
  begin
  check_types(typex, type1);
  if Operator = Asterisk1 then
    emit1(Multiply2)
  else
    if Operator = Div1 then
      emit2(Divide2, line_num)
    else
      if Operator = Mod1 then
        emit2(Modulo2, line_num)
      else { if Operator = and1 then }
        typex_error(typex);
  pop(1);
  end
...
end; { term }
```

Código 10.18: Pseudocódigo para `term`

```
procedure factor(var typex: ptr; stop: symbols);
begin
...
  constant(value , typex, stop);
  emit2(Constant2, value);
  push(1);
...
end; { factor }
```

Código 10.19: La función `factor`

4. Cuando `term` encuentra el operador de división, se almacena temporalmente y se invoca nuevamente a `factor`. `Factor` reconoce ahora la constante 5 (ver Código 10.19) y emite la instrucción `CONSTANT 5`.

5. Finalmente, `term` emite el operador de división como una instrucción `DIVIDE line_num`.

Las funciones `simple_expression` y `expression` se comportan de manera similar.

El compilador determina cuanto espacio (en palabras) es necesario en la parte de temporales para ejecutar el bloque de un programa. El intérprete de código que se escribió en la práctica anterior utiliza esta información al comienzo de un bloque para comprobar si hay suficiente memoria para ejecutar el bloque. Una vez hecho esto, no es necesario comprobar la cantidad de memoria disponible en el resto del bloque. Durante el análisis de ámbito el analizador sintáctico mantenía una estructura de datos en la que almacenaba los objetos de cada bloque; cuando el analizador sintáctico entra en un nuevo bloque del programa fuente, se coloca un nuevo registro en esa estructura. Al final del bloque dicho registro es eliminado de la estructura (a través de una operación *reset*. Durante la generación de código, se usa la misma estructura de datos para mantener el tamaño de memoria que necesita cada bloque. La información correspondiente a un bloque se amplía como se muestra en el Código 10.20.

```
type  block_record =  record
                        TempLength, MaxTemp: integer;
                        last_object: ptr;
                        end;
      block_table = Array [0..Max_level] of block_record;
```

Código 10.20: Registros de la tabla de símbolos

Cuando el compilador emita una instrucción, tendrá que predecir cuál será el tamaño en que cambiará la pila cuando se ejecute la instrucción. El analizador sintáctico almacenará la longitud actual del área de temporales en el campo `temp_length`. El máximo número de localizaciones temporales que se utilizan en el bloque se almacenan el el campo `Max_temp`. Al comienzo de un bloque ambos estarán a cero. Cuando el analizador sintáctico emite una instrucción `VARIABLE`, se invoca a la función `push` para indicar que la ejecución de esta instrucción incrementa la pila en una palabra: `push(1)`. La función `push` incrementa el campo `temp_length` del bloque actual y cambia `Max_temp` si fuese necesario (ver Código 10.21).

```
procedure push(length: integer);
begin
block[block_level].TempLength := block[block_level].TempLength+length;
if block[block_level].MaxTemp < block[block_level].TempLength then
  block[block_level].MaxTemp:= block[block_level].TempLength
end;
```

Código 10.21: La función `push`

Después de emitir una instrucción `MULTIPLY`, el analizador sintáctico invoca a una función llamado `pop` para indicar que la ejecución de esta instrucción decrementará la pila en una palabra: `pop(1)`. La función `pop` decrementará el `temp_length` correspondiente al bloque actual, como muestra el Código 10.22.

```
procedure pop(length: integer);
begin
block[block_level].TempLength := block[block_level].TempLength - length
end;
```

Código 10.22: Pseudocódigo para la función `pop`

10.2.3. Código para las sentencias.

La sentencia de asignación.

El código de una sentencia de asignación tiene la siguiente forma:

```
variable_access
expression
ASSIGN length
```

La función que analiza una sentencia de asignación ha de modificarse como se muestra en el Código 10.23.

