



## **PROCESADORES DE LENGUAJE**

**Ingeniería Informática  
Especialidad de computación  
Tercer curso, segundo cuatrimestre**



Departamento de Informática y Análisis Numérico  
Escuela Politécnica Superior de Córdoba  
Universidad de Córdoba

---

# **GUION DE LAS CLASES DE PRÁCTICAS**

**BISON/YACC y FLEX/LEX**

## Contenido

Ejemplo 1. Reconocimiento de expresiones aritméticas simples.....	3
Ejemplo 2. Análisis de un fichero pasado como argumento .....	6
Ejemplo 3. Reconocimiento de operadores unarios.....	7
Ejemplo 4. Evaluación de expresiones aritméticas.....	9
Ejemplo 5. Separador de expresiones y nuevos operadores .....	10
Ejemplo 6. Recuperación de errores de ejecución .....	11
Ejemplo 7. Sentencia de asignación de variables .....	12
Ejemplo 8. Conflicto de desplazamiento-reducción .....	14
Ejemplo 9. Solución del conflicto de desplazamiento - reducción y uso de sentencias de lectura y escritura .....	15
Ejemplo 10. Constantes predefinidas que se pueden modificar .....	16
Ejemplo 11.- Constantes numéricas predefinidas que no se pueden modificar. ....	17
Ejemplo 12. Palabras claves pre-instaladas en la tabla de símbolos .....	19
Ejemplo 13. Funciones matemáticas predefinidas con un argumento.....	20
Ejemplo 14.- Funciones predefinidas con cero o dos argumentos.....	22
Ejemplo 15. Uso de AST para la generación de código intermedio .....	24
Ejemplo 16.- Constantes y variables lógicas, operadores relacionales y lógicos.....	27
Ejemplo 17.- Sentencias de control de flujo y conflicto del “else danzante” .....	31

## Ejemplo 1. Reconocimiento de expresiones aritméticas simples

- **DESCRIPCIÓN**

- Comprueba si las expresiones aritméticas son léxica y sintácticamente correctas.
- Las expresiones aritméticas están compuestas solamente por números.
- Se permite la suma, resta, multiplicación y división.
- También permite expresiones entre paréntesis.
- Las expresiones deben terminar con un salto de línea.
- Muestra un mensaje cuando se detecta un error
  - Comando incluido en el fichero ***interpreter.y*** para mostrar más información de un error  
***%error-verbose***

- **OBSERVACIÓN**

- No permite operadores unarios:
  - signo "+" unario  
✓ +2
  - signo "-" unario  
✓ -2

- **FICHEROS Y SUBDIRECTORIOS**

- ***interpreter.cpp***: programa principal
- ***makefile***: fichero para la compilación del intérprete
- ***Doxyfile***: fichero de configuración de doxygen
- ***Subdirectorio parser***
  - ***interpreter.y***: fichero de yacc con la gramática del analizador sintáctico
  - ***interpreter.l***: fichero de lex con las expresiones regulares del analizador léxico
  - ***makefile***: fichero de compilación del subdirectorio parser
- ***Subdirectorio error***
  - ***error.hpp***: prototipos de las funciones de recuperación de error
  - ***error.cpp***: código de las funciones de recuperación de error
  - ***makefile***: fichero de compilación del subdirectorio error
- ***Subdirectorio includes***
  - ***macros.hpp***: macros de pantalla
- ***Subdirectorio examples***
  - ***test.txt***: fichero de ejemplo sin errores
  - ***test-error.txt***: fichero de ejemplo con errores

- **FUNCIONAMIENTO DEL INTÉRPRETE**

- **Interactivo**  
*\$/interpreter.exe*

```
stmtlist --> epsilon
2+3
exp --> NUMBER
exp --> NUMBER
exp --> exp '+' exp
stmtlist --> stmtlist exp '\n'
Correct expression
```

- El programa finaliza
  - ✓ pulsando *Control + D*
  - ✓ o tecleando el carácter *#* al principio de línea

```
#
>>>>>> End of file <<<<<<<
program --> stmtlist
```

- **Redirigiendo un fichero de entrada**

```
$ ./interpreter.exe < ./examples/test.txt
stmtlist --> epsilon
exp --> NUMBER
stmtlist --> stmtlist exp '\n'
Correct expression
```

...

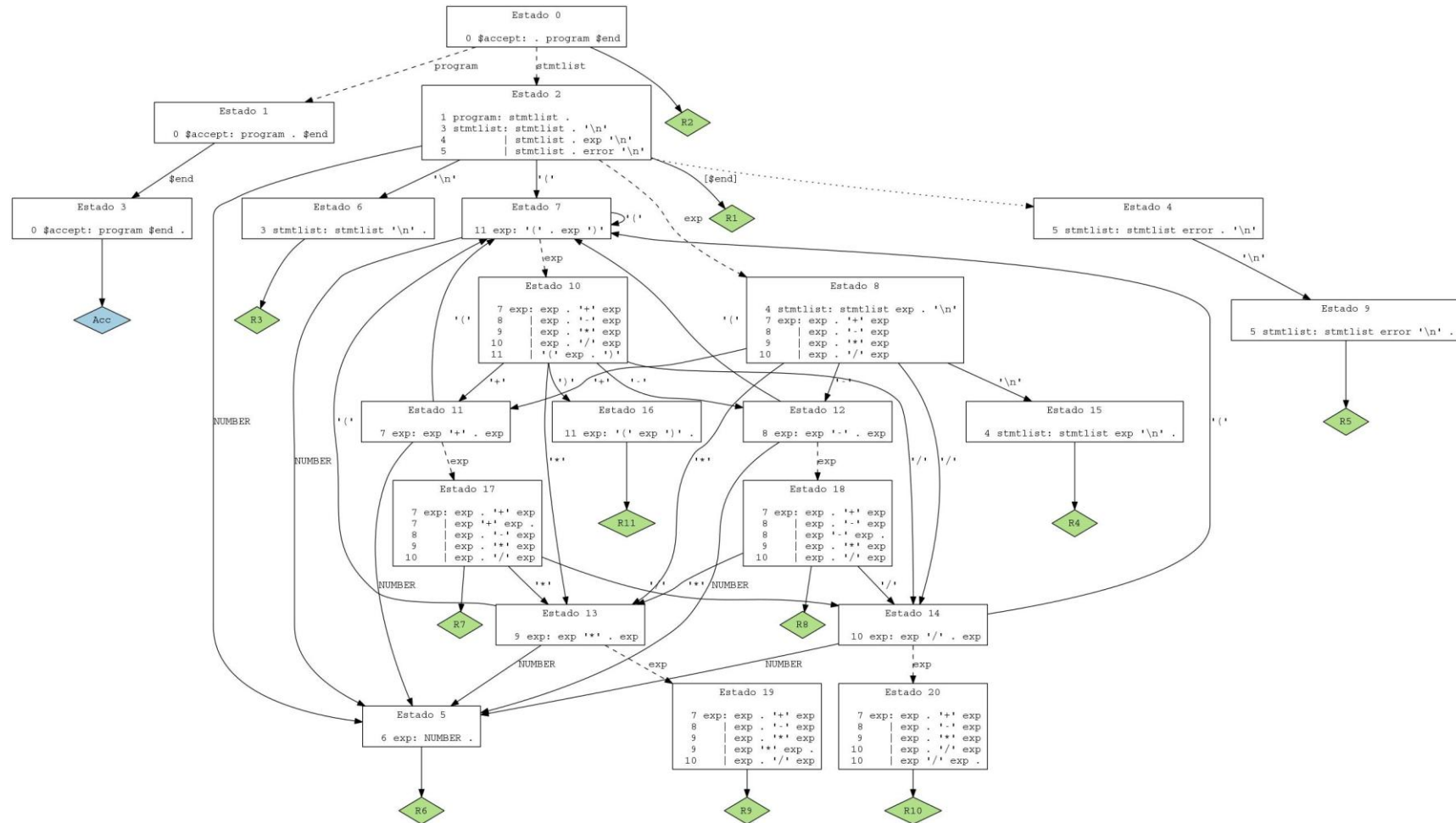
```
>>>>>> End of file <<<<<<<
program --> stmtlist
```

