Universidad La Salle

COMPILADORES

INFORME LENGUAJE DE PROGRAMACIÓN: SERPY

Autor:
Basurco Monroy Luis Gonzalo
Lozano Vega Nicolle Andrea
Saico Ccahuana Katherine Naomi
Silva Cabrera Marcelo Vieri

Índice

1. Justificación y Descripción del Lenguaje

SERPY es un lenguaje de programación diseñado para facilitar el aprendizaje de la programación a hablantes de español. El lenguaje está orientado a la enseñanza de conceptos básicos de programación, como variables, estructuras de control, bucles y funciones, utilizando una sintaxis clara y legible. SERPY se asemeja al lenguajes de programación Python en términos de su estructura y sintaxis:

- *Simplicidad y Legibilidad*: Al igual que Python, SERPY prioriza la legibilidad del código, utilizando palabras clave en español que son intuitivas para los hablantes nativos.
- *Estructuras de Control*: Ambos lenguajes utilizan estructuras de control como `if`, `for`, y `while`, lo que permite a los programadores aplicar conceptos similares en ambos entornos.

2. Ejemplos de Código Fuente

2.1. Hola Mundo

```
imprimir("Hola mundo");
```

2.2. Ejemplo con Bucles Anidados

```
i = 0
mientras i < 3:
    j = 0
    mientras j < 3:
        imprimir(f"Posición: ({i}, {j})")
        j = j + 1
    i = i + 1
```

2.3. Ejemplo con Recursividad

```
definir factorial(n):
    si n == 0:
        retornar 1
    si no:
        retornar n * factorial(n - 1)

imprimir(factorial(5))
```

3. Especificación Léxica

3.1. Tabla de Tokens y Expresiones Regulares

Categoría	Token	Expresión Regular
6*Palabras Reservadas	VAR	var
	RETORNAR	retornar
	IMPRIMIR	imprimir
	DEFINIR	definir
	VERDADERO	verdadero
	FALSO	falso
4*Estructuras de Control	SI	si
	SINO	sino
	MIENTRAS	mientras
	PARA	para
Identificadores	IDENTIFICADOR	[a-zA-Z_][a-zA-Z0-9_]*
2*Literales	NÚMERO	[0-9]+(.[0-9]+)
	CADENA	"([^"//] * (//.[^"//]*)*)"
12*Operadores	IGUAL	=
	IGUALIGUAL	==
	DIFERENTE	!=
	SUMA	+
	RESTA	-
	MULTIPLICACIÓN	*
	DIVISIÓN	/
	MAYOR	¿
	MENOR	¡
	MAYORIGUAL	¿=
	MENORIGUAL	¡=
	YLOGICO	y
	OLOGICO	o
	NEGACIÓN	no
8*Delimitadores	COMA	,
	PUNTOYCOMA	;
	PARIZQ	(
	PARDER	)
	LLAVEIZQ	{
	LLAVEDER	}

Cuadro 1: Tabla de tokens y expresiones regulares para el lenguaje propuesto.

