Universidad de La Laguna

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

Comprobación de tipos en *Pascal-*

Factor de ponderación [0-10]: 10

9.1. Objetivos

Ampliar el compilador de *Pascal-* que ya realiza el análisis sintáctico y de ámbito para que lleve a cabo el análisis de tipo de las diferentes construcciones del lenguaje.

9.2. Descripción

9.2.1. Los tipos básicos

Cada tipo básico de *Pascal-* (integer o boolean) se describe mediante un registro (`object_record`) de la Tabla de Símbolos (TS) en la que se almacena el identificador (`name`) del tipo. En el caso de los tipos básicos, del registro de la TS sólo se utilizará el campo `kind` (clase) con el valor `CL_STANDARD_TYPE` (el correspondiente a un tipo estándar).

Durante el análisis de tipo, cada tipo básico se representa mediante un puntero al registro correspondiente. Estos punteros se almacenan en las variables `type_integer`, `type_boolean` y `type_error` (ver figura 9.1) y son inicializados en la función `standard_block()`.

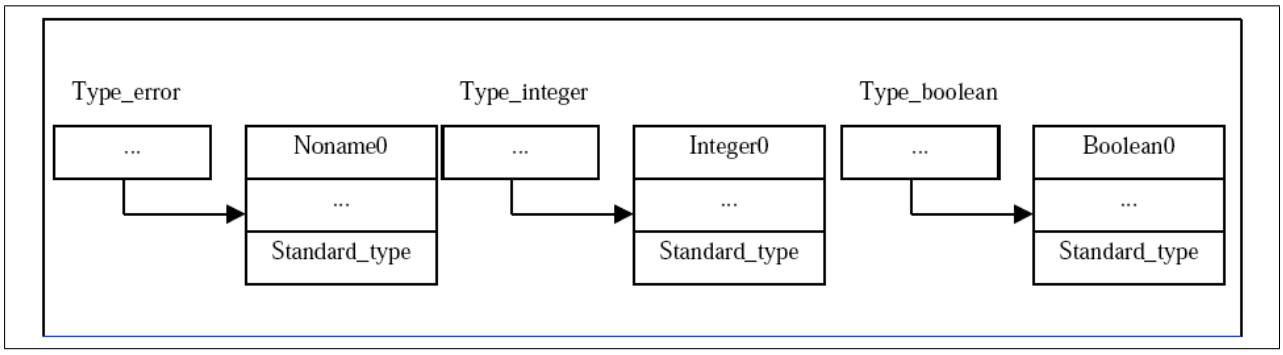

Figura 9.1: Los tipos básicos

9.2.2. Sistema de Tipos

Constantes

Para representar las constantes se utilizan los dos campos en los que se almacenan el tipo y el valor de la constante, tal como se muestra en el Código 9.1.

```
typedef struct ObjectRecord {
  int name;
  class kind;
  union {
    . . .
      /* CL_CONSTANT */
      struct {
         int ConstValue;
         ptr ConstType;
      } CS; /* Constant Struct */
  } KU; /* Kind Union */
} ObjectRecord;
```

Código 9.1: Constantes

Por ejemplo: En el bloque estándar se define la constante `false` del siguiente modo (ver Figura 9.2 y Código 9.2).

```
  ptr obj;
  ...
  Insert(FALSE0, CL_CONSTANT, &obj);
  obj->KU.CS.ConstValue = 0;
  obj->KU.CS.ConstType = TypeBoolean;
```

Código 9.2: Definición de la constante `false`

Cuando se encuentra en el código una constante, debe determinarse su tipo y su valor. Para ello debe modificarse la función que analiza una constante. De este modo, haciendo uso del campo `kind` podrá distinguirse entre los identificadores de constantes e identificadores de otra clase.

- Si se trata de un número, el tipo asociado a la constante es `type_integer` y su valor será el atributo del token `NUMERAL`
- En caso de que se trate de un identificador, se busca el mismo en el ámbito correspondiente y se devuelve un puntero al mismo.

  Si el objeto fue declarado como una constante, entonces se devuelven los valores almacenados en los campos `const_value` y `const_type`.