- El programa finaliza cuando
  - ✓ lee el carácter de fin de fichero
  - ✓ o lee el carácter *#* al principio de una línea del fichero

- **GENERACIÓN DEL AUTÓMATA FINITO DETERMINISTA QUE RECONOCE LOS PREFIJOS VIABLES**

- En el directorio *parser*
- Representación gráfica
  - *dot -TXXX interpreter.dot -o interpreter.XXX*
  - donde XXX puede ser:
    - ✓ svg, png, jpg, gif, pdf, ps, fig, ...
  - Ejemplo
    - ✓ *dot -Tjpg interpreter.dot -o interpreter.jpg*
- Fichero de texto y tabla LALR
  - *bison -v interterper.y*
  - Se genera el fichero *interpreter.output*

## AUTÓMATA FINITO DETERMINISTA QUE RECONOCE LOS PREFIJOS VIABLES



## Ejemplo 2. Análisis de un fichero pasado como argumento

- **NOVEDADES**

- Se analiza un fichero de entrada pasado como argumento desde la línea de comandos

```
$ ./interpreter.exe ./examples/test.txt
```

*...*

- Se muestra el nombre del programa en los mensajes de error:

```
$ ./interpreter.exe
```

```
afd
```

```
Program: ./interpreter.exe
```

```
Error line 1 --> Parsing error
```

```
syntax error, unexpected $undefined, expecting $end or NUMBER or '\n' or '('
```

- **FICHEROS MODIFICADOS**

- interpreter.cpp

```
int main(int argc, char *argv[])
```

*...*

```
if (argc == 2)
```

```
    yyin = fopen(argv[1], "r");
```

- error.cpp

```
void warning(std::string errorMessage1, std::string errorMessage2)
```

```
{
```

```
    /*****
```

```
    /* NEW in example 2 */
```

```
    std::cerr << IGREEN;
```

```
    std::cerr << " Program: " << progname << std::endl;
```

*...*

## Ejemplo 3. Reconocimiento de operadores unarios

- **NOVEDADES**

- Se permiten operadores unarios:
  - signo "+" unario  
✓ + 2
  - signo "-" unario  
✓ -2
- Observación:
  - permite expresiones como  
✓ ++ 2  
✓ + - + 3
  - Curiosidad: el lenguaje C también lo permite.
- Se utilizan identificadores para los componentes léxicos o tokens:
  - PLUS
  - MINUS
  - MULTIPLICATION
  - DIVISION
  - UNARY

```
$ ./interpreter.exe examples/test.txt
```

```
stmtlist --> epsilon
```

```
exp --> NUMBER
```

```
exp --> MINUS exp
```

```
stmtlist --> stmtlist exp NEWLINE
```

```
Correct expression
```

```
stmtlist --> stmtlist NEWLINE
```

```
exp --> NUMBER
```

```
exp --> PLUS exp
```

```
stmtlist --> stmtlist exp NEWLINE
```

```
Correct expression
```

```
stmtlist --> stmtlist NEWLINE
```

```
stmtlist --> stmtlist NEWLINE
```

```
exp --> NUMBER
```

```
exp --> PLUS exp
```

```
exp --> NUMBER
```

```
exp --> MINUS exp
```

```
exp --> NUMBER
```

```
exp --> exp PLUS exp
```

```
exp --> LPAREN exp RPAREN
```

```
exp --> exp MULTIPLICATION exp
```

```
stmtlist --> stmtlist exp NEWLINE
```

```
Correct expression
```

```
>>>>>> End of file <<<<<<<
```

```
program --> stmtlist
```

- Se muestran los errores léxicos.

```
$ ./interpreter.exe examples/test-error.txt
```

```
stmtlist --> epsilon
```

```
exp --> NUMBER
```

```
Program: ./interpreter.exe
```

```
Error line 1 --> Parsing error
```

```
    syntax error, unexpected MULTIPLICATION, expecting NUMBER
or PLUS or MINUS or LPAREN
```

```
stmtlist --> stmtlist error NEWLINE
```

```
stmtlist --> stmtlist NEWLINE
```

```
exp --> NUMBER
```

```
exp --> NUMBER
```

```
exp --> exp PLUS exp
```

```
stmtlist --> stmtlist exp NEWLINE
```

```
Correct expression
```

```
stmtlist --> stmtlist NEWLINE
```

```
Program: ./interpreter.exe
```

```
Error line 5 --> Lexical error
```

```
dato
```

```
stmtlist --> stmtlist NEWLINE
```

```
stmtlist --> stmtlist NEWLINE
```

```
exp --> NUMBER
```

```
exp --> NUMBER
```

```
exp --> exp MULTIPLICATION exp
```

```
Program: ./interpreter.exe
```

```
Error line 7 --> Parsing error
```

```
    syntax error, unexpected RPAREN
```

```
stmtlist --> stmtlist error NEWLINE
```

```
>>>>>> End of file <<<<<<<
```

```
program --> stmtlist
```

- **FICHEROS MODIFICADOS**

- interpreter.l
  - En particular, se usa el estado de flex ERROR para controlar componentes léxicos no reconocidos (todavía), como los identificadores, etc.
- interpreter.y
  - Reglas para los operadores unarios.

- **EJERCICIO**

- Cambiar operadores aritméticos
  - + --> &
  - \* --> #



## Ejemplo 4. Evaluación de expresiones aritméticas

- **NOVEDADES**

- Se evalúan las expresiones aritméticas compuestas por números y se muestra el resultado

```
./interpreter.exe
```

```
2+3
```

```
Result: 5
```

```
5*4
```

```
Result: 20
```

- **FICHEROS MODIFICADOS**

- interpreter.l

```
{NUMBER1}|{NUMBER2} {
```

```
/* MODIFIED in example 4 */
```

```
/* Conversion of type and sending of the numerical
```

```
value to the parser */
```

```
yyval.number = atof(yytext);
```

```
return NUMBER;
```

```
}
```

- interpreter.y

- Tipo de dato de los valores de las expresiones

```
/* Data type YYSTYPE */
```

```
/* NEW in example 4 */
```

```
%union
```

```
{
```

```
double number;
```

```
}
```

```
/* Data type of the non-terminal symbol "exp" */
```

```
%type <number> exp
```

- **SIGNIFICADO DE NUEVOS TÉRMINOS**

- **yyval**

- atributo de un componente léxico.

- **YYSTYPE**

- tipo de dato del atributo

- Véase el fichero **interpreter.tab.h**

- **\$\$**

- atributo del símbolo no terminal de la parte izquierda de la regla

- **\$1**

- atributo del primer símbolo de la parte derecha de la regla.

- **\$2**

- atributo del segundo símbolo de la parte derecha de la regla.

- **\$n**

- atributo del símbolo n-ésimo de la parte derecha de la regla.

