Procesadores de Lenguajes

Compilador de un subconjunto de Java
Universidad Rey Juan Carlos
4º Ingeniería Informática

Diego Chaparro González
José Pelegrín López
[Grupo 02]

25 de septiembre de 2001

Índice General

Índice de Tablas

Índice de Figuras

Listings

Capítulo 1

Introducción

Hemos elegido como lenguaje para la implementación del compilador *C++*. Las razones para esta decisión han sido la facilidad que este lenguaje ofrece para el tratamiento de las estructuras de datos con el uso de la *Standard Template Library*, así como la orientación a objetos que ha sido el paradigma elegido para la implementación del compilador, además de la posibilidad de usar excepciones en el tratamiento de los errores.

El compilador, será por tanto una clase y compilar será uno de sus métodos, como se puede observar en la siguiente especificación:

```
class CCompilador {
  public:
  CCompilador(const COpciones& op);
  ~CCompilador(void);
  void Compilar(void);
  };
```

Al compilador, se le pasan unas opciones, que generalmente son la recogidas de la línea de comandos, en la creación de su objeto correspondiente, con lo que el uso del compilador es:

```
COpciones* opciones = new COpciones(argc, argv);

CCompilador* compilador = new CCompilador(*opciones);
compilador->Compilar();
```

Capítulo 2

Descripción

2.1 Analizador léxico

El analizador léxico se realizará con la variante de *lex* para *C++*, *flex++*.

El analizador léxico va reconociendo cada uno de los tokens [1] del lenguaje Java. Este analizador proporcionará lo tokens al analizador sintáctico por medio de una subrutina.

En nuestro caso concreto y al estar utilizando *flex++* la declaración y la utilización del analizador léxico en el sintáctico se ha implementado como:

```
yyFlexLexer* analizador_lexico =
  new yyFlexLexer((istream*)entrada);

int token = analizador_lexico->yylex();

string cadena_token = analizador_lexico->YYText();
```

Este analizador_lexico es uno de los atributos que forman parte del compilador.

La gramática utilizada para generar el analizador léxico, puede verse en la tabla 2.1. El fichero que genera el analizador léxico por medio de *flex++* puede verse en el fichero entregado como `jcc-analex.lex`

2.2 Tabla de símbolos.

2.2.1 Contenido

La tabla de símbolos mantendrá la información de aquellos lexemas que requieran de mantenerla durante el tiempo de la compilación. Debido a la naturaleza de la práctica, hemos decidido almacenar la información de variables, constantes y clases.

```
typedef enum { L_VARIABLE, L_CONSTANTE, L_CLASE } TipoLexema;
```

Por lo tanto para cada lexema tendremos que guardar información acerca de la cadena que lo identifica, el tipo de lexema que es y los datos que guardaremos en relación con su tipo.

[1]Los tokens se representarán (en la implementación) como enteros. Puede observarse esto en el fichero `data.hpp`

```
<Digito>            ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Octal>             ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<Letra>             ::=  a | A | b | B | c | C | d | D | e | E
                         | f | F | g | G | h | G | i | I | j | J
                         | k | K | l | L | m | M | n | N | o | O
                         | p | P | q | Q | r | R | s | S | t | T
                         | u | U | v | V | w | W | x | X | y | Y
                         | z | Z
<Hexadecimal>       ::=  <Digito> | a | A | b | B | c | C
                         | d | D | e | E | f | F
<Car>               ::=  Unicode
<Comentario>        ::=  //<RestoComentario>\n
<RestoComentario>   ::=  <Car><RestoComentario>
                         | <Car>
<CteOctal>          ::=  0<RestoCteOctal>
<RestoCteOctal>     ::=  <Octal>
                         | <Octal><RestoCteOctal>
<CteDecimal>        ::=  0x<RestoCteHex>
<RestoCteHex>       ::=  <Hexadecimal>
                         | <Hexadecimal><RestoCteHex>
<CteLogica>         ::=  true | false
<Cadcaract>         ::=  ‘‘<Car>’’
<AutoIncr>          ::=  ++
<AutoDecr>          ::=  --
<OperArit>          ::=  + | - | * |   | %
<OperRelac>         ::=  == | != | < | > | <= | >=
<OperLogi>          ::=  && | || | !
<Asig>              ::=  =
<OpPunto>           ::=  .
<TInt>              ::=  int
<TBoolean>          ::=  boolean
<TVoid>             ::=  void
<PyC>               ::=  ;
<Parent>            ::=  ( | )
<Llaves>            ::=  { | }
<Return>            ::=  return
<Class>             ::=  class
<Public>            ::=  public
<Private>           ::=  private
<While>             ::=  while
<Do>                ::=  do
<Id>                ::=  <Letra><RestoId> | $<RestoId>
                         | _<RestoId>
<RestoId>           ::=  <Letra> | <Digito> | _
                         | <Letra><RestoId> | <Digito><RestoId>
                         | _<RestoId>
```