```
{ assign_statement = variable_access ":=" expression. }
procedure assign_statement(stop: symbols);
begin
...
length := type_length(expr_type);
emit2(Assign2, length);
pop(1 + length);
...
end; { assign_statement }
```

Código 10.23: Sentencia de asignación

Durante la ejecución de una sentencia de asignación, la máquina empuja dos temporales en su pila: La dirección de una variable y el valor de una expresión. Estos temporales deben ser quitados de la pila por la instrucción de asignación (`pop(1 + length)`).

La sentencia de repetición.

El código de una sentencia while de la forma:

```
While B do S
```

incluye dos instrucciones de salto, como puede verse en el Código 10.24. El analizador sintáctico (generador de código) revisa el programa de izquierda a derecha y emite código directamente. Cuando el analizador sintáctico está listo para emitir la instrucción `GOFALSE` todavía no ha compilado las instrucciones de `S`. Por lo tanto es incapaz de emitir la dirección `L2`.

```
L1: B
    GOFALSE L2
    S
    GOTO L1
```

```
L2:
```

Código 10.24: El código para una sentencia `while`

El problema de las referencias hacia adelante se resuelve en tres pasos:

1. El generador de código asignará una etiqueta numérica a cada destino de salto. Las etiquetas son números enteros distintos. Podríamos suponer, por ejemplo, que los puntos `L1` y `L2` del programa tienen etiquetas 17 y 18. En este caso el analizador sintáctico emitiría el siguiente código intermedio para la sentencia while:

   ```
   DEFADDR 17
   B Code
   GOFALSE 18
   S Code
   GOTO  17
   DEFADDR 18
   ```

2. En su primera pasada, el ensamblador revisará el código intermedio y hará un seguimiento de la dirección de la instrucción en curso. La dirección de la instrucción se computará de forma relativa al comienzo del programa

   Supongamos que las instrucciones correspondientes a la compilación de una sentencia `while` tienen las siguientes direcciones:

   ```
   Direcciones       Código
   279      DEFADDR 17
   279      B Code
   287      GOFALSE 18
   289      S Code
   320      GOTO  17
   322      DEFADDR 18
   ```

   El ensamblador almacenará la dirección de cada destino de salto en una tabla. Cuando el ensamblador reciba la entrada `DEFADDR 17` se almacenará la dirección actual en la entrada número 17 de la tabla: `table[17] = 279`.

   Cuando encuentre la instrucción `DEFADDR 18` se almacenará la dirección en la entrada número 18: `table[18] = 322`. Durante el análisis del código, el ensamblador define las direcciones de todas las etiquetas pero no emite código. La instrucción `DEFADDR` sólo sirve para definir direcciones de salto. Cuando se leen estas pseudoinstrucciones, no se modifica la dirección actual y no se emite nada.

3. En este último paso, el ensamblador revisa nuevamente el código y emite el código final: Cuando el ensamblador lee la instrucción `GOFALSE 18` computa el desplazamiento de la etiqueta 18 relativo a la propia instrucción `GOFALSE`:

   ```
   go_false_label = Table[18] − 287 = 322 − 287 = 35
   ```

y emite la instrucción con ese desplazamiento: `GOFALSE 35`

El desplazamiento de una instrucción `GOTO` se calcula del mismo modo:

```
goto_displ = Table[17] − 320 = 279 − 320 = −41
```

El código final queda:

```
B Code
GOFALSE 35
S Code
GOTO  −41
```

Para implementar lo explicado anteriormente, el analizador sintáctico utiliza la variable global `label_no` para contar el número de etiquetas creadas. Cuando el analizador sintáctico necesita una nueva etiqueta, se ejecuta la función que se muestra en el Código 10.25.

```
var label_no: integer;
...
procedure new_label(var No: integer);
begin
if label_no > max_label then
  error(' Program too large.');
label_no := label_no + 1;
No := label_no
end;
```

Código 10.25: Generación de etiquetas de salto

El análisis de una sentencia `while` es ahora directo (véase el Código 10.27).

El ensamblador mantiene la dirección de la instrucción actual en la variable `address`. La dirección se incrementa después de la emisión de una instrucción. Las instrucciones son almacenadas en una tabla. (Ver las definiciones del Código 10.26).