## Ejemplo 5. Separador de expresiones y nuevos operadores

- **NOVEDADES**

- Se utiliza el símbolo “;” para separar expresiones

*2+3; 4\*5;*

*Result: 5*

*Result: 20*

- Nuevos operadores

- Resto de la división entera:

*8%3;*

*Result: 2*

- Potencia (asociativa por la derecha)

*2^3;*

*Result: 8*

*2^3^2;*

*Result: 512*

*(2^3)^2;*

*Result: 64*

- **OBSERVACIÓN**

- No se controla la división por cero del operador de división (/) ni del resto de la división entera (%)

- Se controlará en el ejemplo 6

- **FICHEROS MODIFICADOS**

- interpreter.y

- Definición de los componentes léxicos: *MODULO*, *POWER*

- Asociatividad por la derecha

✓ *%right POWER*

- Regla para la división entera

✓ conversión de tipo con (int)

- Regla para la potencia

✓ Uso de la función pow de math.h

- interpreter.l

- Token MODULO → %

- Token POWER → ^

- Token SEMICOLON → ;

- Al reconocer “\n”,

✓ no se devuelve NEWLINE,

✓ pero se incrementa el contador de líneas.

- test.txt

- test-error.txt

## Ejemplo 6. Recuperación de errores de ejecución

- **NOVEDADES**

- Se ha incluido un mecanismo para recuperarse de un error de ejecución:
  - Si hay un error de ejecución, el intérprete lo comunica pero no termina la ejecución.
  - Se controla la división por cero de los operadores de división y de resto de la división entera.

*8%0;*

*Program: ./interpreter.exe*

*Error line 1 --> Runtime error in modulo*

*Division by zero*

*3/0;*

*Program: ./interpreter.exe*

*Error line 2 --> Runtime error in division*

*Division by zero*

- **FICHEROS MODIFICADOS**

- interpreter.cpp
  - // Use for recovery of runtime errors*
  - #include <setjmp.h>*
  - #include <signal.h>*
  - ...*
  - /\* Sets a viable state to continue after a runtime error \*/*
  - setjmp(begin);*
  - /\* The name of the function to handle floating-point errors is set \*/*
  - signal(SIGFPE,fpecatch);*
- interpreter.y
  - #include <setjmp.h>*
  - #include <signal.h>*
  - ...*
  - jmp\_buf begin; //!< It enables recovery of runtime errors*
  - ....*
- error.hpp
  - execerror
  - fpecatch
- error.cpp
  - execerror
  - fpecatch

## Ejemplo 7. Sentencia de asignación de variables

- **NOVEDADES**

- Permite la creación de variables de tipo real y su uso en expresiones aritméticas

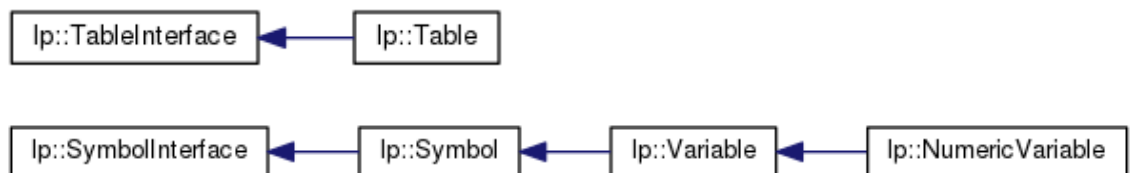
```
a = 2;  
Result: 2  
a;  
Result: 2  
Usa  
dato = 3 * a;  
dato = 2 * a;  
Result: 4  
dato;  
Result: 4
```

- Permite la asignación múltiple en una misma sentencia

```
a = b = c = 7;  
Result: 7  
a; b; c;  
Result: 7  
Result: 7  
Result: 7
```

- Nuevos tipos abstractos de datos

- TableInterface
- Table
- SymbolInterface
- Symbol
- Variable
- NumericVariable



- **OBSERVACIONES**

- Una variable es un identificador que empieza por una letra y que pueda ir seguida de letras o dígitos.
- Los identificadores se almacenan en una tabla de símbolos (map de STL) de variables numéricas o indefinidas.
- Se controlan las variables no definidas.

```
2+iva;
```

```
Program: ./interpreter.exe
```

```
Error line 1 --> The variable is UNDEFINED
```

```
iva
```

- **FICHEROS MODIFICADOS**

- interpreter.y
- interpreter.l
- makefile principal
- makefile del subdirectorio table
- test.txt
- test-error.txt
- Doxyfile

*#Modified in example 7*

*INPUT = interpreter.cpp parser error includes **table***

- **FICHEROS NUEVOS**

- tableInterface.hpp
  - Definición de la clase abstracta TableInterface
- table.hpp
  - Definición de la clase Table
- table.cpp
  - Código del resto de funciones de la clase Table
- symbolInterface.hpp:
  - Definición de la clase abstracta SymbolInterface
- symbol.hpp
  - Definición de la clase Symbol
- symbol.cpp
  - Código del resto de funciones de la clase Symbol
- variable.hpp
  - Definición de la clase Variable, que hereda de Symbol
- variable.cpp
  - Código del resto de funciones de la clase Variable
- numericVariable.hpp
  - Definición de la clase NumericVariable, que hereda de Variable
- numericVariable.cpp
  - Código del resto de funciones de la clase NumericVariable

- **EJERCICIO**

- Modifica el operador de asignación para que se pueda utilizar el operador de asignación de Pascal
  - *dato := 3;*

## Ejemplo 8. Conflicto de desplazamiento-reducción

- **NOVEDADES**

- Ejemplo de diseño de una gramática que genera un conflicto de desplazamiento reducción

*\$ make*

*Accessing directory parser*

*make[1]: se entra en el directorio*

*'... /ejemplo8/parser'*

*Generando: interpreter.tab.c interpreter.tab.h*

*interpreter.y: aviso: 1 **conflicto desplazamiento/reducción** [-Wconflicts-sr]*

- El conflicto será corregido en el ejemplo 9
- La sentencia de asignación se puede generar de dos maneras:

- Primera derivación

✓ *program* → *stmtlist*

→ *stmtlist asgn SEMICOLON*

→ *asgn SEMICOLON*

→ *VARIABLE ASSIGNMENT exp SEMICOLON*

→ *VARIABLE ASSIGNMENT NUMBER SEMICOLON*

- Segunda derivación

✓ *program* → *stmtlist*

→ *stmtlist exp SEMICOLON*

→ *exp SEMICOLON*

→ *asgn SEMICOLON*

→ *VARIABLE ASSIGNMENT exp SEMICOLON*

→ *VARIABLE ASSIGNMENT NUMBER SEMICOLON*

- **REVISIÓN DEL CONFLICTO**

- Se genera el fichero *interpreter.output* para obtener información del conflicto

*\$ bison -v interpreter.y*

- Fichero *interpreter.output*

...

*Estado 11 conflictos: 1 desplazamiento/reducción*

...