Tabla 2.1: Gramática del analizador léxico

```
class CLexema {
private:
  string      identificador;
  TipoLexema  tipo;
  union {
    CVariableDatos*  vd;   // tipo == L_VARIABLE
    CConstanteDatos* cod;  // tipo == L_FUNCION
    CClaseDatos*     cld;  // tipo == L_CLASE
  };
};
```

Por cada tipo de lexema que se guardará en la tabla tenemos diferente información. Para un identficador, guardaremos su tipo de datos y si es una variable o una constante, para una función guardamos, el tipo de su valor de retorno, y los parámetros que posee dicha función. En cuanto a una clase, guardaremos información acerca de los atributos que posee y lo métodos para acceder a ella. De los atributos y métodos de una clase, también guardamos el método de acceso a ellos.

```
class CVariableDatos {
private:
  string tipo_de_datos;
  string valor;
};

class CConstanteDatos {
private:
  string tipo_de_datos;
  string valor;
};

class CClaseDatos {
private:
  TipoAcceso            acceso;
  list<Tipo_Atributo>   atributos;
  list<Tipo_Metodo>     metodos;
};
```

2.2.2 Estructura

La tabla de símbolos se va a implementar como un conjunto de palabras reservadas, la palabras reservadas del lenguaje Java que se trata en esta práctica, una pila de lexemas que contendrá todos los lexemas accesible en cada momento de la compilación y una nueva pila que determinará el ámbito donde empieza y donde acaba cada conjunto de lexemas declarados en cada bloque, con el fin de que cada vez que un ámbito concluya, se elimina de la pila los lexemas de ese ámbito.

```
class CTablaSimbolos
{
  stack<CLexema> ambito;
  stack<int>     bloques;
  set<string>    palabras_reservadas;
};
```

2.2.3 Acceso

La entrada de los valores a la tabla de símbolos, puede establecerse cuando se recibe la primera vez, esto es, cuando el analizador léxico detecta el lexema, y se rellena la información cuando se dispone de ella, esto es, cuando el analizador sintáctico determina el papel que juega el lexema.

En el analizador léxico, sólo podríamos ir introduciendo los identificadores de cada lexema encontrado pero nada de información, por lo que sería más razonable sólo introducir lexemas cuando sepamos el papel que juega dentro de el código que se está compilando y eso sólo podemos hacer en tiempo de análisis sintáctico.

A la tabla de símbolos se le asocian una serie de métodos para su utilización.

- CTablaSimbolos()
  El constructor de la tabla de símbolos, introduce las palabras reservadas e inicializa la pila de ámbitos.
- Inserta(**const** CLexema& lexema)
  Inserta en la tabla de símbolos un nuevo lexema. Si el identificador es una palabra reservada, entonces se devuelve un error.
- Busca(**const** string& id_lexema)
  Busca un lexema en la tabla de símbolos coincidente con el identificador dado. Se devuelve `true` si existe. Por supuesto, esto incluye a las palabras reservadas.
- Recupera(**const** string& id_lexema)
  Recupera el primer lexema completo que se encuentra dentro de la tabla de símbolos.
- Modifica(**const** CLexema& lexema)
  Sustituye la información del primer lexema encontrado, por la información del lexema dado.
- PushBloque()
  Crea un nuevo ámbito.
- PopBloque()
  Deshabilita el ámbito actual.