```
const max_label = 1000;        { Número Máximo de etiquetas }
type    assembly_table = array[1..max_label] of integer;
var     Address: integer;        { Contador de direcciones }
        table: assembly_table;   { Tabla de direcciones }
```

Código 10.26: Definiciones para el ensamblador

Después de la entrada de una pseudoinstrucción `DEFADDR`, el ensamblador almacena la dirección actual en la tabla (`table`) y lee la siguiente instrucción. Cuando el ensamblador encuentra una instrucción de salto emite la instrucción de desplazamiento. Durante la primera fase del ensamblado, el compilador no emite código. Para controlar cuando se ha de emitir código y cuando no, se utiliza la variable lógica `emitting`.

```
{ while_statement −> "while" expression "do" statement }
procedure while_statement(stop: symbols);
```

```
var expr_type: ptr;
    Label1, Label2: integer;

begin
new_label(Label1);
emit2(Def_addr2, Label1);
match(While1, Expression_symbols + [Do1] + Statement_symbols + stop);
expression(expr_type, [Do1] + Statement_symbols + stop);
check_types(expr_type, type_boolean);
match(Do1, Statement_symbols + stop);
new_label(Label2);
emit2(Go_false2, Label2);
pop(1);
statement(stop);
emit2(Goto2, Label1);
emit2(Def_addr2, Label2);
end; { while_statement }
```

Código 10.27: Sentencia while

La sentencia condicional.

El código para una sentencia `if` se obtiene de forma similar siguiendo los siguientes esquemas de traducción:

```
if B then S1 else S2

B Code
GOFALSE L1
S1 Code
GOTO L2
L1: S2 Code
L2:
```

```
if B then S

B Code
GOFALSE L1
S Code
L1:
```

```
PROCEDURE var_length temp_length displ line_num
Partition Code
SL Code
ENDPROC param_length
```

Código 10.28: Código objeto correspondiente al fuente del Código 10.29

10.2.4. Código para los procedimientos.

El código para el procedimiento QuickSort que se muestra en el Código 10.29 tiene la forma que muestra el Código 10.28.

```
procedure QuickSort( m, n : Integer);
var i, j : Integer;

procedure Partition;
begin ... end;

begin
SL
end;
```

Código 10.29: El procedimiento *Quicksort*

Cuando el generador de código alcanza el procedimiento *QuickSort*, debe emitir la instrucción `PROCEDURE`; pero en este punto, no es capaz de calcular los siguientes argumentos:

1. `Var_length` es el tamaño de las variables locales `i` y `j`. Este argumento sólo se conoce una vez que se haya hecho el análisis sintáctico de la parte de definición de las variables locales al procedimiento:

   ```
   var i, j: integer;
   ```

2. `Temp_length` es la extensión máxima de los temporales en la lista de sentencias `SL`. Este argumento se conocerá sólo al alcanzar el final del procedimiento *QuickSort*.

3. `Displ` es la dirección de la lista de sentencias relativa a la instrucción `PROCEDURE`. Este argumento es conocido al comienzo de `SL`.

Estas referencias hacia adelante las resolverá el ensamblador. El analizador sintáctico asignará una etiqueta numérica a cada uno de los argumentos mencionados, y emitirá la instrucción `PROCEDURE` con esas etiquetas:

```
PROCEDURE var_label temp_label begin_label line_num
```

Cuando el analizador sintáctico alcance el punto donde el valor de alguno de los argumentos sea conocido, emitirá su valor (y la etiqueta correspondiente) en forma de una pseudoinstrucción `DEFARG` (Define Argument):

```
DEFARG var_label var_length
```

Durante la primera pasada, el ensamblador utilizará la pseudoinstrucción `DEFARG` para introducir el argumento en la tabla: `table[var_label] = var_length`. En la segunda pasada, el ensamblador reemplazará la etiqueta `var_label` en la instrucción `PROCEDURE` por el valor correspondiente en la tabla.

El Código 10.30 es el código intermedio que genera el compilador (queda almacenado en el fichero TEMP.$$$) al compilar el fichero 10.29.