*Estado 11*

*4 stmtlist: stmtlist asgn . SEMICOLON*

*18 exp: asgn .*

*SEMICOLON **desplazar e ir al estado 19***

*SEMICOLON **[reduce usando la regla 18 (exp)]***

*\$default **reduce usando la regla 18 (exp)***

- **FICHERO NUEVO**

- *interpreter.output*

- Fichero que describe la tabla LALR y el conflicto de desplazamiento - reducción

- **FICHEROS MODIFICADOS**

- *interpreter.y*

- Uso del no terminal *asgn* y de su regla

## Ejemplo 9. Solución del conflicto de desplazamiento - reducción y uso de sentencias de lectura y escritura

- **NOVEDADES**

- Se resuelve el problema del conflicto de desplazamiento – reducción del ejemplo 8.
  - Modificación de las reglas de *asgn* y *exp*.
- El intérprete permite la lectura de variables y la escritura de los valores de las expresiones aritméticas
  - Sentencia *read*
  - Sentencia *print*

- **FICHEROS MODIFICADOS**

- interpreter.l
  - Las palabras reservadas son reconocidas mediante reglas específicas.

```
print      {return PRINT;}
read       {return READ;}
```

- interpreter.y
  - Nuevos tokens: PRINT, READ
  - Nuevos símbolos no terminales: print, read
  - Nuevas reglas

```
stmtlist:      /* empty: epsilon rule */
              | stmtlist stmt
              | stmtlist error
              ;

stmt:  SEMICOLON /* Empty statement: ";" */
      | asgn SEMICOLON
      | print SEMICOLON
      | read SEMICOLON
      ;

asgn:  VARIABLE ASSIGNMENT exp
      | VARIABLE ASSIGNMENT asgn
      ;

print: PRINT exp
...
read:  READ LPAREN VARIABLE RPAREN
```

- Se ha **eliminado** la regla

```
exp: ...
    | asgn
    ...
```

- **EJERCICIOS**

- Definición regular para no distinguir mayúsculas de minúsculas en *print*
  - *(?i:print)* {return PRINT;}
- No distinguir mayúsculas de minúsculas en los identificadores

```
for (int i = 0; yytext[i] != '\0'; i++)
{
    yytext[i] = toupper(yytext[i]);
}
```

## Ejemplo 10. Constantes predefinidas que se pueden modificar

- **NOVEDADES**

- El intérprete permite el uso de constantes predefinidas
  - "PI", 3.14159265358979323846
    - ✓  $\pi = \frac{\text{circunferencia}}{\text{diámetro}}$
  - "E", 2.71828182845904523536
    - ✓ Base natural
    - ✓  $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$
  - "GAMMA", 0.57721566490153286060
    - ✓ Constante de Euler-Mascheroni
    - ✓  $\gamma = \lim_{n \rightarrow \infty} \left( -\ln(n) + \sum_{k=1}^n \frac{1}{k} \right)$
  - "DEG", 57.29577951308232087680
    - ✓ Grado por radián
    - ✓  $180/\pi$
  - "PHI", 1.61803398874989484820
    - ✓ Proporción áurea
    - ✓  $\varphi = \frac{\sqrt{5}+1}{2}$

- **OBSERVACIÓN**

- Se puede cambiar el valor de una constante predefinida:
  - $PI = 3.14$
  - $PI = 0.0$
- El ejemplo 11 evitará modificar las constantes predefinidas.

- **FICHEROS NUEVOS**

- init.hpp:
  - Definición de las constantes predefinidas
  - Prototipo de la función *init*
- init.cpp:
  - Código de la función *init* que inicializa la tabla de símbolos con las constantes predefinidas.

- **FICHEROS MODIFICADOS**

- interpreter.cpp
  - Llamada a *init(table)*, que inicializa la tabla de símbolos con las constantes predefinidas.
- interpreter.y
  - Inclusión del fichero de cabecera
    - $\#include \text{ "../table/init.hpp" }$
- makefile del subdirectorio table
  - Compilación de los ficheros init.cpp e init.hpp



## Ejemplo 11.- Constantes numéricas predefinidas que no se pueden modificar.

- **NOVEDADES**

- Las constantes predefinidas no se puede modificar en las sentencias de asignación o lectura.

*PI = 7;*

*Program: ./interpreter.exe*

*Error line 3 --> Semantic error in assignment: it is not allowed to modify a constant*

*PI*

*read(PI);*

*Program: ./interpreter.exe*

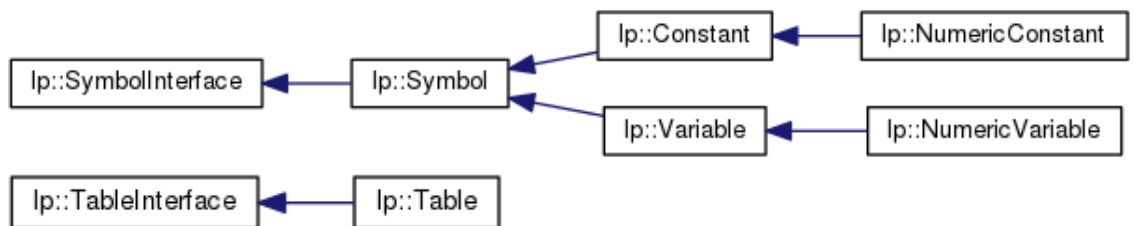
*Error line 4 --> Semantic error in "read statement": it is not allowed to modify a constant*

*PI*

- Nuevo tipo de dato:
  - Contante numérica
- Se utilizan reglas gramaticales para controlar los errores.

- **FICHEROS NUEVOS**

- constant.hpp:
  - Definición de la clase *Constant*, que hereda de *Symbol*.
- constant.cpp:
  - Código del resto de funciones de la clase *Constant*.
- numericConstant.hpp
  - Definición de la clase *NumericConstant*, que hereda de *Constant*.
- numericConstant.cpp
  - Código del resto de funciones de la clase *NumericConstant*.



- **FICHEROS MODIFICADOS**

- interpreter.l
  - Si un identificador está instalado en la tabla de símbolos, se devuelve su token o componente léxico: VARIABLE o CONSTANT
- interpreter.y
  - Reglas gramaticales de control de errores

```
asgn: ...  
      / CONSTANT ASSIGNMENT exp  
      / CONSTANT ASSIGNMENT asgn  
...  
read: ...
```

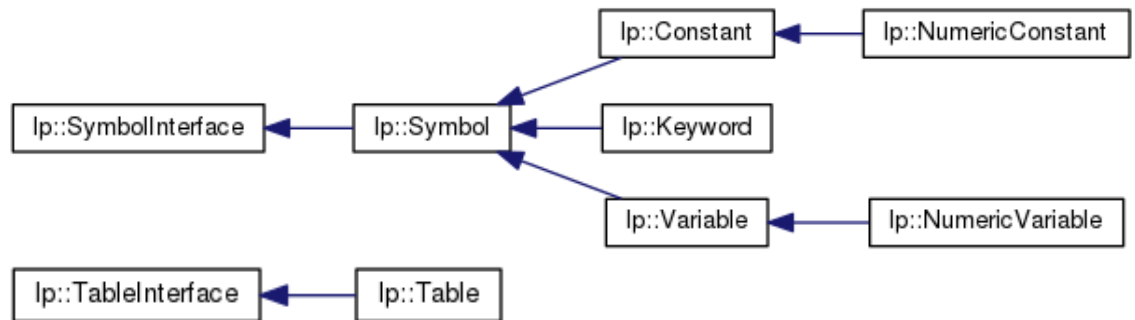
*/ READ LPAREN CONSTANT RPAREN*

- init.cpp
  - Se instalan las constantes predefinidas en la tabla de símbolos
    - ✓ con el token CONSTANT
    - ✓ y el tipo NUMBER
- makefile del subdirectorio table
  - Compilación de los nuevos ficheros
    - ✓ constant.hpp
    - ✓ constant.cpp
    - ✓ numericConstant.hpp
    - ✓ numericConstant.cpp

## Ejemplo 12. Palabras claves pre-instaladas en la tabla de símbolos

- **NOVEDADES**

- Las palabras claves son preinstaladas en la tabla de símbolos
- Nuevo tipo de dato:
  - Keyword: Palabra clave



