uc3m

Procesadores del Lenguaje

Curso 2019-2020

Analizador lenguaje BSL, campus de Colmenarejo, curso 2019-20

1 Introducción

La práctica consistirá en la implementación de un compilador capaz de analizar programas deL lenguaje BSL (Basic Structs Language), que contiene expresiones aritmético-lógicas, estructuras de datos, funciones y estructuras de control sencillas, comprobando la corrección de la entrada en términos de los tipos de los datos utilizados.

Tras conocer los fundamentos del tratamiento léxico, sintáctico y semántico y la familiarización con las herramientas de construcción de compiladores, la práctica permitirá conocer la implementación completa de un analizador en las fases, léxica, sintáctica y semántica, y emplear estos conocimientos para reconocer errores en el uso de tipos e informar al programador.

2 Tareas a realizar

La práctica se dividirá en varias partes:

- Generación del analizador léxico (patrones léxicos) que reconozcan los elementos del lenguaje a este nivel.
- Generación de la gramática (LALR principalmente) que permitirá reconocer el lenguaje proporcionado y los símbolos terminales y no terminales de la misma mediante el ***parser*** creado a partir de ésta.
- Añadir la capacidad para registrar y tratar variables declaradas por el programador, almacenando el tipo con el que se declaren.
    - **Se deberá crear una estructura tabla de símbolos capaz de registrar la información de las variables del programa.**
    - **Se precisará además una tabla de registros para almacenar los tipos complejos declarados en el programa, de tipo STRUCT**
- Comprobaciones semánticas de todas las expresiones del programa, incluyendo la inicialización de variables, las expresiones asignadas a variables, operadores aplicados y condiciones de los controles de flujo.
- Opcionalmente, extender las tablas de variables y tipos para registrar las funciones declaradas y hacer comprobaciones semánticas en las llamadas a función.
- Opcionalmente, el compilador se recuperará de determinados errores (Léxicos, gramaticales, semánticos, etc.).

Nota: la máxima calificación se obtendrá si se implementa una de las dos partes opcionales indicadas

2.1 Especificación de la entrada

El lenguaje objetivo contiene operaciones aritmética y lógicas, así como estructuras de control de flujo, muy simplificadas[1]. Los tipos de datos que se incluyen son tipos básicos como números reales o booleanos, y tipos compuestos de estilo registro, declarados como **STRUCT**. Sobre los números se ejecutarán operaciones de suma, resta, multiplicación y división ("+", "-",

[1] Se remite al detalle de especificación de lenguaje disponible en la guía "Especificaciones del lenguaje de programación BSL"

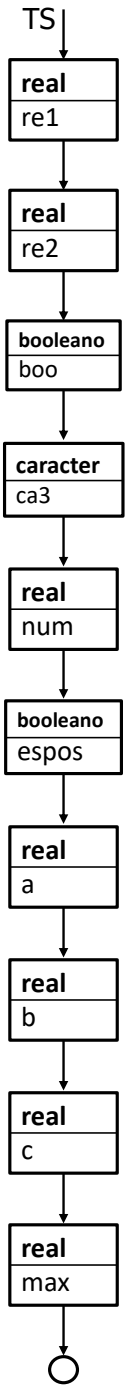
"*", "/"), tanto literales (ej. "5.13") como variables cuyo valor puede ser modificado a lo largo del programa. Análogamente, las operaciones booleanas serán la conjunción (**AND**), disyunción (**OR**) y negación (**NOT**). También existen operadores de comparación de valores numéricos de igual, menor y mayor ("==", "<=", ">="). El lenguaje permite dos tipos de sentencias: evaluación de expresiones matemáticas y asignaciones a variables. Las variables deberán haberse declarado con su tipo correspondiente antes de utilizarlas. Se recomienda leer cuidadosamente la especificación del lenguaje proporcionada para realizar la práctica

Se indican varios ejemplos de entrada junto con la representación intermedia del árbol de sintaxis abstracta y las comprobaciones que deben llevarse a cabo:

Ejemplo 1

```
<!-- Operaciones aritméticas y comparación
-----------------------------------------------
-----------------------------------------------
-->
REAL re1, re2;
BOOLEANO bo;
CARACTER ca := 'h'; # variable de tipo carácter, y su literal

re1 :=5 * +3 - 80/10; # esta expresion debe resultar 7
re2 :=10 / 5 * re1;     # esta expresion debe resultar 4
bo :=5<3; # comparación, debe resultar true


# si un número es positivo
REAL num;
BOOLEANO espos;
SI num >= 0 ENTONCES
   Espositivo := TRUE;
SINO
   Espositivo := FALSO;
FINSI

#cálculo del máximo
REAL a,b,c,max;
a :=1;
b :=2;
c :=3;
SI a>=b AND a>=c ENTONCES # a es el máximo
   max :=a;
SINO # el máximo es b o c
   SI b>=a AND b>=c ENTONCES # b es el máximo
         max :=b;
   SINO # c es el máximo, por descarte
         max :=c;
   FINSI
FINSI
```

En este ejemplo todas las variables son de tipos básicos predefinidos (real, entero, booleano, carácter). A continuación, se muestra la lista completa de símbolos (TS) almacenada con una estructura de tipo lista.



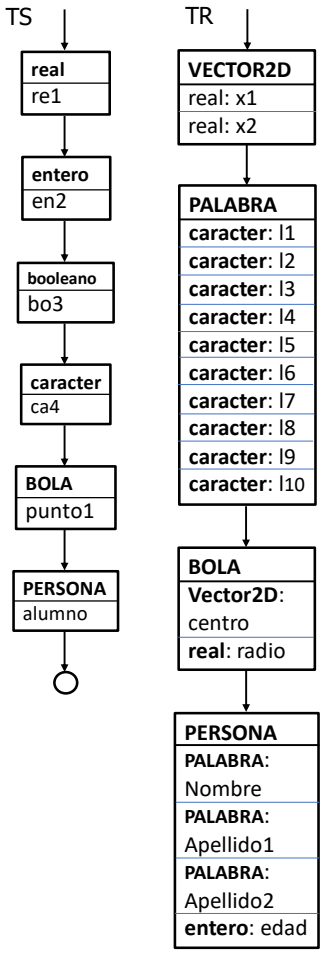
Ejemplo 2

Ejemplo con datos de tipo STRUCT:

```
<!-- Declaracion de variables y tipos STRUCT
-----------------------------------------------
-----------------------------------------------
-->

REAL re1;
ENTERO en2 := 6; # con asignación de valor
BOOLEANO bo3 := 5<3; # se asigna el resultado de una expresión
CARACTER ca4 := 'h'; # variable de tipo carácter, y su literal
STRUCT VECTOR2D {REAL x1; REAL x2};
STRUCT BOLA {VECTOR2D centro; REAL radio}; #definición tipo compuesto
BOLA punto1; # declaración variable struct (no permite asignación al
#               mismo tiempo que la declaracion
STRUCT PALABRA {CARACTER l1; CARACTER l2; CARACTER l3; CARACTER l4;
      CARACTER l5; CARACTER l6; CARACTER l7; CARACTER l8; CARACTER l9;
      CARACTER l10};
STRUCT PERSONA {PALABRA nombre; PALABRA apellido1; PALABRA apellido1;
      ENTERO edad};
PERSONA alumno;


### Expresiones
5 + 6; # operacion
re1 := 3.7; # asignación simple
bo3 := en2 < 7 AND 5.46+7*en2 > 4; # asignación compleja
punto1.centro.x1 := 0.0; # campo de variable struct anidada
punto1.centro.x2 := 0.0; # campo de variable struct anidada
punto1.radio := 10.0; # campo de variable struct
alumno.nombre.l1 :='A';
alumno.nombre.l2 : 'l';
alumno.nombre.l3 :='b'; alumno.apellido2.l1:=0;
alumno.nombre.l4 :='e';
alumno.nombre.l5 :='r';
alumno.nombre.l6 :='t';
alumno.nombre.l7 :='o';
alumno.nombre.l8 :=0;

alumno.apellido1.l1 :='G';
alumno.apellido1.l2 :='a';
alumno.apellido1.l3 :='r';
alumno.apellido1.l4 :='c';
alumno.apellido1.l5 :='í';
alumno.apellido1.l6 :='a';
alumno.apellido1.l7 :=0;

alumno.apellido2.l1 :='S';
alumno.apellido2.l2 :='a';
alumno.apellido2.l3 :='n';
alumno.apellido2.l4 :='z';
alumno.apellido2.l5 :=0;

alumno.edad :=19;


### Sentencia de control de flujo bucle condicional
BOOLEANO seguir := true;
```

```
MIENTRAS seguir AND punto1.radio>0 # bucle condicional
SI punto1.centro.x1 <0 AND punto1.centro.x2 <0 # cuerpo si condición
verdadera
    seguir := false;
SINO # cuerpo cuando la condición es falsa
    Seguir := true;
FINSI # fin de condicional
punto1.radio := punto1.radio/2.0;
punto1.centro.x1 := punto1.centro.x1 - 1.0;
punto1.centro.x2 := punto1.centro.x2 - 1.0;

FINMIENTRAS # fin de bucle MIENTRAS
```