La recuperación de errores se realizará del siguiente modo:

- Si en esta situación (en la que se espera un identificador de constante) se encuentra que el identificador que no corresponde a una constante (no tiene clase `constantx`):

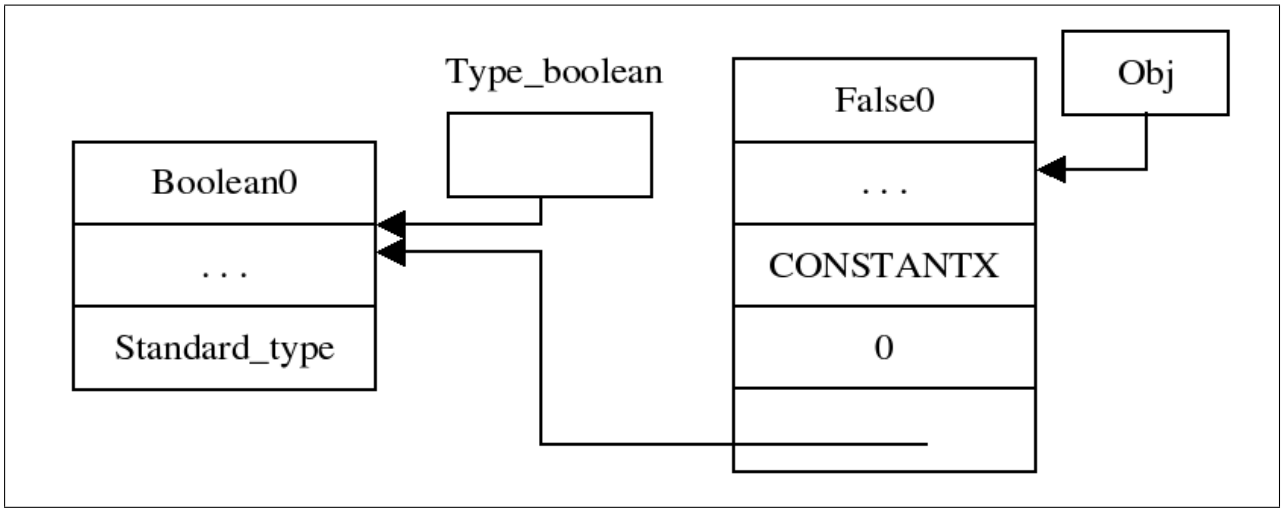

Figura 9.2: La constante `false`

- Se emitirá un mensaje de error indicando que la clase del identificador es errónea:

```
void KindError(ptr obj) {
  if (obj->kind != CL_UNDEFINED)
  Error("Invalid identifier kind.");
}
```

Si el identificador al que se hace referencia es un objeto no definido, ya se ha emitido un mensaje de error durante el análisis de ámbito. En este caso, el compilador no emitirá otro mensaje de error.

- Devolverá el valor cero y el tipo `type_error`.
- Si no se trata ni de un número ni de un identificador se produce un error sintáctico y en ese caso se devolverán también el valor cero y el tipo `type_error`.

A) Reescribir la función *constant()* siguiendo las indicaciones anteriores. La función tendrá una definición similar a:

```
void Constant(int *value, ptr *type, set *stop);
```

En la declaración de una constante se deben rellenar los campos correspondientes al valor y al tipo de la misma. Para conseguirlo, modificar:

```
void ConstantDefinition(set *stop);
```

Variables

Las diferentes clases de variables que utiliza *Pascal-* se ilustran en la Figura 9.3. Durante el proceso de compilación cada variable será descrita por su identificador (name), su clase y su tipo, tal como muestra el Código 9.3.

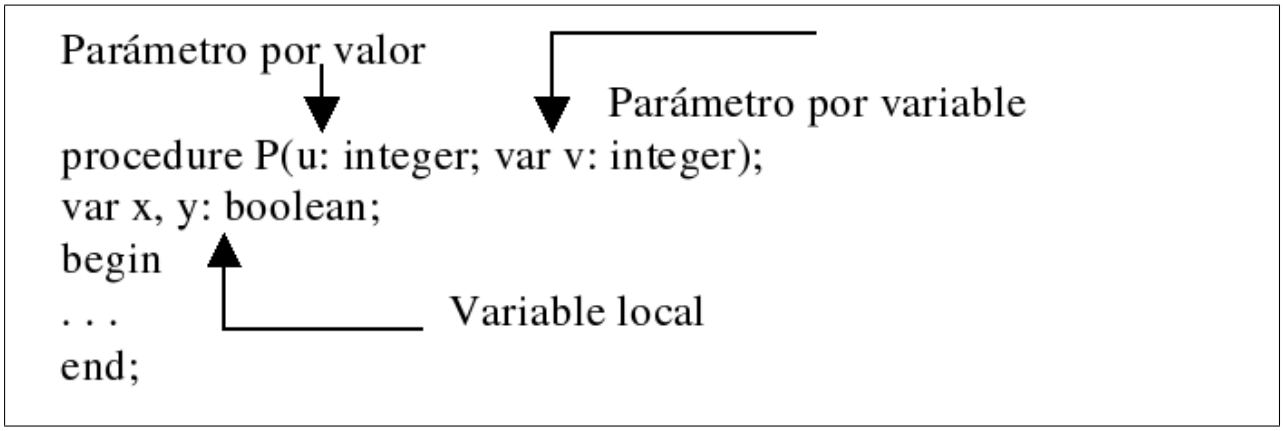

Figura 9.3: Clases de variables

```
typedef struct ObjectRecord {
  int name;
  class kind;

  union {
   ptr VarType; /* CL_VALUE_PARAMETER CL_VAR_PARAMETER CL_VARIABLE */
  } KU; /* Kind Union */
} ObjectRecord;
```