- **FICHEROS NUEVOS**

- keyword.hpp
  - Definición de la clase *Keyword*, que hereda de *Symbol*
- keyword.cpp
  - Código del resto de funciones de la clase *Keyword*

- **FICHEROS MODIFICADOS**

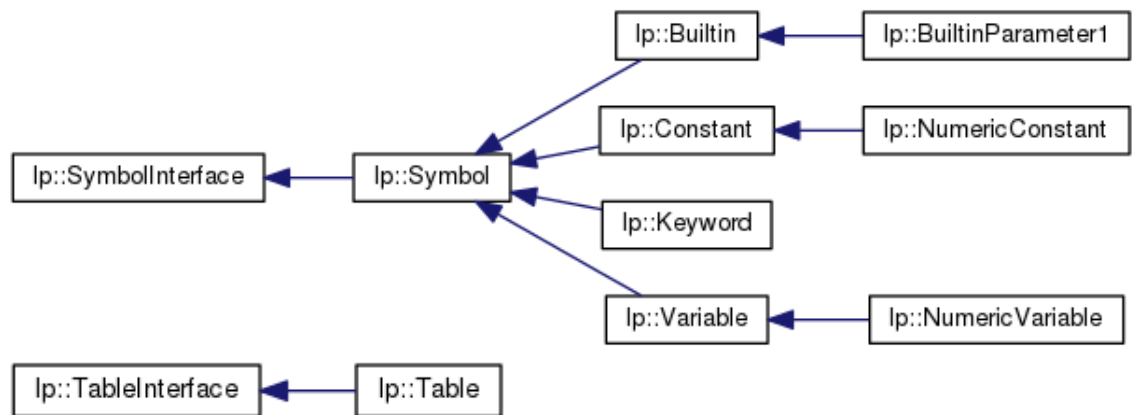
- interpreter.l:
  - Las reglas de PRINT y READ se han eliminado porque son preinstaladas en la tabla de símbolos
- init.hpp
  - Definición de las palabras claves
- init.cpp
  - Instalación de las palabras claves en la tabla de símbolos.
- makefile del subdirectorio table
  - Compilación de los nuevos ficheros
    - ✓ keyword.hpp
    - ✓ keyword.cpp

## Ejemplo 13. Funciones matemáticas predefinidas con un argumento

- **NOVEDADES**

- El intérprete permite el uso de funciones matemáticas predefinidas con un argumento
  - `sin`, `cos`, `atan`, `log`, `log10`, `exp`, `sqrt`, `int`, `abs`
  - Ejemplo

```
print sin(PI/2);
Print: 1
```
- Nuevos tipos abstractos de datos:
  - Función predefinida: *Builtin*
  - Función predefinida con un argumento: *BuiltinParameter1*
- Función *errcheck* para comprobar si una función matemática genera algún error en su dominio o rango



- **FICHEROS NUEVOS**

- `mathFunction.hpp`
  - Prototipo de las funciones matemáticas predefinidas
- `mathFunction.cpp`
  - Código de las funciones matemáticas predefinidas
- `builtin.hpp`:
  - Definición de la clase *Builtin*
- `builtin.cpp`:
  - Código de la clase *Builtin*
- `builtinParameter1.hpp`:
  - Definición de la clase *BuiltinParameter1*
- `builtinParameter1.cpp`
  - Código de la clase *BuiltinParameter1*

- **FICHEROS MODIFICADOS**

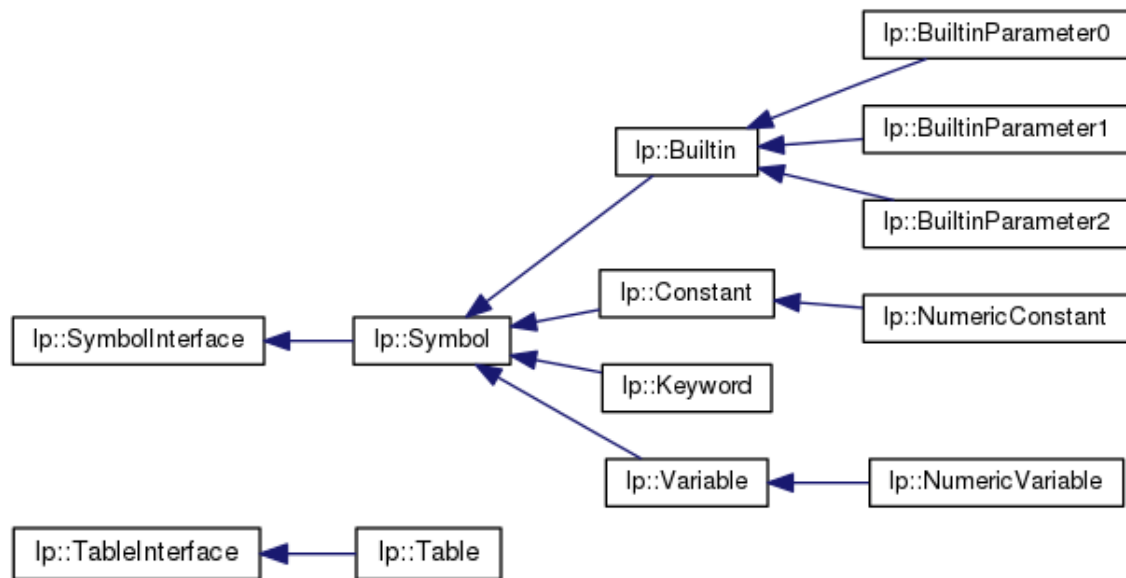
- `interpreter.y`
  - Regla para el uso de funciones matemáticas con un argumento
- `error.hpp`:
  - Prototipo de la nueva función *errcheck*
- `error.cpp`

- Código de la nueva función *errcheck*
  - init.hpp
    - Definición de las funciones matemáticas predefinidas y con un argumento
  - init.cpp
    - Instalación de las funciones matemáticas predefinidas y con un argumento
  - makefile el subdirectorio table
    - Compilación de los nuevos ficheros
      - ✓ mathFunction.hpp
      - ✓ mathFunction.cpp
      - ✓ builtin.hpp
      - ✓ builtin.cpp
      - ✓ builtinParameter1.hpp
      - ✓ builtinParameter1.cpp
  - test.txt
    - Nuevos ejemplos
  - test-error.txt
    - Nuevos ejemplos con errores
- **EJERCICIO**
    - Poner la funciones predefinidas en castellano:
      - seno, coseno, atan, log, log10, exp, raiz, entero, abs.