Por último destacar que la tabla de símbolos será una propiedad de la clase CCompilador dada por la tabla_simbolos.

2.3 Analizador Sintáctico

El **análisis descendente con tablas**, es el algoritmo que se implementa para el análisis sintáctico. Debido a que no existen herramientas automáticas para este tipo de análisis, la implementación, que pasamos a explicar, ha partido de cero.

2.3.1 Gramática

Este tipo de algoritmo requiere que la gramática cumpla las condiciones LL(1) para ser correcto, con lo que a la gramática tomada como referencia la tuvimos que eliminar de recursividad (indirecta) por la izquierda, así como en muchas de las producciones, realizar factorización debido a que existían símbolos en las cabeceras de los consecuentes de dichas producciones que producían indecisión en el algoritmo. Con todo eso, la gramática final utilizada se puede ver en las tablas 2.2, 2.3.

2.3.2 Algoritmo

El algoritmo utilizado está tomado de la página 192 del libro de *Compiladores: Principios, técnicas y herramientas* [2], que se puede observar en pseudocódigo:

```
apuntar ae al primer símbolo de w$;
repeat
  sea X el símbolo de la cima de la pila y
    a el símbolo apuntado por ae;
  if X es un terminal o $ then
    if X = a then
      extraer X de la pila y avanzar ae
    else error()
  else /* X es un no terminal */
    if M[X,a] = X -> Y1 Y2 ... Yk then begin
      extraer X de la pila;
      meter Yk, Yk-1, ... Y1 en la pila, con Y1 en la cima;
      emitir la producción X -> Y1 Y2 ... Yk
    end
    else error()
until X = $ /* la pila está vacía */
```

Esto será implementado en el método Compilar de CCompilador.

En nuestra implementación, los siguientes atributos son relevantes dentro de la clase CCompilador para el análisis sintáctico:

```
class CCompilador {
private:
  NoTerminales      no_terminales;      // Conjunto de no terminales.
  Terminales        terminales;         // Conjunto de terminales.
  TablaAS           tabla_as;           // Tabla LL(1).
  Gramatica         gramatica;          // Gramática utilizada.
  stack<string>     pila_gramatical;    // Pila de símbolos gramaticales.
};
```

2.3.3 Tabla de análisis LL(1)

Con la gramática cumpliendo LL(1) y los conjuntos siguiente de todas las producciones, hemos elaborado la tabla (2.3.3, 2.3.3,2.3.3,**??**) para el análisis sintáctico.

[2] Aho, Alfred V. Sethi, Ravi. Ullman, Jeffrey D. *Compiladores: Principios, técnicas y herramientas.* Addison Wesley Longman, 1998.

```
compilation_unit                ::=  lista_type_declaration
lista_type_declaration          ::=  type_declaration lista_type_declaration
                                     | λ
type_declaration                ::=  class_declaration
class_declaration               ::=  modifier ``class''
                                     identifier ``{`` lista_field_declaration ``}''
lista_field_declaration         ::=  field_declaration lista_field_declaration
                                     | λ
modifier                        ::=  ``public``
                                     | ''private''
                                     | λ
field_declaration               ::=  modifier type field_declaration_prima
                                     | '';''
field_declaration_prima         ::=  constructor_declaration
                                     | identifier field_declaration_prima_prima
field_declaration_prima_prima   ::=  variable_declaration
                                     | method_declaration
method_declaration              ::=  ``(`` opcional_parameter_list ``)''
                                     lista_corchetes method_declaration_final
opcional_parameter_list         ::=  parameter_list
                                     | λ
lista_corchetes                 ::=  ``[`` ``]'' lista_corchetes
                                     | λ
method_declaration_final        ::=  statment_block
                                     | '';''
type                            ::=  type_specifier lista_corchetes
type_specifier                  ::=  ``void''
                                     | ''boolean''
                                     | ''int''
                                     | class_name
class_name                      ::=  identifier
parameter_list                  ::=  parameter lista_parameter_list
lista_parameter_list            ::=  ``,'' parameter lista_parameter_list
                                     | λ
parameter                       ::=  type identifier lista_corchetes
statement_block                 ::=  ```` lista_statment ``''
lista_statment                  ::=  statment lista_statment
                                     | λ
```