Código 9.3: Variables

La figura 9.4 muestra la estructura para las distintas variables del ejemplo anterior. Las

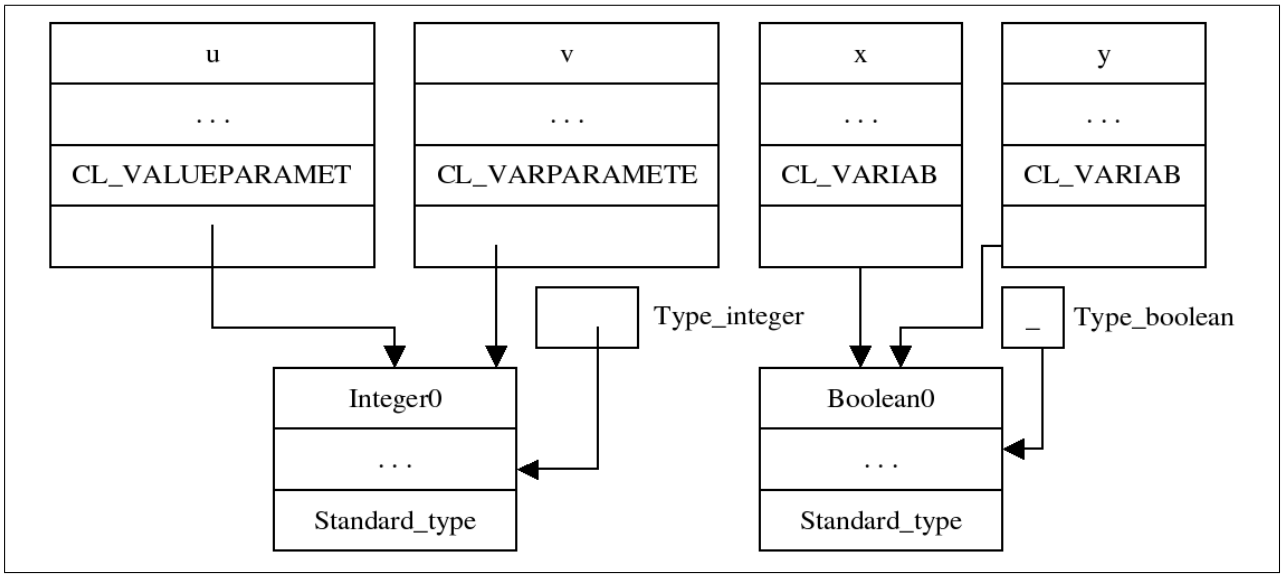

Figura 9.4: Estructura para las variables

estructuras de datos que representan a las variables locales y a los parámetros sólo se

diferencian en el campo clase (kind). Las definiciones de los parámetros y las variables tienen la misma sintaxis:

variable_group → variable_name { , variable_name } : type_name

Se debe entonces extender la función *variable_group()* con un parámetro `kind` y añadir el código necesario para crear de forma adecuada los registros que representan a las variables. Primero se debe revisar la lista de identificadores de variables y crear los registros correspondientes. Cuando se alcance el identificador del tipo, se debe ir hacia atrás en esa lista e insertar los punteros al registro de tipo correspondiente

Cuando el compilador espera un identificador de tipo en una sentencia se necesita un puntero al correspondiente registro de tipo. Para obtener dicho puntero se utiliza la función *type_name()* cuyo código podría ser similar al que se muestra en el Código 9.4.

```
#define TypesS(c) (c == CL_STANDARD_TYPE || c == CL_ARRAY_TYPE || \
       c == CL_RECORD_TYPE)
void Type_Name(ptr *type, set *stop) {
  ptr obj;

  if (LookAhead == ID) {
    LookUp(ArgValue.ptr->Index, &obj); /* Buscar el id */
    if (TypesS(obj->kind))  /* Si es id de tipo */
      *type = obj;
    else {
      KindError(obj);
      *type = TypeError;
    }
  }
  else
    *type = TypeError;
  Match(ID, set_union(stop, ENDS));
  ...
}
```

Código 9.4: Idenficadores de tipo

B) Modificar la función *variable_group()* siguiendo las indicaciones anteriores. Se debe devolver un puntero al último elemento de un grupo de definiciones de variables (será necesario para los procedimientos).

```
void VariableGroup(class kind, ptr *LastVar, set  *stop);
```

Al realizar las definiciones de variables se tiene que indicar que se trata de objetos de clase variable:

```
void VariableDefinition(set *stop);
```