## Ejemplo 14.- Funciones predefinidas con cero o dos argumentos

- **NOVEDADES**

- El intérprete permite el uso de funciones matemáticas predefinidas con cero, o dos argumentos
  - Función predefinida con cero argumentos
    - ✓ `random()`
  - Función predefinida con dos argumentos:
    - ✓ `atan2(x,y)`
- Nuevos tipos abstractos de datos:
  - Función predefinida con cero argumentos: *BuiltinParameter0*
  - Función predefinida con dos argumentos: *BuiltinParameter2*



- **FICHEROS NUEVOS**

- builtinParameter0.hpp:
  - Definición de la clase *BuiltinParameter0*
- builtinParameter0.cpp
  - Código de la clase *BuiltinParameter0*
- builtinParameter2.hpp:
  - Definición de la clase *BuiltinParameter2*
- builtinParameter2.cpp
  - Código de la clase *BuiltinParameter2*

- **FICHEROS MODIFICADOS**

- Interpreter.cpp
  - `#include <list>`
- mathFunction.hpp
  - Prototipo de las funciones matemáticas predefinidas con cero o dos argumentos
- mathFunction.cpp
  - Código de las funciones matemáticas predefinidas con cero o dos argumentos
- Interpreter.l

- Reconocimiento de COMMA
  - interpreter.y
    - Actualización de YYSTYPE
 

```
%union {
    double number;
    char * identifier;
    std::list<double> *parameters;
}
```
    - Definición del token COMMA
 

```
%nonassoc COMMA
```
    - Tipo de nuevos símbolos no terminales
 

```
%type <parameters> listOfExp restOfListOfExp
```
    - Reglas para el uso de funciones matemáticas con cualquier número de argumentos o parámetros
 

```
exp:
    ...
    | BUILTIN LPAREN listOfExp RPAREN
    ;

listOfExp:
    /* Empty list of numeric expressions */
    | exp restOfListOfExp
    ;

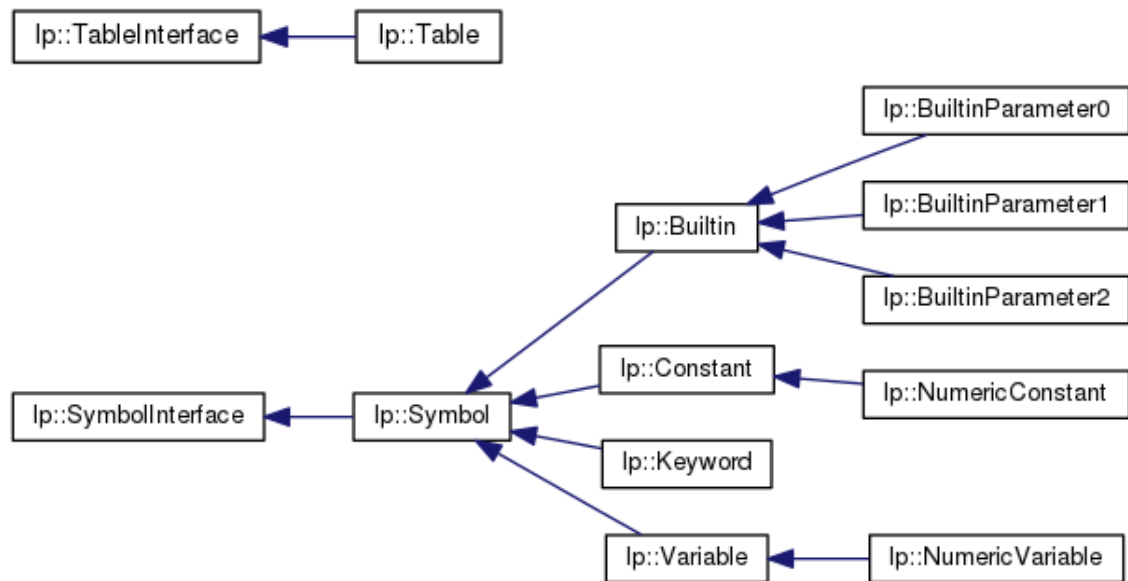
restOfListOfExp:
    /* Empty list of numeric expressions */
    | COMMA exp restOfListOfExp
    ;
```
  - init.hpp
    - Definición de las funciones matemáticas predefinidas y con cero o dos argumentos
  - init.cpp
    - #include <list>
    - Instalación de las funciones matemáticas predefinidas y con cero o dos argumentos
  - makefile del subdirectorio table
    - Compilación de los nuevos ficheros
      - ✓ builtinParameter0.hpp
      - ✓ builtinParameter0.cpp
      - ✓ builtinParameter2.hpp
      - ✓ builtinParameter2.cpp
  - test.txt
    - Nuevos ejemplos
  - test-error.txt
    - Nuevos ejemplos con errores

## Ejemplo 15. Uso de AST para la generación de código intermedio

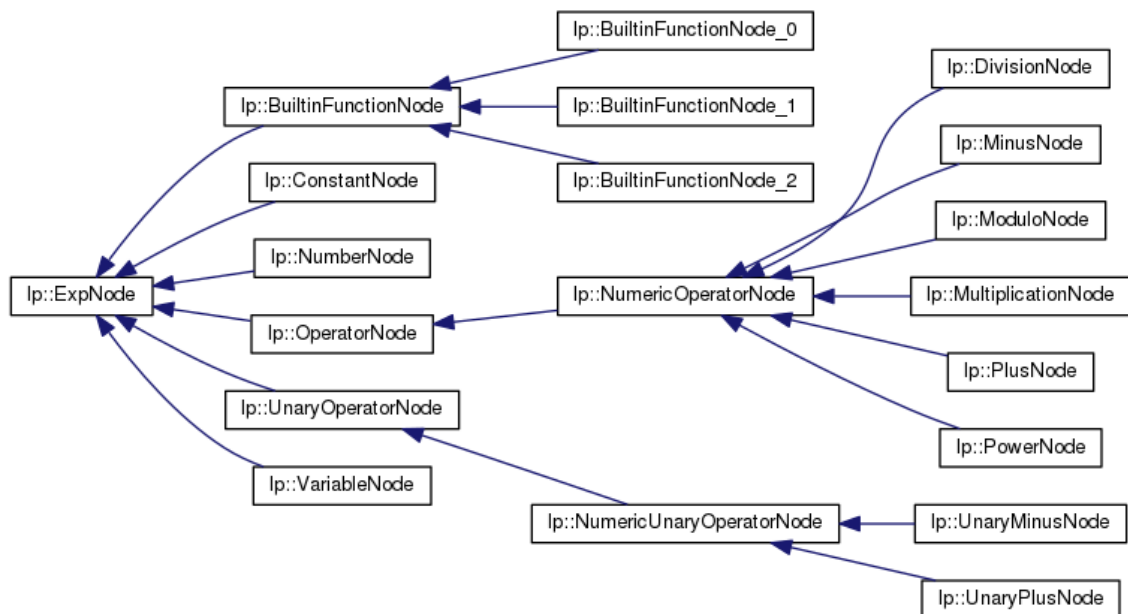
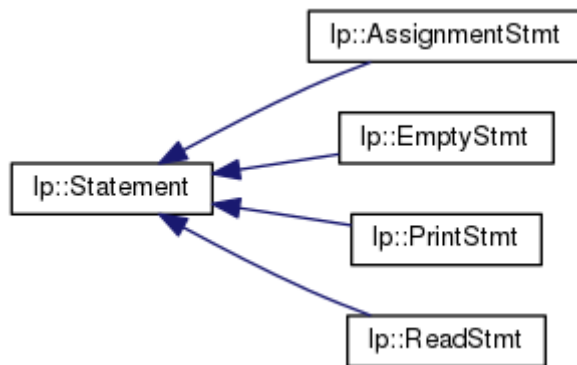
- **NOVEDADES**

- Uso de los árboles de sintaxis abstracta (AST) para generar código intermedio que es posteriormente evaluado.
  - Se controla el modo de ejecución del intérprete usando la variable *interactiveMode*
- Nuevos tipos abstractos de datos
  - *AST*
  - *Statement*
    - ✓ *AssignmentStmt*
    - ✓ *EmptyStmt*
    - ✓ *PrintStmt*
    - ✓ *ReadStmt*
  - *ExpNode*
    - ✓ *BuiltinFunctionNode*
      - *BuiltinFunctionNode\_1*
      - *BuiltinFunctionNode\_0*
      - *BuiltinFunctionNode\_2*
    - ✓ *NumberNode*
    - ✓ *NumericConstantNode*
    - ✓ *NumericVariableNode*
    - ✓ *OperatorNode*
      - *NumericOperatorNode*
        - *PlusNode*
        - *MinusNode*
        - *MultiplicationNode*
        - *DivisionNode*
        - *ModuloNode*
        - *PowerNode*
    - ✓ *UnaryOperatorNode*
      - *NumericUnaryOperatorNode*
        - *UnaryMinusNode*
        - *UnaryPlusNode*