```
DEFADDR     proc_label
PROCEDURE var_label Temp_label  Begin_label line_num
Partition Code
DEFADDR   begin_label
SL Code
DEFARG    var_label var_length
DEFARG    temp_label  Max_temp
ENDPROC   param_length
```

Código 10.30: Código generado al compilar el programa 10.29

El analizador sintáctico también asignará una etiqueta a la instrucción `PROCEDURE` (`Proc_label`). Cada llamada al procedimiento *QuickSort* utilizará esta etiqueta para referenciar el código del procedimiento. Por ejemplo, la sentencia

```
QuickSort(m, j)
```

se compila en código que empuja los parámetros actuales `m` y `j` en el top de la pila, seguidos de una instrucción `PROCCALL`:

```
m Code
j Code
PROCCALL level proc_label
```

El nivel (`level`) de un procedimiento se define de forma relativa al bloque en el que es invocado. El ensamblador reemplazará la etiqueta del procedimiento con el desplazamiento del código correspondiente. Durante la generación de código, el nivel y la etiqueta de un procedimiento se almacenan en la estructura de datos que se le ha asignado (ver Código 10.31).

```
type object_record =  record
                        name: integer;
                        Case kind: class of
                          ...
                          procedur: (proc_level, proc_label: integer;
                                     last_param: ptr);
                        end;
```

Código 10.31: Información para un procedimiento en los registros de la tabla de símbolos

La versión final de la función que analiza un procedimiento es la que se muestra en el Código 10.32.

Los argumentos de la instrucción `PROCEDURE` son definidos al final del cuerpo del bloque, como se muestra en el Código 10.33.

Si se obvian los procedimientos estándar, un procedimiento definido por el programador se invoca como se muestra en el Código 10.34. Cuando el código de una sentencia de procedimiento se ejecuta, la parte temporal del registro de activación actual se incrementa con los parámetros actuales y la parte de contexto del nuevo registro de activación. La parte de contexto ocupa tres palabras. Cuando se regresa de la llamada

al procedimiento, los parámetros y el contexto son retirados de la pila. Cuando el ensamblador encuentra la instrucción `PROCCALL`, reemplaza las etiquetas del procedimiento con la dirección del código del procedimiento (relativo a la dirección actual).

El alumno puede generar los ficheros ensamblador (.ASM) y ejecutable correspondientes a la implementación en *Pascal-* del QuickSort mostrado en el Código 10.35, utilizando el compilador `p-c` que se puede descargar en el portal web de la asignatura. También se puede descargar del portal de la asignatura un intérprete que puede usarse para ejecutar (interpretar) el código intermedio producido por el compilador.

El Código 10.36 corresponde al contenido del fichero `.ASM` (ensamblador generado por el compilador `p-c` para el programa del Código 10.35 (quicksort).

```
0: PROGRAM 11 6 311 3
5: PROCEDURE 2 5 262 8
10: PROCEDURE 2 4 5 12
15: VARIABLE 0 3
18: VARIABLE 2 3
21: VARIABLE 1 -2
24: VALUE 1
26: VARIABLE 1 -1
29: VALUE 1
31: IADD
32: CONSTANT 2
34: DIVIDE 15
36: INDEX 1 10 1 15
41: VALUE 1
43: ASSIGN 1
45: VARIABLE 1 3
48: VARIABLE 1 -2
51: VALUE 1
53: ASSIGN 1
55: VARIABLE 1 4
58: VARIABLE 1 -1
61: VALUE 1
63: ASSIGN 1
65: VARIABLE 1 3
68: VALUE 1
70: VARIABLE 1 4
73: VALUE 1
75: NOTGREATER
76: GOFALSE 189
78: VARIABLE 2 3
81: VARIABLE 1 3
84: VALUE 1
86: INDEX 1 10 1 20
91: VALUE 1
93: VARIABLE 0 3
96: VALUE 1
```

```
98: LESS
99: GOFALSE 17
101: VARIABLE 1 3
104: VARIABLE 1 3
107: VALUE 1
109: CONSTANT 1
111: IADD
112: ASSIGN 1
114: GOTO -36
116: VARIABLE 0 3
119: VALUE 1
121: VARIABLE 2 3
124: VARIABLE 1 4
127: VALUE 1
129: INDEX 1 10 1 21
134: VALUE 1
136: LESS
137: GOFALSE 17
139: VARIABLE 1 4
142: VARIABLE 1 4
145: VALUE 1
147: CONSTANT 1
149: SUBTRACT
150: ASSIGN 1
152: GOTO -36
154: VARIABLE 1 3
157: VALUE 1
159: VARIABLE 1 4
162: VALUE 1
164: NOTGREATER
165: GOFALSE 98
167: VARIABLE 0 4
170: VARIABLE 2 3
173: VARIABLE 1 3
176: VALUE 1
178: INDEX 1 10 1 24
183: VALUE 1
185: ASSIGN 1
187: VARIABLE 2 3
190: VARIABLE 1 3
193: VALUE 1
195: INDEX 1 10 1 24
200: VARIABLE 2 3
203: VARIABLE 1 4
206: VALUE 1
208: INDEX 1 10 1 24
213: VALUE 1
```