La definición de parámetros requiere que se realice una llamada a *variable_group()*; si se trata de parámetros pasados por valor con el parámetro `kind` igual a `value_parameter`. En caso de que sean pasados por variable, `kind` debe valer `var_parameter`.

```
void ParameterDefinition(ptr *LastParam, set *stop);
```

Arrays

Los vectores se describen mediante un registro en el que se definen campos para los límites inferior y superior (`lower_bound` y `upper_bound`), el tipo de las cotas (`index_type`) y un campo en el que se almacena el tipo de los elementos del array (`element_type`), tal como vemos en el Código 9.5.

```
typedef struct ObjectRecord {
  int name;
  class kind;

  union {
    . . .
      /* CL_ARRAY_TYPE */
      struct {
         int LowerBound,
       UpperBound;
         ptr IndexType,
       ElementType;
      } AS; /* Array Struct */
  } KU; /* Kind Union */
} ObjectRecord;
```

Código 9.5: Estructura para los arrays

La figura 9.5 muestra la estructura de datos para una definición de tipo como la siguiente:

```
type T = array[1..10] of boolean;
```

Una definición de tipo completa tiene la sintaxis:

type_definition → type_name = new_type ;
new_type → new_array_type | new_record_type

La función que analiza una definición de tipo (*type_definition*) toma el identificador de un nuevo tipo array y lo pasa como parámetro a la función *new_array_type()*, que analiza la definición de un nuevo tipo array y construye el correspondiente registro. En caso de que el primer índice del subrango sea mayor que el segundo, se ha de emitir un mensaje de error ('Invalid Range.') y asignarle a ambas cotas el mismo valor. Para comprobar que el tipo asociado a los índices es el mismo utilizar una función similar a la que se muestra en el Código 9.6.

```
  void CheckTypes(ptr *type1, ptr *type2) {
    if (*type1 != *type2) {
      if (*type1 != TypeError && *type2 != TypeError)
```

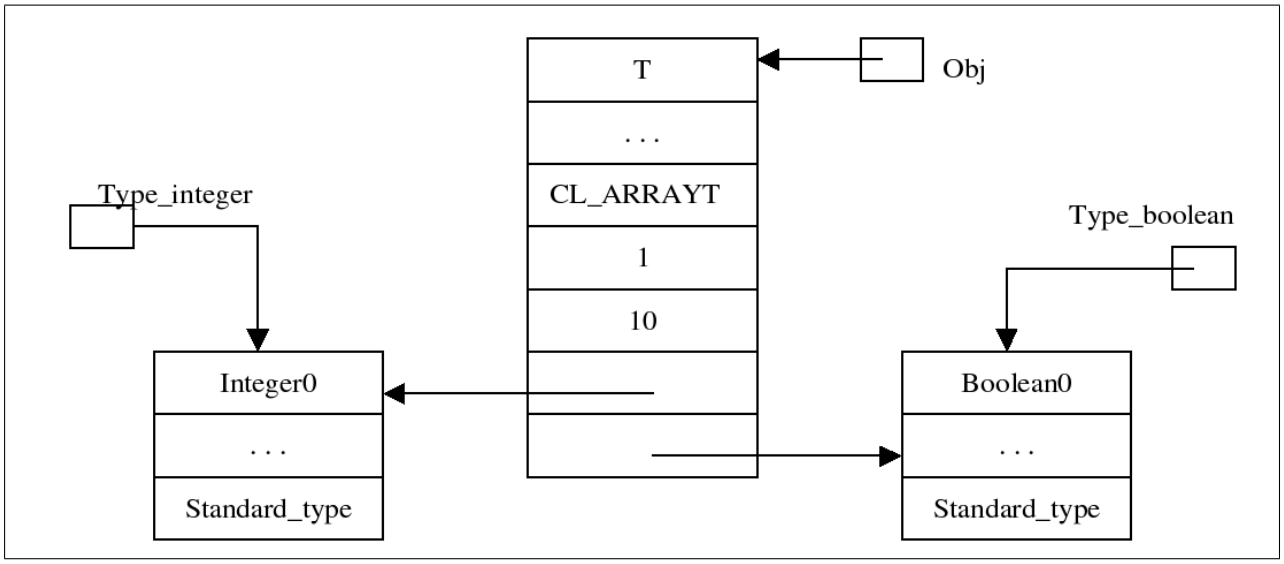

Figura 9.5: Estructura de la definición de tipos

```
        Error("Invalid type.");
      *type1 = TypeError;
    }
  }
```

Código 9.6: Tipos de los índices de un array

Si los dos tipos no son idénticos el analizador emitirá un error de tipo y sustituirá el primer tipo por `type_error` para eliminar los errores subsiguientes.