Tabla 2.2: Gramática de Análisis Sintáctico

```
statment                        ::=  identifier posible_statment ``;''
                                     | ``void'' identifier variable_declaration ``;''
                                     | ``int'' identifier variable_declaration ``;''
                                     | ``boolean'' identifier variable_declaration ``;''
                                     | statment_block
                                     | do_statment
                                     | ``return'' opcional_expresion ``;''
                                     | ``;''
                                     | ``const posible_const ``;''
opcional_expresion              ::=  expresion
                                     | λ
variable_declaration            ::=  variable_declarator lista_variable_declarator
lista_variable_declarator       ::=  ``,'' identifier variable_declarator lista_variable_declarator
                                     | λ
variable_declarator             ::=  lista_corchetes opcional_variable_initializer
                                     | ``(`` opcional_arglist ``)''
opcional_variable_initializer   ::=  ``='' variable_initializer
                                     | λ
variable_initializer            ::=  expresion
expresion                       ::=  expresion_1 expresion_prima
expresion_prima                 ::=  op_indecremento expresion_prima
                                     | op_aritmetico expresion_1 expresion_prima
                                     | op_relacional expresion_1 expresion_prima
                                     | op_logico expresion_1 expresion_prima
                                     | ``.''  expresion_1 expresion_prima
                                     | λ
expresion_1                     ::=  expresion_2 expresion_1_prima
expresion_1_prima               ::=  expresion_3 expresion_1_prima
                                     | λ
expresion_2                     ::=  constante_logica
                                     | literal_expresion
                                     | identifier
                                     | ``(`` expresion ``)''
expresion_3                     ::=  agrupa_expresion
op_incremento                   ::=  ++
                                     | −−
op_aritmetico                   ::=  +
                                     | −
                                     | *
                                     | /
                                     | %
op_relacional                   ::=  >
                                     | <
                                     | >=
                                     | <=
                                     | ==
                                     | !=
op_logico                       ::=  &&
                                     | ||
agrupa_expresion                ::=  ``(`` opcional_arglist ``)''
                                     | ``[`` expresion ``]''
opcional_arglist                ::=  arglist
                                     | λ
literal_expresion               ::=  integer_literal
                                     | string
integer_literal                 ::=  ``octal''
                                     | ``decimal''
                                     | ``hexadecimal''
string                          ::=  ``cadena_caracteres''
arglist                         ::=  expresion lista_expresion
lista_expresion                 ::=  ``,'' expresion lista_expresion
                                     | λ
do_statment                     ::=  ``do'' statment ``while'' ``(`` expresion ``)'' ``;''
constructor_declaration         ::=  ``(`` opcional_parameter_list ``)'' statment_block
constante_logica                ::=  ``true''
                                     | ``false''
opcional_identifier             ::=  ``.''  identifier
                                     | λ
op_asignacion                   ::=  ``=''
posible_statment                ::=  expresion_prima
                                     | agrupa_expresion
                                     | op_asignacion expresion
                                     | identifier variable_declaration
posible_const                   ::=  ``int'' identifier ``='' integer_literal
                                     | ``boolean'' identifier ``='' constante_logica
```