Ip::AST



- **FICHEROS NUEVOS**

- ast.hpp
  - Definición de los tipos abstractos de datos de AST
- ast.cpp
  - Código de los tipos abstractos de datos de AST

- **FICHEROS MODIFICADOS**

- interpreter.cpp
  - Declaración de la variable de control de ejecución
    - ✓ bool interactiveMode;
  - Inclusión de “ast.hpp”
  - Declaración de la raíz del árbol de sintaxis abstracta
    - ✓ lp::AST \*root; *//!< Root of the abstract syntax tree AST*
  - Evaluación del AST
    - ✓ root->evaluate();
- interpreter.l
  - Inclusión de “ast.hpp”
- interpreter.y
  - Inclusión de “ast.hpp”
  - Referencia a la declaración de la variable de control de ejecución
    - ✓ extern bool interactiveMode;
  - Modificación de **YYSTYPE**

```

/* Data type YYSTYPE */
/* NEW in example 4 */
%union {
    double number;
    char * identifier;          /* NEW in example 7 */
    lp::ExpNode *expNode;      /* NEW in example 16 */
    std::list<lp::Statement *> *stmts; /* NEW in example 16 */
    lp::Statement *st;         /* NEW in example 16 */
    lp::AST *prog;             /* NEW in example 16 */
}
          
```
- init.hpp
  - Inclusión de “ast.hpp”
- Makefile
  - Compilación de los nuevos ficheros
    - ✓ ast.hpp
    - ✓ ast.cpp
- Doxyfile
  - #Modified in examples 7, 16
  - INPUT = interpreter.cpp parser error includes table ast
- text.txt
- text-error.txt

## Ejemplo 16.- Constantes y variables lógicas, operadores relacionales y lógicos

- **NOVEDADES**

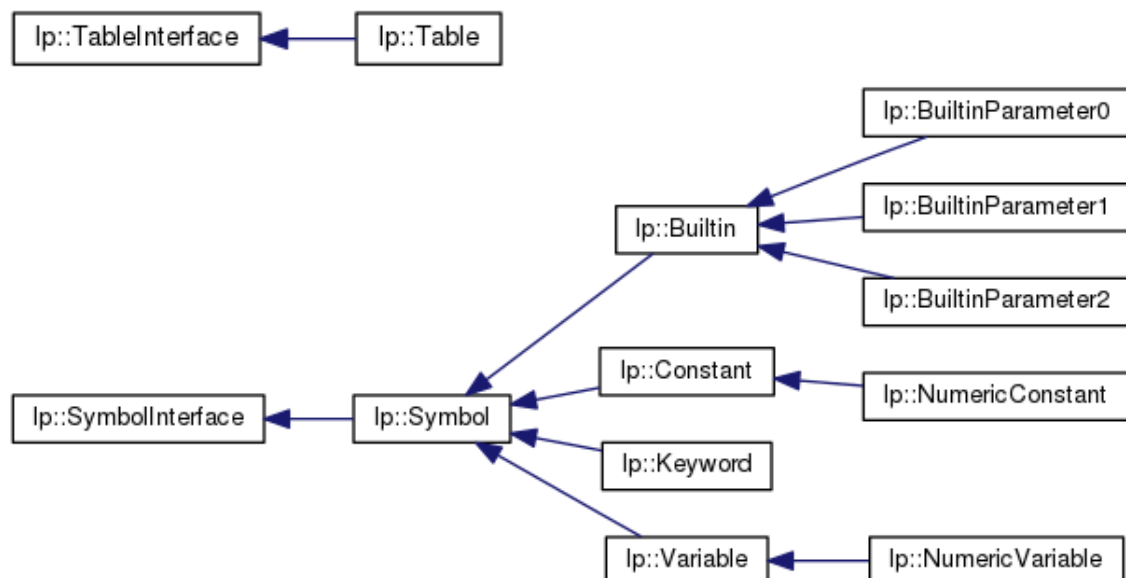
- Conversión dinámica del tipo de variable
  - Una variable puede ser de tipo numérico y luego de tipo lógico y viceversa

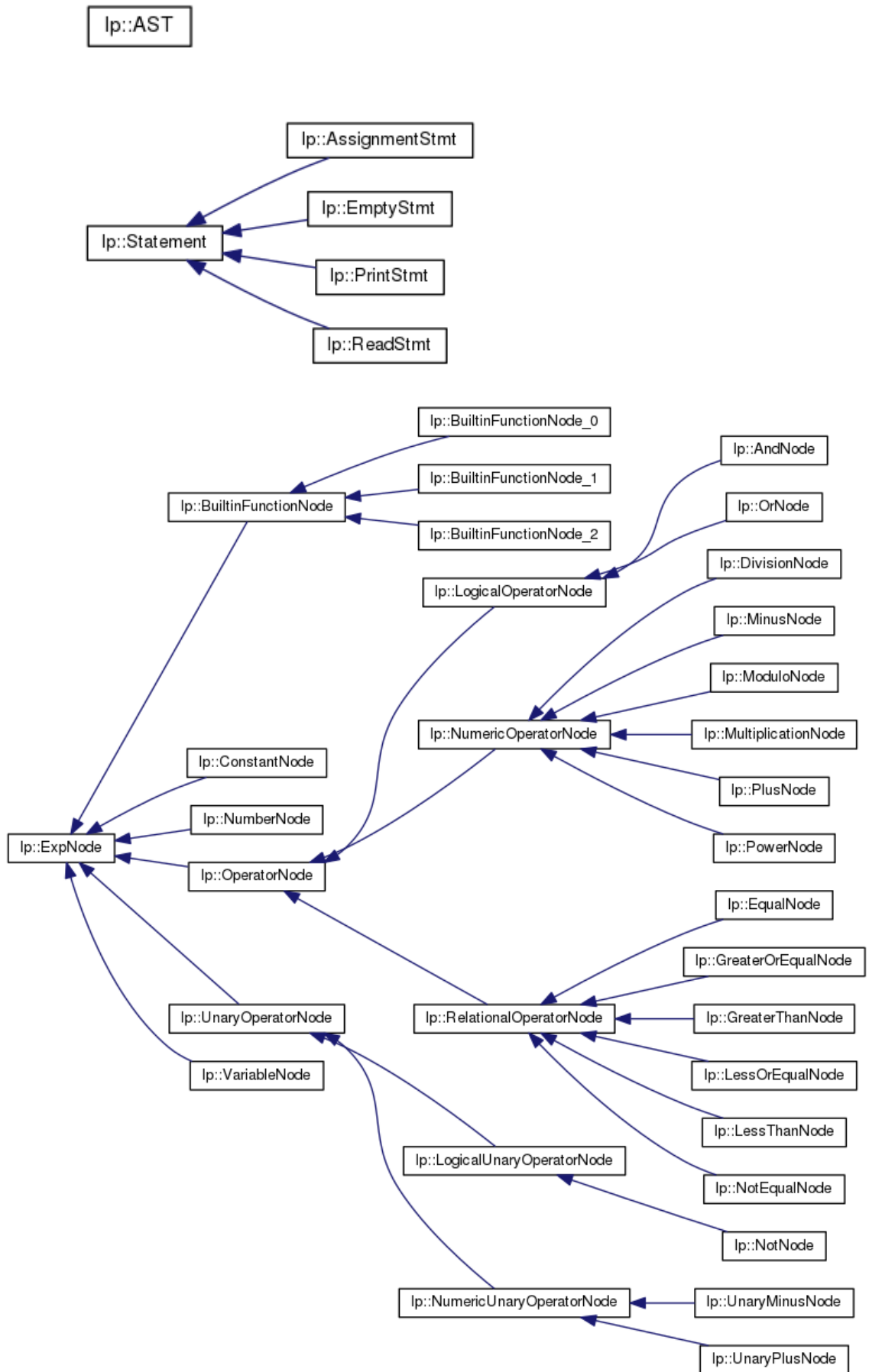
```
dato = 2;
print dato;
Print: 2

dato = 3>0;
print dato;
Print: true

dato = 3;
print dato;
Print: 3
```
- El intérprete permite el uso de
  - Constantes lógicas
    - ✓ true
    - ✓ false
  - Variables con valores lógicos
  - operadores relacionales
    - ✓ igualdad: ==
    - ✓ desigualdad: !=
    - ✓ menor que: <
    - ✓ menor o igual que: <=
    - ✓ mayor que: >
    - ✓ mayor o igual que: >=
  - operadores lógicos
    - ✓ conjunción lógica: &&
    - ✓ disyunción lógica: ||
    - ✓ negación lógica: not
- Nuevos tipos abstractos de datos
  - *AST*
  - *Statement*
    - ✓ *AssignmentStmt*
    - ✓ *EmptyStmt*
    - ✓ *PrintStmt*
    - ✓ *ReadStmt*
  - *ExpNode*
    - ✓ *BuiltinFunctionNode*
      - *BuiltinFunctionNode\_1*
      - *BuiltinFunctionNode\_0*
      - *BuiltinFunctionNode\_2*
    - ✓ *NumberNode*