Para obtener el tipo de los elementos del array utilizar la función *type_name()* descrita anteriormente.

C) Reescribir la función *new_array_type()* para que reciba como parámetro el identificador del tipo array que se está definiendo, compruebe que en la definición no hay errores de tipo y cree el objeto asociado al nuevo tipo al final de la declaración.

```
void NewArrayType(ptr obj, set *stop);
```

Modificar la función *type_definition()* para que realice la llamada a *new_array_type()* de forma adecuada.

```
void TypeDefinition(set *stop);
```

Registros

Hasta ahora no se habían dado las reglas de ámbito para los registros. Se enuncian a continuación:

1. Regla 1: Todos los campos definidos por el mismo tipo registro deben tener identificadores diferentes.

2. Regla 2: Un campo `f` de una variable `x` se denota por `x.f` y es conocido solamente dentro del ámbito de `f`. Si el identificador `f` se utiliza de cualquier otra forma en el ámbito de `x`, éste denota a otro objeto.

En el registro `object_record` correspondiente a un `record` sólo se almacena un puntero al último de los campos definidos (`last_field`). Los campos se definen mediante otra clase de objetos en el que se almacena el tipo del campo (`field_type`).

```
typedef struct ObjectRecord {
  int name;
  class kind;

  union {
    . . .
      ptr FieldType; /* CL_FIELD */
      ptr LastField; /* CL_RECORD_TYPE */
  } KU; /* Kind Union */
} ObjectRecord;
```

Código 9.7: Estructura para el tipo registro

La figura 9.6 muestra la estructura de datos para el tipo:

```
type R = record f: boolean, g: T end;
```

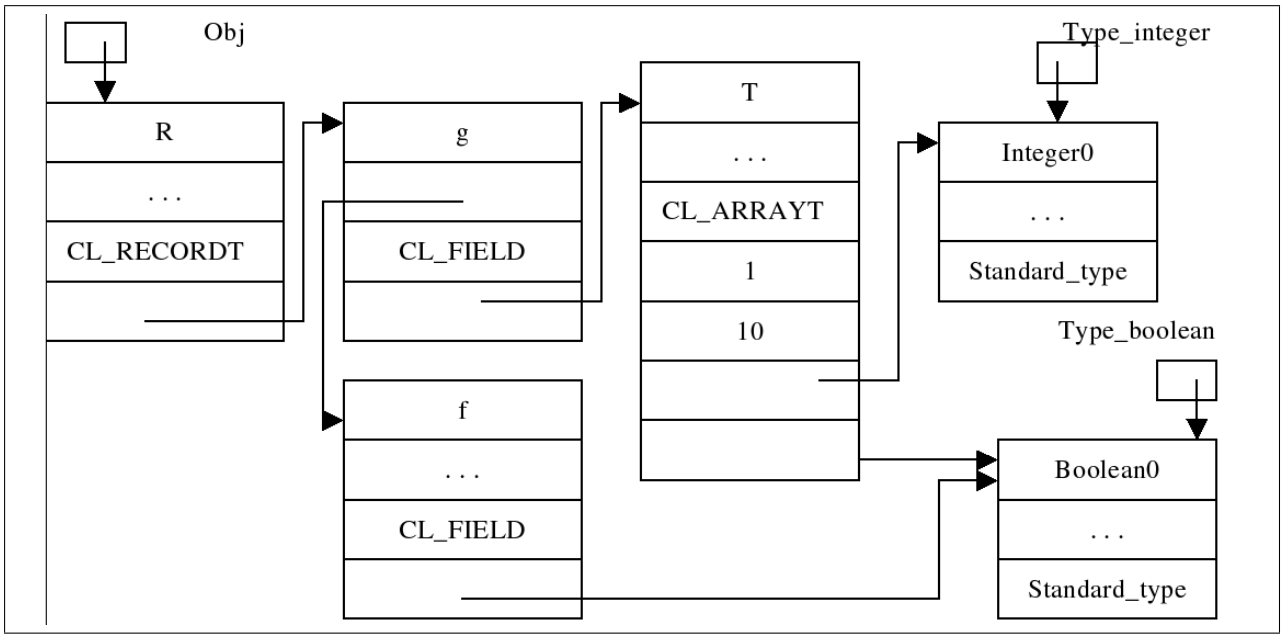

Figura 9.6: Estructura para el tipo registro

El compilador ha de comprobar que los campos de un tipo registro tengan identificadores diferentes y debe enlazarlos juntos en una lista. Las funciones de análisis de ámbito harán esto de forma automática si tratamos un nuevo tipo registro como un bloque

(utilizar *set()* y *block_level* en los puntos apropiados). Del mismo modo que *new_array_type()* recibe como parámetro el identificador del tipo que se está definiendo, *new_record_type()* también lo ha de recibir. La lista de campos se analiza de la misma forma que las listas de variables:

field_list → record_section { ; record_section }
record_section → field_name { , field_name } : type_name

Para analizar la lista de campos se sigue el mismo esquema que para la función *variable_group()*, con la salvedad de que en el caso de *field_list()* se crean objetos de clase `FIELD`.