En este segundo ejemplo tenemos variables declaradas de tipos construidos en el programa (punto1 de tipo BOLA, alumno de tipo PERSONA). Por tanto, además de la tabla de símbolos con todas las variables, necesitamos una tabla de registros (TR) con los tipos que forman las estructuras declaradas

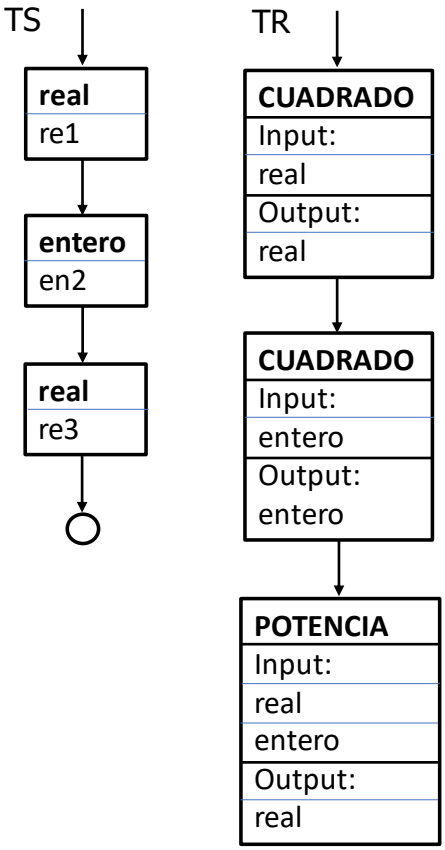


Ejemplo 3:

Ejemplo con declaración y uso de funciones:

```
<!—Definición de funciones
-----------------------------------------------
-----------------------------------------------
-->
FUNCION cuadrado( REAL a ) RETURN REAL
{
    a * a;
}
### Sobrecarga de funciones
FUNCION cuadrado( ENTERO e ) RETURN ENTERO
{
    e * e;
}

### Funcion de dos argumentos
FUNCION potencia( REAl a, ENTERO e ) RETURN REAL
{
    REAL r :=1;
    ENTERO i :=1;
    MIENTRAS i<e
      R := r*a;
      I := i+1;
    FINMIENTRAS
}


ENTERO en2 := 6; # con asignación de valor

### Expresiones
5 + 6; # operacion
re1 := 3.7; # asignación simple
re1 := 3.7 + cuadrado(re1); # usando llamada a función
re2 := potencia(re1,en2);

### Sentencia condicional de control de flujo
SI en2==3 OR bo3 AND re1 # condición
ENTONCES # cuerpo cuando la condición es verdadera
    en2 := en2+3;
SINO # cuerpo cuando la condición es falsa, llamada recursiva
    re1 := cuadrado( cuadrado (en2) );
FINSI # fin de la sentencia
```