- ✓ *NumericConstantNode*
- ✓ *NumericVariableNode*
- ✓ *OperatorNode*
  - *NumericOperatorNode*
    - *PlusNode*
    - *MinusNode*
    - *MultiplicationNode*
    - *DivisionNode*
    - *ModuloNode*
    - *PowerNode*
  - *RelationalOperatorNode*
    - *EqualNode*
    - *NotEqualNode*
    - *GreaterThanNode*
    - *GreaterOrEqualNode*
    - *LessThanNode*
    - *LessOrEqualNode*
  - *LogicalOperatorNode*
    - *AndNode*
    - *OrNode*
- ✓ *UnaryOperatorNode*
  - *NumericUnaryOperatorNode*
    - *UnaryMinusNode*
    - *UnaryPlusNode*
  - *LogicalUnaryOperatorNode*
    - *NotNode*





- **FICHEROS NUEVOS**

- table
  - *logicalConstant.hpp*
  - *logicalVariable.cpp*
  - *logicalVariable.hpp*
  - *logicalConstant.cpp*
- Nuevos ficheros de test
  - test2.txt, test3.txt
  - test-error2.txt

- **FICHEROS MODIFICADOS**

- interpreter.l
  - Reglas léxicas para reconocer los operadores relacionales y lógicos
- interpreter.y
  - Reglas sintácticas para reconocer las sentencias con operadores relacionales o lógicos.
- ast.hpp
  - Definición de los nuevos tipos abstractos de datos de AST
- ast.cpp
  - Código de los nuevos tipos abstractos de datos de AST
- test.txt, test2.txt, test3.txt
  - Nuevos ejemplos
- test-error.txt, test-error2.txt
  - Nuevos ejemplos con errores
- Makefile
  - Compilación de los nuevos ficheros.

- **EJERCICIO**

- Cambiar las constantes lógicas true y false por verdadero y falso.

## Ejemplo 17.- Sentencias de control de flujo y conflicto del “else danzante”

- **NOVEDADES**

- El intérprete permite el uso de sentencias de control de flujo
  - Sentencia condicional: if
  - Sentencia iterativa: while
- Se presenta un conflicto de desplazamiento – reducción provocado por la alternativa “else”
  - Fichero interpreter.output, generado con bison –v interpreter.y

*Estado 62 conflictos: 1 desplazamiento/reducción*

...

*Estado 62*

*13 if: IF cond stmt .*

*14 | IF cond stmt . ELSE stmt*

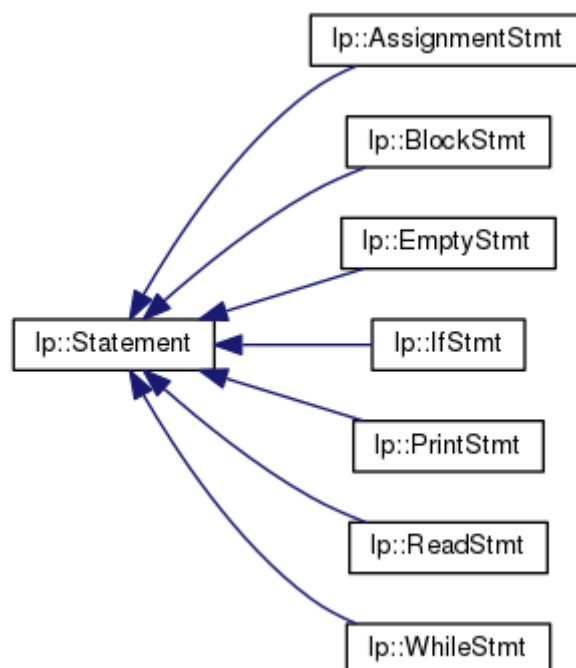
*ELSE desplazar e ir al estado 90*

*ELSE [reduce usando la regla 13 (if)]*

*\$default reduce usando la regla 13 (if)*

- Nuevos tipos abstractos de datos

- *BlockStmt*
- *IfStmt*
- *WhileStmt*



- **FICHEROS MODIFICADOS**

- interpreter.l
  - Reconocimiento de los caracteres “{” y “}”

```
"{" { return LETFCURLYBRACKET; } /* NEW in example 17 */
"}" { return RIGHTCURLYBRACKET; } /* NEW in example 17 */
```
- interpreter.y

- Tipo de los nuevos símbolos no terminales  
*/\* Type of the non-terminal symbols \*/*  
*// New in example 17: cond*  
*%type <expNode> exp cond*  
*%type <stmts> stmtlist*  
*// New in example 17: if, while, block*  
*%type <st> stmt asgn print read if while block*
- Símbolos no terminales y reglas gramaticales
  - ✓ *if*
  - ✓ *while*
  - ✓ *block*
  - ✓ *cond*
- `init.hpp`
  - Inclusión *if*, *else* y *while* en el grupo de palabras reservadas:  

```
static struct {
    std::string name ;
    int token;
} keywords[] = {
    "print", PRINT,
    "read", READ,
    "if",    IF,    // NEW in example 17
    "else",  ELSE,  // NEW in example 17
    "while", WHILE, // NEW in example 17
    "",      0
};
```
- `init.cpp`
  - Inclusión de "`ast.hpp`"  
*// NEW in example 17*  
*// This file must be before interpreter.tab.h*  
*#include "../ast/ast.hpp"*
- `ast.hpp`
  - Definición de las nuevas clases
    - ✓ *IfStmt*
    - ✓ *WhileStmt*
    - ✓ *BlockStmt*
- `ast.cpp`
  - Código de las nuevas clases
- `text.txt`
  - Nuevos ejemplos
- `text-error.txt`
  - Nuevos ejemplos con errores