D) Reescribir la función *new_record_type()*. Debería tener una definición similar a:

```
void NewRecordType(ptr obj, set *stop);
```

Las definiciones para las funciones *record_section()* y *field_list()* podrían ser:

```
void RecordSection(ptr *LastField, set *stop);
void FieldList(ptr *LastField, set *stop);
```

Accesos a variables

Una referencia a una variable se realiza mediante una llamada a la función *variable_access()*. Cuando el compilador encuentra el identificador de una variable, utiliza el correspondiente registro para determinar el tipo de la variable. Si el identificador de variable aparece aislado, se refiere a una variable no estructurada. Si una variable es de tipo array, el identificador de la variable puede ir seguido de una expresión que indique el índice del array seleccionado. Por último, si una variable es de tipo registro, el identificador puede ir seguido de otro identificador que indique el campo del registro seleccionado. El acceso a una variable tiene la siguiente sintaxis:

variable_access → variable_name [selector]
selector → indexed_selector | field_selector

La función *variable_access()* debe devolver un puntero al tipo de la variable (`typex`) para ser utilizado en el análisis de tipo (en caso de detectar un error, el tipo que se ha de devolver es `type_error`). Consideremos el siguiente ejemplo:

```
type T = array[1..10] of boolean;
var x: T; i: integer;
... x[i + 1] ...
```

Cuando se accede a una variable de la forma `x[i+1]` la función *variable_access()* toma el identificador de la variable (`x`) y obtiene un puntero al registro que describe su tipo (`T`). Se invoca entonces a la función *indexed_selector()* para analizar el selector de índice. Esta función recibe como parámetro el tipo de la variable `x` (es decir, `T`) y comprueba si es `ARRAYTYPE` con índices del mismo tipo que la expresión `i + 1` y devuelve el tipo

de `x[i+1]` que es boolean. De lo anterior se deduce que la función *expression()* ha de devolver un puntero al tipo de la expresión que se ha analizado.

Consideremos nuevamente *variable_access()* y el siguiente ejemplo:

```
type R = record f: integer; g: T end;
var x: R;
... x.f ...
```

El analizador sintáctico invoca a la función *variableAccess()* para cazar el identificador `x` y obtener el puntero a su tipo `R`. Luego se invoca a *field_selector()* para acceder a alguno de sus campos. La función *field_selector()* recibe como parámetro el tipo de la variable `x`. Al final dicho parámetro apunta al tipo del campo seleccionado a menos que se produzca un error en cuyo caso apunta a `type_error`.

E) Escribir la función *variable_access()* para que realice el análisis de tipo. (En este punto, hace falta la definición de la función *expression()*)

```
void Expresion(ptr *type, set *stop);
void IndexSelector(ptr *type, set *stop);
void FieldSelector(ptr *type, set *stop);
void VariableAccess(ptr *type, set *stop);
```

Procedimientos

En general, una definición de procedimiento está descrita por un registro de la forma que se muestra en el Código 9.8.

```
typedef struct ObjectRecord {
  int name;
  class kind;
  union {
      . . .
      ptr LastParam; /* CL_PROCEDURE */
  } KU; /* Kind Union */
} ObjectRecord;
```

Código 9.8: Definición de procedimiento

`LastParam` apunta al registro que describe el último parámetro del procedimiento. Dada la siguiente definición:

```
procedure P(x: integer; var y: boolean);
begin
end;
```

la figura 9.7 describe la estructura de datos utilizada para representar al procedimiento.

La definición de un procedimiento (*procedure_definition()*) tiene la sintaxis:

procedure_definition → procedure procedure_name procedure_block ;
procedure_block → [(formal_parameter_list)] ; block_body.