```
215: ASSIGN 1
217: VARIABLE 2 3
220: VARIABLE 1 4
223: VALUE 1
225: INDEX 1 10 1 25
230: VARIABLE 0 4
233: VALUE 1
235: ASSIGN 1
237: VARIABLE 1 3
240: VARIABLE 1 3
243: VALUE 1
245: CONSTANT 1
247: IADD
248: ASSIGN 1
250: VARIABLE 1 4
253: VARIABLE 1 4
256: VALUE 1
258: CONSTANT 1
260: SUBTRACT
261: ASSIGN 1
263: GOTO -198
265: ENDPROC 0
267: VARIABLE 0 -2
270: VALUE 1
272: VARIABLE 0 -1
275: VALUE 1
277: LESS
278: GOFALSE 31
280: PROCCALL 0 -270
283: VARIABLE 0 -2
286: VALUE 1
288: VARIABLE 0 4
291: VALUE 1
293: PROCCALL 1 -288
296: VARIABLE 0 3
299: VALUE 1
301: VARIABLE 0 -1
304: VALUE 1
306: PROCCALL 1 -301
309: ENDPROC 2
311: VARIABLE 0 13
314: CONSTANT 1
316: ASSIGN 1
318: VARIABLE 0 13
321: VALUE 1
323: CONSTANT 10
325: NOTGREATER
```

```
326: GOFALSE 31
328: VARIABLE 0 3
331: VARIABLE 0 13
334: VALUE 1
336: INDEX 1 10 1 44
341: READ
342: VARIABLE 0 13
345: VARIABLE 0 13
348: VALUE 1
350: CONSTANT 1
352: IADD
353: ASSIGN 1
355: GOTO −37
357: CONSTANT 1
359: CONSTANT 10
361: PROCCALL 0 −356
364: VARIABLE 0 13
367: CONSTANT 1
369: ASSIGN 1
371: VARIABLE 0 13
374: VALUE 1
376: CONSTANT 10
378: NOTGREATER
379: GOFALSE 33
381: VARIABLE 0 3
384: VARIABLE 0 13
387: VALUE 1
389: INDEX 1 10 1 51
394: VALUE 1
396: WRITE
397: VARIABLE 0 13
400: VARIABLE 0 13
403: VALUE 1
405: CONSTANT 1
407: IADD
408: ASSIGN 1
410: GOTO −39
412: ENDPROG
```

Código 10.36: Código ensamblador correspondiente al Código 10.35 (*Quicksort*)

```
ADD
AND
ASSING length
CONSTANT value
DIVIDE line_num
ENDPROC param_length
ENDPROG
EQUAL
FIELD displ
GOFALSE label
GOTO label
GREATER
INDEX lower_bound upper_bound length line_num
LESS
MINUS
MODULO line_num
MULTIPLY
NOT
NOTEQUAL
NOTGREATER
NOTLESS
OR
PROCCALL level displ
PROCEDURE var_length temp_length displ line_num
PROGRAM var_length temp_length displ line_num
SUBTRACT
VALUE length
VARIABLE level displ
VARPARAM level displ
READ
WRITE
DEFADDR label_no
DEFARG label_no value
```

Código 10.1: Juego de instrucciones de la P-Máquina