Tabla 2.3: Gramática de Análisis Sintáctico

	public	private	class	identifier	void	boolean	int	(	)	[	]	{	}
compilation_unit													
lista_type_declaration													
type_declaration													
class_declaration													
lista_field_declaration 6	6						7			6			
modifier 10	10												
identifier													
field_declaration 12	12									13			
field_declaration_prima		14											
field_declaration_prima_prima		17		16				16		16			
method_declaration		18											
opcional_parameter_list 19	19		20										
lista_corchetes		22	22	21		22		22		22	22		
method_declaration_final						23				24			
type 25	25												
type_specifier 27	28												
class_name													
parameter_list 31	31												
lista_parameter_list			33					32					
parameter 34	34												
statment_block						35							
lista_statment 36	36	36				36	37			36		36	36
statment 41	40					42				45			
opcional_expresion		46								47		46	46
variable_declaration		48	48	48				48		48	48		
lista_variable_declarator								49		50			
variable_declarator		112	51	51				51		51	51		
opcional_variable_initializer								53		53	52		
variable_initializer		54										54	54

	,	.	;	=	decimal	octal	hex	cad_car	do	return	const	true	false	$
compilation_unit	1	1	1											
lista_type_declaration	2	2	2											3
type_declaration	4	4	4											
class_declaration	5	5	5											
lista_field_declaration	6	6		6	6									
modifier	6	6		6	6									
identifier				11										
field_declaration	12	12		12	12									
field_declaration_prima				15										
field_declaration_prima_prima														
method_declaration														
opcional_parameter_list				19	19									
lista_corchetes				22										
method_declaration_final														
type				25	25									
type_specifier				29	26									
class_name				30										
parameter_list				31	31									
lista_parameter_list														
parameter				34	34									
statment_block														
lista_statment				36	36		36	36	36	36	36	36	36	
statment				38	39				43	44	109			
opcional_expresion				46			46	46				46	46	
variable_declaration				48										
lista_variable_declarator														
variable_declarator				51										
opcional_variable_initializer														
variable_initializer				54			54	54				54	54	

	identifier	boolean	int	(	)	[	]	{	}	,	.	;	=	decimal	octal
expresion	55			55										55	55
expresion_prima					61		61			61	60	61			
expresion_1	62			62										62	62
expresion_1_prima				63	64	63	64			64	64	64			
expresion_2	67			68										66	66
expresion_3				69		69									
op_indecremento															
op_aritmetico															
op_relacional															
op_logico															
agrupa_expresion				85		86									
opcional_arglist	87			87	88									87	87
literal_expresion														89	89
integer_literal														92	91
string															
arglist	95			95										95	95
lista_expresion					97					96					
do_statment															
constructor_declaration				99											
constante_logica															
opcional_identifier											102		103		
op_asignacion													104		
posible_statment	108			106	105	106	105			105	105	105	107		
posible_const		111	110												

	hex	cad_car	do	++	−−	+	−	*	/	%	>	<	>=	<=	==	!=	&&	||	true	false
expresion	55	55																	55	55
expresion_prima				56	56	57	57	57	57	57	58	58	58	58	58	58	59	58		
expresion_1	62	62																	62	62
expresion_1_prima				64	64	64	64	64	64	64	64	64	64	64	64	64	64	64		
expresion_2	66	66																	65	65
expresion_3																				
op_indecremento				70	71															
op_aritmetico						72	72	74	75	76										
op_relacional											77	78	79	80	81	82				
op_logico																	84	83		
agrupa_expresion																				
opcional_arglist	87	87																	87	87
literal_expresion	89	90																		
integer_literal	93																			
string		94																		
arglist	95	95																	95	95
lista_expresion																				
do_statment			98																	
constructor_declaration																				
constante_logica																			100	101
opcional_identifier																				
op_asignacion																				
posible_statment				105	105	105	105	105	105	105	105	105	105	105	105	105	105	105		
posible_const																				

2.4 Analizador Semántico

El analizador semántico se ha implementado, incluyendo símbolos en la pila del análisis sintáctico (pila_gramatical) obliguen al algoritmo a realizar las acciones semánticas pertinentes.