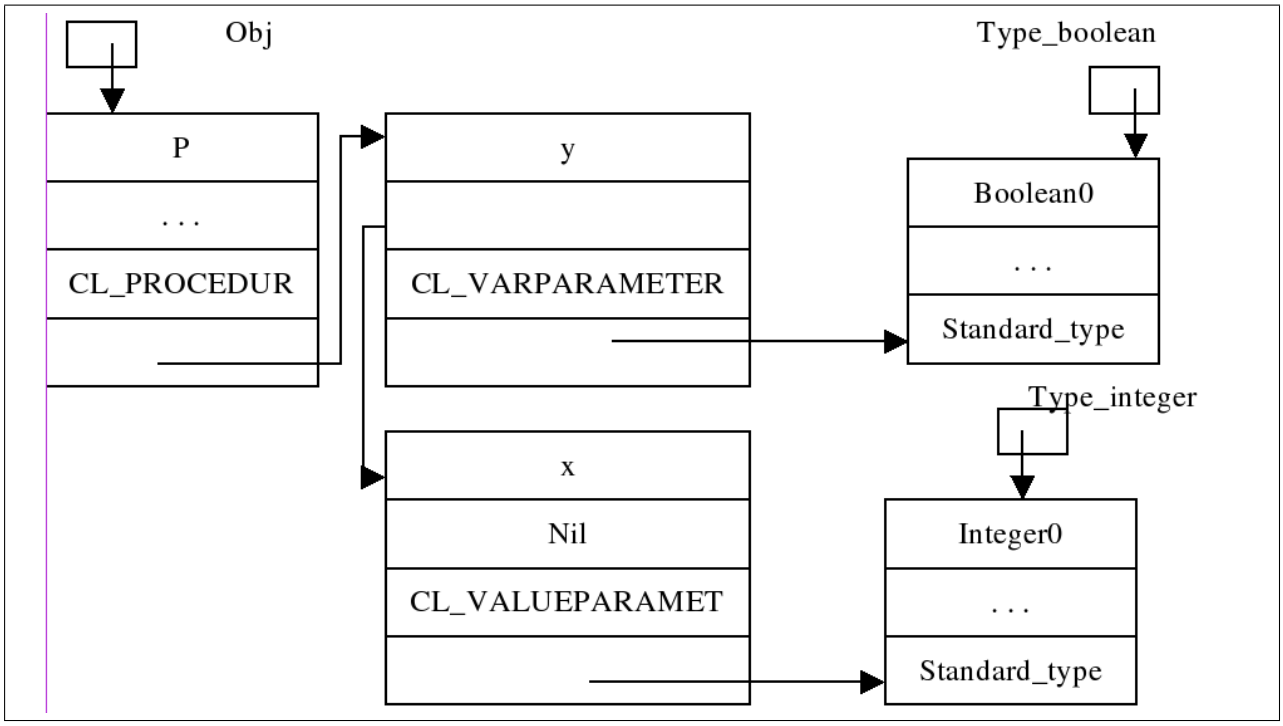

Figura 9.7: Estructura para la definición del procedimiento

Téngase en cuenta lo siguiente:

1. El registro del procedimiento se ha de crear al comienzo de la definición del procedimiento, para permitir llamadas recursivas en el bloque del procedimiento.

2. El bloque del procedimiento es tratado como un bloque separado. Los parámetros formales son locales al bloque del procedimiento, mientras que el procedimiento mismo es local al bloque inmediato superior. Esto se debe a que el identificador del procedimiento está definido antes de que el nuevo bloque sea analizado.

3. Si el procedimiento no tiene parámetros, el puntero a la secuencia de parámetros será `NULL`. Si el procedimiento tiene una lista de parámetros, el primer parámetro es también el primer objeto definido en el bloque del procedimiento. El resultado es que los parámetros formales de un procedimiento están siempre enlazados mediante una cadena de punteros que comienzan en el campo `last_param` del registro y finaliza con un puntero `NULL`. Posteriormente se utilizará esta cadena para realizar el análisis de tipo de las sentencias de llamada a procedimiento.

La función que analiza una lista de parámetros formales:

formal_parameter_list $\rightarrow$ paramenter_definiton $\{$; parameter_definition$\}$

devolverá un puntero al último registro que representa un parámetro. Los registros de los parámetros se crean uno de cada vez por medio de la función *parameter_definition()*.

F) Reescribir la función que analiza la definición de un procedimiento *procedure_definition()* para que invoque a *formal_parameter_list()* de la forma adecuada.

Para rellenar el campo `last_param` del registro que representa al procedimiento utilizar el puntero que se obtiene al analizar la lista de parámetros formales:

```
void FormalParameterList(ptr *LastParam, set *stop);
```

Expresiones

El tipo de una expresión está determinado por las funciones: *expression()*, *simple_expression()*, *term()* y *factor()*. Todas ellas han de devolver el tipo de la porción de la expresión que hayan analizado.

Revisemos en primer lugar *factor()*. Si se encuentra una expresión entre paréntesis, se invoca recursivamente a la función *expression()*. Si encuentra un operador `not` frente a un operando, *factor()* se invoca a sí misma. La función ha de comprobar que el operando del operador `not` es de tipo Boolean. Además se han de analizar los números con una llamada a la función *constant()* que ya devuelve el tipo integer. Al analizar un identificador ahora es posible distinguir entre el identificador de una constante en cuyo caso se invoca a *constant()* y el identificador de una variable en cuyo caso se invoca a *variable_access()*.