```
{ procedure_definition -> "procedure" ProcedureName ProcedureBlock ";"
  ProcedureBlock -> ["(" formal_parameter_list ")"] ";" block_body }
procedure procedure_definition(stop: symbols);
var name: integer;
    last_param, obj: ptr;
    param_length, var_label, temp_label, begin_label: integer;

begin
match(Procedure1, [Id1, LeftParenthesis1, Semicolon1] + Block_symbols +
    stop);
match_id(name, [LeftParenthesis1, Semicolon1] + Block_symbols + stop);
insert(name, procedur, obj);
obj^.ProcLevel := block_level;
new_label(obj^.ProcLabel);
setx;
if lookahead = LeftParenthesis1 then
  begin
  match(LeftParenthesis1, Parameter_symbols + [RightParenthesis1,
     Semicolon1] + Block_symbols + stop);
  formal_parameter_list(last_param, param_length, [RightParenthesis1,
     Semicolon1] + Block_symbols + stop);
  match(RightParenthesis1, [Semicolon1] + Block_symbols + stop);
  end
else { no parameter list }
  begin
  last_param := NIL;
  param_length := 0;
  end;
obj^.last_param := last_param;
new_label(var_label);
new_label(temp_label);
new_label(begin_label);
emit2(Def_addr2, obj^.ProcLabel);
emit5(Procedure2, var_label, temp_label, begin_label, line_num);
match(SemiColon1, [SemiColon1] + Block_symbols + stop);
block_body(begin_label, var_label, temp_label, [SemiColon1] + stop);
resetx;
match(SemiColon1, stop);
emit2(End_proc2, param_length);
end; { procedure_definition }
```

Código 10.32: La función `procedure_definition`

```
{ block_body -> [constant_definition_part] [type_definition_part]
                [variable_definition_part] { procedure_definition }
                compound_statement                              }
procedure block_body(var begin_label, var_label, temp_label: integer;
   stop: symbols);
var length: integer;

begin
syntax_check(Block_symbols + stop);
if lookahead = Const1 then
  constant_definition_part([type1, Var1, Procedure1, Begin1] + stop);
if lookahead = type1 then
  type_definition_part([Var1, Procedure1, Begin1] + stop);
if lookahead = Var1 then
  variable_definition_part(length, [Procedure1, Begin1] + stop)
else
  length := 0;
while lookahead = Procedure1 do
  procedure_definition([Procedure1, Begin1] + stop);

emit2(Def_addr2, begin_label);
compound_statement(stop);
emit3(Def_arg2, var_label, length);
emit3(Def_arg2, temp_label, block[block_level].MaxTemp);
end; { block_body }
```

Código 10.33: Análisis de un bloque

```
{ procedure_statement = io_statement | ProcedureName ["("
   actual_parameter_list ")"]. }
procedure procedure_statement(stop: symbols);
var stop_aux: symbols;
    obj: ptr;
    param_length: integer;

begin
lookup(ArgValue.ptr^.Index, obj);
{ lookahead = (procedure) Name1, al entrar }
if obj^.kind = standard_proc then
  io_statement(stop)
else
  begin
  if obj^.last_param <> NIL then
    begin
    stop_aux := [RightParenthesis1] + stop;
    match(Id1, [LeftParenthesis1] + Expression_symbols + stop_aux);
    match(LeftParenthesis1, Expression_symbols + stop_aux);
    actual_parameter_list(obj^.last_param, param_length, stop_aux);
    match(RightParenthesis1, stop);
    end
  else { No hay parámetros }
    begin
    match(Id1, stop);
    param_length := 0;
    end;
  emit3(Proc_call2, block_level - obj^.ProcLevel, obj^.ProcLabel);
  push(3);
  pop(param_length + 3);
  end
end; { procedure_statement }
```

Código 10.34: Sentencia de llamada a procedimiento

```
program Ordenacion;
const max = 10;
type T = array[1..max] of integer;
var
  a : T; K : integer;

procedure QuickSort(m,n: integer);
var
i, j : integer;

  procedure Partition;
  var r, w : integer;
  begin
    r := A[ (m + n) div 2];
    i := m;
    j := n;
    while i <= j do
    begin
      while A[i] < r do i := i + 1;
      while r < A[j] do j := j - 1;
      if i <= j then
        begin
          w := A[i]; A[i] := A[j];
          A[j] := w; i := i + 1;
          j := j - 1
        end;
    end;
  end;

begin
  if m < n then
    begin
      Partition;
      QuickSort(m,j);
      QuickSort(i,n)
    end;
end;

begin
  k := 1;
  while k <= max do
    begin
      read(a[k]);
      k := k + 1
    end;
  Quicksort(1, max);
  k := 1;
  while k <= max do
    begin
      write(a[k]);
      k := k + 1
    end;
end
```