Para ello cuando debemos incluir acciones semánticas, modificamos la gramática, de tal forma, que no sólo tendrá símbolos en sus producciones terminales o no terminales, además aparecerán símbolos semánticos. Por ejemplo:

```
class_declaration ::=
  modifier "class" identifier S_5_2
  "{" S_5_1 lista_field_declaration "}" S_5_0
```

Con esto, debemos modificar el algoritmo implementado en Compilar teniendo en cuenta que pueden existir símbolos semánticos en la pila de símbolos gramaticales y aplicar la acción que el símbolos nos indique.

Para realizar el análisis semántico, se han utilizado tanto atributos sintetizados como heredados, y la forma de ir obteniendo los valores para cada producción, ha sido simulando el recorrido en profundidad del árbol sintáctico con la ayuda de la pila pila_producciones.

2.5 Generación de Código

Al igual que para el análisis semántico, para la generación de código se han introducido símbolos en la pila del análisis sintáctico(pila_gramatical), para ir realizando las acciones necesarias para la generación del código intermedio correspondiente.

Después de cada produccion se introduce un símbolo para la generación del código correspondiente:

```
class_declaration ::=
modifier "class"identifier S_5_2
"{"S_5_1 lista_field_declaration "}"S_5_0 G_5_0
```

Y el código correspondiente a este símbolo se ejecuta cuando ya se han reconocido todos los símbolos de la producción.

De esta forma lo que conseguimos son atributos sintetizados. Se va construyendo el árbol sintáctico y el código se va generando de abajo hacia arriba. El árbol se recorre en profundidad y cuando se va subiendo el código generado y los atributos necesarios se van propagando hacia arriba en el árbol sintáctico.

Al final lo que conseguimos es tener todo el código generado en la producción raiz del árbol sintáctico, y en ese momento dejamos de generar código.

Para ayudarnos a realizar este trabajo utilizamos una pila auxiliar, que la que se van almacenando las producciones que hemos ido derivando. De esta forma podemos ir guardando algunos atributos de algunas producciones para luego poder utilizarlos.

También se hace uso de la tabla de símbolos para conseguir información de alguno de los tokens reconocidos, tales como funciones, variables ...

2.6 Manejo de errores

Como ya se especificó en la introducción, el tratamiento de errores es implementado con las excepciones que posee el lenguaje. Cada parte de análisis puede elevar excepciones, que serán recogidas en el momento de la compilación y dependiendo del error se podrá dejar continuar para poder analizar más código, o propagar la excepción.

Si un error ha ocurrido, esto será notificado utilizando el atributo hubo_error de CCompilador, de esta forma la fase de síntesis podrá saber si debe continuar con su tarea o abortarla.

Cada vez que un error aparezca se dará información acerca del fichero que se está compilando, la línea donde el error se produjo y una cadena explicativa. El formato elegido es el siguiente:

```
fichero:número_de_línea:error
```

Un ejemplo:

```
casos8.java:28:Semántico:Es método privado.
```

Se ha elegido este formato por ser la forma en que aparecen los errores con el compilador *gcc*, y además en el editor que hemos utilizado (XEmacs), este tipo de cadena, hace que pulsando sobre esta línea, el cursor se mueva hacia el fichero y línea indicados, lo cual es una ventaja.

Capítulo 3

Casos de prueba

A continuación se presentan 8 casos de prueba. Estos ejemplos muestran programas en Java. Cuatro de ellos son válidos para el analizador léxico, y otros 4 no son válidos para el analizador léxico.