En este tercer ejemplo tenemos variables de tipos básicos, y además tres funciones declaradas (CUADRADO, CUADRADO, POTENCIA), las dos primeras con sobrecarga de nombres. Por tanto, además de la tabla de símbolos con todas las variables, necesitamos una tabla de registros (TR) que refleje las funciones, con los tipos que forman la entrada (Input) y salida (Output)


TS
real
re1
entero
en2
real
re3
TR
CUADRADO
Input:
real
Output:
real
CUADRADO
Input:
entero
Output:
entero
POTENCIA
Input:
real
entero
Output:
real

3 Entrega Parcial

En una primera entrega deberá completarse la construcción del analizador léxico del lenguaje y la especificación de la gramática. Como se ha visto, el lenguaje de trabajo comprende símbolos (tokens) pertenecientes a diferentes tipos:

- Palabras reservadas
- Identificadores
- Números, con varios formatos (enteros con base decimal y hexadecimal, reales con parte decimal y notación científica)
- Operadores aritmético-lógicos
- Comentarios, los que comienzen por el símbolo "#" hasta fin de línea, y los delimitados entre "<!—" y "-->"

Puede crearse un método main que nos permita ejecutar el análisis léxico por separado del analizador sintáctico para ejecutar este analizador de forma independiente del compilador. El siguiente código sirve de ejemplo para ello:

```
package cup.example;

import java_cup.runtime.*;
import java.io.*;
import java.util.ArrayList;
import java_cup.runtime.Symbol;
import java_cup.runtime.ComplexSymbolFactory;

class AnalisisLexico {

	public static void main(String[] args) throws Exception {
		//Parser parser = new Parser();
		//parser.parse();

		// Entrada de datos: teclado por defecto, fichero si hay argumento
				InputStream dataStream = System.in;
				ComplexSymbolFactory f = new ComplexSymbolFactory();
				File file = new File("input.txt");
				FileInputStream fis = null;
				try {
				  fis = new FileInputStream(file);
				 } catch (IOException e) {
				  e.printStackTrace();
				 }
				 // Creamos el objeto scanner
				Lexer scanner = new Lexer(f,fis);
				ArrayList<Symbol> symbols = new ArrayList<Symbol>();
				// Mientras no alcancemos el fin de la entrada
				boolean end = false;
				while (!end) {
					try {
						Symbol token = scanner.next_token();
						symbols.add(token);
						end = (token.sym == 0);
						//0 es EOF y por tanto el ultimo
					} catch (Exception x) {
```

```
                                        System.out.println("Ups... algo ha ido mal");
                                        x.printStackTrace();
                                }
                        }
                        //fin while
                        symbols.trimToSize();

                        System.out.println("Numero de tokens:"+symbols.size());
                        for (Symbol simbolo:symbols)
                        {
                             System.out.println("Almacenado: {" + simbolo.sym + " - "+
                                    sym.terminalNames[simbolo.sym]+
                                     "} >>      " + simbolo.toString());
                              if (simbolo.value!=null)
                                 System.out.println("Valor: "+ simbolo.value.toString());
                        }
                        System.out.println("\n\n -- Bye-bye -- ");
        }
}
```

En este momento, tendremos el proyecto listo. Si ejecutamos la clase "AnalisisLexico" como una aplicación java, el script de ant, gererará el fichero "src/lex/generado/Lexer.java" de acuerdo a las reglas establecidas en el fichero "lexer.jflex", se compilará el conjunto y se ejecutará el main de la clase AnalisisLexico que incluye la creación y ejecución de un objeto de la clase Lexer.