Un término se revisa utilizando el conjunto de tokens:

multiply_symbols = {AND, ASTERISK, DIV, MOD}

Esta función comprueba que los operadores aritméticos se apliquen solamente a operandos enteros y que el operador `and` tenga operandos booleanos. En otro caso se emite un error de tipo ('Invalid Type') y se sustituye el tipo erróneo por el tipo `type_error`. Si el tipo ya era `type_error` se suprime el mensaje de error. Esto se logra mediante una llamada a la función *TypexError()* que se muestra en el Código 9.9.

```
void TypexError(ptr *type) {
  if (*type != TypeError) {
    Error("Invalid Type.");
    *type1 = TypeError;
  }
}
```

Código 9.9: La función TypexError

El resto de las funciones para analizar expresiones se modifican del mismo modo.

G) Reescribir las funciones:

```
void Factor(ptr *type, set *stop);
void Term(ptr *type, set *stop);
void SimpleExpresion(ptr *type, set *stop);
void Expresion(ptr *type, set *stop);
```

Sentencias

Las sentencias tienen la siguiente sintaxis:

statement → assign_statement | procedure_statement | if_statement | while_statement | compound_statement | ϵ

La función que el compilador utiliza para analizar sentencias es:

```
void Statement(set *stop);
```

Puesto que ahora se puede distinguir entre el identificador de un procedimiento y el de una variable, se hará en cada caso la llamada que corresponda a las funciones *assign_statement()* o *procedure_statement()*.

En una sentencia de asignación (*assign_statement()*):

assign_statement → variable_access := expression

la variable del lado izquierdo y la expresión deben ser del mismo tipo. Los tipos son determinados por las funciones *variable_access()* y *expression()*. El compilador utiliza la descripción de un procedimiento para analizar las sentencias de llamada procedimiento (*procedure_statement()*). El código del Código 9.10 muestra una llamada al procedimiento P.

```
var a: integer; b: boolean;
  ...P(a + 1, b)...
```

Código 9.10: Llamada a P

Primero el compilador encuentra el registro que describe al procedimiento P. Entonces sigue la cadena de registros de los parámetros y revisa lo siguiente:

- El número de parámetros actuales en la sentencia de llamada a procedimiento debe de ser igual al número de parámetros formales en la definición del procedimiento (en nuestro ejemplo, dos).
- El parámetro actual `a + 1` debe ser una expresión del mismo tipo que el parámetro por valor `x` (`type_integer`).
- El parámetro actual `b` debe ser un acceso a variable del mismo tipo que el parámetro por varible `y` (`type_boolean`).

Una sentencia de llamada a procedimiento tiene la siguiente sintaxis:

procedure_statement → IO_statement | procedure_name[(actual_parameter_list)]

Por tanto, se han de distinguir dos clases de procedimientos:

1. Procedimientos estándar
2. Procedimientos que son definidos por el usuario.

En el caso de que se trate de un procedimiento estándar se invocará a una función similar a la que se muestra en el Código 9.11.

```
void IO_Statement(ptr obj, set *stop) {
     ptr type, Id;

     if (obj->name == READ) {
  if (LookAhead != ID)
     Error("Integer variable expected as paremeter.");
  MatchRefID(&Id, set_union(stop, ENDS));
  type = Id;
  VariableAccess(&type, set_union(stop, ENDS));
  if (type != TypeInteger)
     Error("Integer variable expected as paremeter.");
     }
     else { /* WRITE */
  Expresion(&type, set_union(stop, ENDS));
  if (type != TypeInteger)
     Error("Integer expresion expected as paremeter.");
     }
}
```

Código 9.11: La función IO_Statement

La parte que requiere más atención en el análisis de una sentencia de llamada a procedimiento está en el análisis del tipo de los parámetros actuales. Puesto que los parámetros actuales se leen de izquierda a derecha, se deben examinar los parámetros formales en ese mismo orden. Como los restantes objetos, los parámetros formales, están descritos por registros que están enlazados en orden inverso comenzando por el último parámetro. Para solucionar este problema, redefinimos la sintaxis de una lista de parámetros actuales por una regla recursiva:

actual_parameter_list → [actual_parameter_list ,] actual_paramenter
actual_paremeter → expression | variable_access

En una sentencia *while* (*while_statement()*) y en una sentencia *if* (*if_statement()*) se ha de revisar que la expresión sea de tipo boolean.

while_statement → **while** expression **do** statement
if_statement → **if** expression **then** statement [**else** statement]

H) Teniendo en cuenta los párrafos anteriores, reescribir las siguientes funciones:

```
void Statement(set *stop);
void AssignStatement(ptr obj, set *stop);
void ActualParameterList(ptr param, set *stop);
void ProcedureStatement(ptr obj, set *stop);
void WhileStatement(set *stop);
void IfStatement(set *stop);
```