3.1 Casos válidos

3.1.1 Ejemplo 1

Objetivos

- Comprobar los tipos

```
public class clase1 {
    public int funcion1(int par1)
    {
        int par3;
        par3 = par1 ++;
    }
   public int funcion2 (int par2, boolean par3)
   {
        par2 = (par2 + 4) - 3;
        return par2;
   }
}

public class clase2 {

   public void  funcion0()
   {
        print ("hola mundo");
   }

   public void funcion1()
   {
        clase1 c1;
        int uno, dos;
        int tres;

        c1.funcion2();
        tres = dos;
```

```
    }

    public void main()
    {
        int mayor, menor;
        boolean pepe;

        pepe = mayor > menor;
    }
}
```

Resultado

```
goto MAIN

clase1funcion1:
pop par1
par3 = par1
tem1 =  1
par1 = par1 + tem1
RETURN


clase1funcion2:
pop par3
pop par2
tem2 = 4
tem3 = par2 + tem2
tem4 = 3
tem5 = tem3 - tem4
par2 = tem5
push par2
RETURN


clase1:
RETURN


clase2funcion0:
tem6 = "hola mundo"
OUTPUTCHAR tem6
RETURN


clase2funcion1:
call clase1
call clase1funcion2
pop tem7
call clase2c1
```

```
tres = dos
RETURN


MAIN:
tem8 = mayor > menor
pepe = tem8
RETURN


clase2:
RETURN
```

3.1.2 Ejemplo 2

Objetivos

- Comprobar las sentencias.

```
public class Suma {
    Suma (int car1)
    {
        int i;
        i = i + 1;
    }
    public int Resta () {
        int num;
        return num - 1;
    }
}

public class Resta {
    int restar ()
    {
        int j;
        j = 3 - 4;

        return j;
    }

    void main()
    {
        Suma s1 (3), s2;
        int entero;

        entero = (s2.Resta()) * 17;
        entero = entero + 30;
    }
}
```

Resultado

```
goto MAIN

Suma:
```

```
pop car1
tem1 = 1
tem2 = i + tem1
i = tem2
RETURN


SumaResta:
tem3 = 1
tem4 = num - tem3
push tem4
RETURN


Restarestar:
tem5 = 3
tem6 = 4
tem7 = tem5 - tem6
j = tem7
push j
RETURN


MAIN:
tem8 = 3
push tem8
call Suma
call Suma
call SumaResta
pop tem9
tem10 = 17
tem11 = tem9 * tem10
entero = tem11
tem12 = 30
tem13 = entero + tem12
entero = tem13
RETURN


Resta:
RETURN
```

3.1.3 Ejemplo 3

Objetivos

- El acceso a las clases.

```
public class clase1 {
```

```
        public int funcion1 (int j) {
            j = 3;
        }


    }

    public class clase2 {
        public int i;

        public void funcion2 () {
            clase1 c1;
        }

        public void main () {

            funcion2();
        }

    }
```

Resultado

```
goto MAIN

clase1funcion1:
pop j
tem1 = 3
j = tem1
RETURN


clase1:
RETURN


clase2funcion2:
call clase1
RETURN


MAIN:
call clase2funcion2
RETURN


clase2:
RETURN
```

3.1.4 Ejemplo 4

Objetivos

- Comprobar las sentencias.

```
public class Clase1 {
    public int i;
    public boolean vulean;
    private int j;

    public int Imprime_Public(int j)
    {
        print ("hola");
        print(3);
    }

    private boolean Imprime_Private(int j)
    {
        int otro_entero;
        int var;

        read();

        return true;
    }

    public boolean Nada()
    {
        int i = 0;
        Imprime_Private(3);
    }
}

public class Principal {
    public void main()
    {
        Clase1 c1, c2 ;
        const int constante_entera = 3;
        const boolean constante_vuleana = true;
        int h1;
        int h2, h3, h4;

        h1 = 1;
        h2 = 2;

        h1 = 1 + 3;
    }
}
```

Resultados

```
call Clase1Imprime_Private
pop tem9
RETURN


Clase1:
RETURN
```

```
MAIN:
call Clase1
call Clase1
tem10 = 3
constante_entera = tem10
tem11 = 1
constante_vuleana = tem11
tem12 = 1
h1 = tem12
tem13 = 2
h2 = tem13
tem14 = 1
tem15 = 3
tem16 = tem14 + tem15
h1 = tem16
RETURN


Principal:
RETURN
```