Cada grupo debe entregar el contenido de su práctica en un único archivo comprimido –preferiblemente en formato ZIP–. El nombre del comprimido debe ser "`pl_ grupo_XX`", (pe-ele_) donde XX son los apellidos de los integrantes del grupo. El código fuente del proyecto incluirá al menos:

- Archivo .lex/.flex con la especificación de un scanner que sirva a los propósitos de la práctica.
- Clases auxiliares necesarias para ejecutar el analizador léxico y visualizar todos los tokens en la entrada.
- Diseño de la gramática con la estructura sintáctica del lenguaje (sin necesidad de hacerse funcionar aún en cup)

4 Entrega final

En la entrega final, la práctica deberá contener las tareas indicadas (scanner, parser, tabla de símbolos que almacene variables y tipos y análisis semántico). La gramática debe ser diseñada de tal manera que el parser no tenga conflictos de ningún tipo.

4.1 Especificación de la salida

El programa generará información de los errores de tipos del programa, indicando claramente los tipos de error, tales como variables no declaradas, asingación de tipo incompatibles, usar un campo de un ***Struct*** en una expresión invalida, errores de tipos en expresiones o condiciones, llamada a función con tipos incorrectos, etc.

En caso de no tener errores, la salida del programa consistirá en un archivo que liste las variables declaradas y sus tipos. En el caso de no existir estructuras de control de flujo ni llamadas a funciones, el programa además mostrará los valores de las variables tras la ejecución del programa.

4.2 Entrega

Cada grupo debe entregar todo el contenido de su práctica en un único archivo comprimido –preferiblemente en formato ZIP–. El nombre del comprimido debe ser "`pl_ grupo_XX`", (pe-ele_) donde XX son los apellidos de los integrantes del grupo. El código fuente del proyecto incluirá al menos:

- Archivo .lex/.flex con la especificación de un scanner que sirva a los propósitos de la práctica.
- Archivo .cup con la especificación del parser que, trabajando conjuntamente con el scanner anterior, satisfaga los requisitos de la práctica.
- Clases auxiliares necesarias para implementar el compilador (tabla de símbolos, comprobaciones semánticas, etc.)

> - ***IMPORTANTE:*** *todos los proyectos son compilados por el corrector, de forma que NO se deben realizar modificaciones a los ficheros generados por JFlex y CUP, puesto que al compilar de nuevo estos cambios se perderán. No obstante, se permite añadir nuevas clases Java si se considera necesario.*

- Un conjunto de archivos con programas expresiones aritméticas correctamente formadas.
    - Las expresiones correctas deben demostrar el correcto funcionamiento del programa y su traducción a código intermedio.
- En caso de implementar las partes opcionales, se deberá entregar otro conjunto de archivos con pruebas específicas para probar la funcionalidad adicional

- Las expresiones incorrectas deben demostrar que el programa falla con casos no cubiertos en el archivo que se proporciona con el código base.

El archivo comprimido también incluirá una memoria en formato PDF con, al menos, los siguientes apartados:

- **Portada:** asignatura, práctica, composición y nombre del grupo, año lectivo.
- **Tabla de contenidos:** secciones y números de página correspondientes.
- **Introducción**
- **Gramática:** versión final de la gramática implementada, en notación BNF.
- **Breve descripción de la solución:** explicar la solución implementada y los controles semánticos realizados.
- **Archivos de prueba:** explicación de los aspectos probados en los archivos de prueba entregados.
- **Conclusiones:** aspectos adicionales a tener en cuenta por el corrector, y hechos a destacar sobre desarrollo de la práctica (incluyendo, pero no limitándose, a opiniones personales sobre el material de partida, conocimientos y cantidad de trabajo necesaria para su consecución).

4.3 Consideración final

Es muy importante respetar de forma estricta el formato de entrega. De no hacerlo, se corre el riesgo de perder puntos de calificación. Cada grupo responderá de forma totalmente responsable del contenido de su práctica. Esto implica que los autores deben conocer en profundidad todo el material creado por ellos.

Ante dudas de plagio o ayuda externa, el corrector convocará al grupo para responder a cuestiones sobre cualquier aspecto de la memoria o código entregados. En caso de no ser respondidas, se procederá a aplicar la sanción correspondiente.