3.2 Casos no válidos

3.2.1 Ejemplo 5

Objetivos

- Comprobar las sentencias.

```
public class clase1 {
    public int funcion1(int par1)
    {
        boolean par3;
        par3 = par1 ++;
    }
   public int funcion2 (int par2, boolean par3)
   {
        par2 = (par2 + 4) - 3;
        return par3;
   }
}

public class clase2 {

   public void  funcion0()
   {
        print ("hola_mundo");
   }

   public void funcion1()
   {
        clase1 c1;
        int uno, dos;
        int tres;
```

```
        c1.funcion2();
        tres = c1;
    }

    public void main()
    {
        int mayor, menor;
        boolean pepe;

        pepe = pepe > menor;
    }
}
```

Resultados

```
../jcc 5.java
5.java:9:Semántico: Violación de tipos.
5.java:15:Semántico: Violación de tipos.
5.java:32:Semántico: Violación de tipos.
5.java:40:Semántico: Violación de tipos.
5.java:40:Semántico: Violación de tipos.
Error en la compilación. Tue Sep 23 22:27:16 2001
```

3.2.2 Ejemplo 6

Objetivos

- Comprobar las sentencias.

```
public class Suma {
    Suma (int car1)
    {
        int i;
        i = i + 1;
        return i;
    }
    public int Resta () {
        int num;
        return num - 1;
    }
}

public class Resta {
    int restar ()
    {
        int j;
        j = 3 - 4;

        return j;
    }

    void main()
    {
        Suma s1 (3), s2;
        int entero;
```

```
        entero = (s2.resta()) * 17;
        s1.pepe();
        entero = entero + 30;

        entero = ppp;
    }
}
```

Resultados

```
../jcc 6.java
6.java:11:Semántico: Violación de tipos.
6.java:32:Semántico: Variable no declarada.
6.java:32:Semántico: Miembro no declarado.
6.java:33:Semántico: Variable no declarada.
6.java:33:Semántico: Miembro no declarado.
6.java:36:Semántico: Variable no declarada.
6.java:36:Semántico: Violación de tipos.
Error en la compilación. Tue Sep 23 22:28:46 2001
```

3.2.3 Ejemplo 7

Objetivos

- Comprobar las sentencias.

```
public class clase1 {

    private int funcion1 (int j) {
        j = true;
    }


}

public class clase2 {
    public int i;

    public void funcion2 () {

        clase1 c1;
        clase3 c3;

        c1.funcion1();
    }

    public void mainnn () {

        funcion2();
    }

}
```

Resultados

```
../jcc 7.java
7.java:8:Semántico: Violación de tipos.
7.java:22:Semántico: Es método privado.
7.java:32:Semántico: No existe main().
Error en la compilación. Tue Sep 23 22:29:48 2001
```

3.2.4 Ejemplo 8

Objetivos

- Comprobar las sentencias.

```
public class Clase1 {
    public int i;
    public boolean vulean;
    private int j;

    public int Imprime_Public(int j)
    {
        print ("hola");
        print(3);
    }

    private boolean Imprime_Private(int j)
    {
        int otro_entero;
        int var;

        var = (Imprime_Public()) + 1 ;
        read();

        return true;
    }

    public boolean Nada()
    {
        int i = 0;
        Imprime_Private(3);
    }
}

public class Principal {
    public void main()
    {
        Clase1 c1(3, 2), c2 (3, 5);
        const int constante_entera = 3;
        const boolean constante_vuleana = true;
        int h1;
        int h2, h3, h4;

        boolean h3, h1;

        h1 = 1;
        h2 = 2;

        h1 = 1 + 3;
        c1.Nada(3);
    }
```

```
}
```

Resultados

```
../jcc 8.java
8.java:44:Semántico: Redefinición no permitida.
8.java:44:Semántico: Redefinición no permitida.
8.java:50:Semántico: Violación de tipos.
Error en la compilación. Tue Sep 23 22:30:23 2001
```