4. Gramática del Lenguaje

La gramática del lenguaje propuesto se define mediante producciones en formato BNF (Backus-Naur Form). Esta gramática ha sido diseñada para ser no ambigua y está factorizada por la izquierda para facilitar el análisis sintáctico LL(1).

```
PROGRAMA -> lista_sentencias

lista_sentencias -> sentencia lista_sentencias
                  | ''

sentencia -> VAR IDENTIFICADOR IGUAL expresion PUNTOYCOMA
           | IDENTIFICADOR asignacion_o_llamada PUNTOYCOMA
           | RETORNAR expresion PUNTOYCOMA
           | IMPRIMIR PAR_IZQ lista_argumentos PAR_DER
    PUNTOYCOMA
           | si_sentencia
           | mientras_sentencia
           | para_sentencia
           | funcion_def

asignacion_o_llamada -> IGUAL expresion
                      | PAR_IZQ lista_argumentos PAR_DER

lista_argumentos -> expresion lista_argumentos_cont
                  | ''

lista_argumentos_cont -> COMA expresion lista_argumentos_cont
                       | ''

si_sentencia -> SI PAR_IZQ expresion PAR_DER bloque
    sino_parte

sino_parte -> SINO bloque
            | ''

mientras_sentencia -> MIENTRAS PAR_IZQ expresion PAR_DER
    bloque

para_sentencia -> PARA PAR_IZQ para_inicio PUNTOYCOMA
    expresion PUNTOYCOMA IDENTIFICADOR IGUAL expresion PAR_DER
     bloque

para_inicio -> VAR IDENTIFICADOR IGUAL expresion
             | IDENTIFICADOR IGUAL expresion

funcion_def -> DEFINIR IDENTIFICADOR PAR_IZQ parametros
    PAR_DER bloque
```

```
37
38 parametros -> IDENTIFICADOR parametros_cont
39             | ''
40
41 parametros_cont -> COMA IDENTIFICADOR parametros_cont
42                 | ''
43
44 bloque -> LLAVE_IZQ lista_sentencias LLAVE_DER
45
46 expresion -> exp_logico_and exp_logico_or_resto
47
48 exp_logico_or_resto -> O_LOGICO exp_logico_and
   exp_logico_or_resto
49                     | ''
50
51 exp_logico_and -> exp_igualdad exp_logico_and_resto
52
53 exp_logico_and_resto -> Y_LOGICO exp_igualdad
   exp_logico_and_resto
54                      | ''
55
56 exp_igualdad -> exp_comparacion exp_igualdad_resto
57
58 exp_igualdad_resto -> op_igualdad exp_comparacion
   exp_igualdad_resto
59                    | ''
60
61 op_igualdad -> IGUAL_IGUAL
62              | DIFERENTE
63
64 exp_comparacion -> exp_suma exp_comparacion_resto
65
66 exp_comparacion_resto -> op_comp exp_suma
   exp_comparacion_resto
67                        | ''
68
69 op_comp -> MAYOR
70          | MENOR
71          | MAYOR_IGUAL
72          | MENOR_IGUAL
73
74 exp_suma -> exp_mult exp_suma_resto
75
76 exp_suma_resto -> op_suma exp_mult exp_suma_resto
77                | ''
78
79 op_suma -> MAS
80          | MENOS
81
```

```
exp_mult -> exp_potencia exp_mult_resto

exp_mult_resto -> op_mult exp_potencia exp_mult_resto
                | ''

op_mult -> MULT
         | DIV

exp_potencia -> exp_unario exp_potencia_resto

exp_potencia_resto -> POTENCIA exp_unario exp_potencia_resto
                    | ''

exp_unario -> NEGACION exp_unario
            | MENOS exp_unario
            | primario

primario -> NUMERO
          | CADENA
          | VERDADERO
          | FALSO
          | IDENTIFICADOR primario_llamada_opcional
          | PAR_IZQ expresion PAR_DER

primario_llamada_opcional -> PAR_IZQ lista_argumentos PAR_DER
                           | ''
```

Listing 1: Gramática del Lenguaje Propuesto

4.1. Análisis de la Gramática

La gramática propuesta ha sido diseñada considerando los siguientes aspectos:

- **No ambigüedad**: Cada derivación posible para una cadena de entrada lleva a una única estructura de árbol sintáctico.
- **Factorización por la izquierda**: Se han eliminado prefijos comunes en las alternativas de una producción, lo que evita conflictos de predicción en el análisis LL(1).
- **Eliminación de recursividad izquierda**: Las producciones con recursividad izquierda (como en EXPRESION y TERMINO) han sido transformadas para usar producciones auxiliares con recursividad derecha (EXPRESION_PRIMA y TERMINO_PRIMA).

La gramática resultante es apropiada para el análisis LL(1), donde cada paso de derivación puede determinarse unívocamente mirando solo el siguiente token de entrada.

5. Implementación del Analizador Sintáctico

La implementación del analizador sintáctico consta de dos componentes principales: un generador de tabla sintáctica LL(1) y el analizador sintáctico propiamente dicho. Ambos componentes están implementados en Python y trabajan juntos para analizar el código fuente del lenguaje propuesto.

5.1. Generador de Tabla Sintáctica LL(1)

El primer componente es un programa que genera la tabla sintáctica LL(1) a partir de la gramática del lenguaje. Este programa está implementado en el archivo `TSLL1.py` y realiza las siguientes funciones:

1. **Carga de la gramática**: Lee un archivo de texto que contiene la gramática en formato BNF.
2. **Identificación de símbolos**: Identifica los símbolos terminales y no terminales de la gramática.
3. **Cálculo de conjuntos FIRST**: Para cada símbolo y producción, calcula el conjunto FIRST, que contiene los terminales que pueden aparecer al principio de las cadenas derivables.
4. **Cálculo de conjuntos FOLLOW**: Para cada no terminal, calcula el conjunto FOLLOW, que contiene los terminales que pueden aparecer inmediatamente después de ese no terminal.
5. **Construcción de la tabla LL(1)**: Utiliza los conjuntos FIRST y FOLLOW para construir la tabla de análisis sintáctico LL(1).
6. **Detección de conflictos**: Verifica si existen conflictos en la tabla, lo que indicaría que la gramática no es LL(1).
7. **Generación de salida**: Guarda la tabla en formato CSV para ser utilizada por el analizador sintáctico.

5.2. Analizador Sintáctico LL(1)

El segundo componente es el analizador sintáctico LL(1) propiamente dicho, implementado en el archivo `arbolito.py`. Este analizador realiza las siguientes funciones:

1. **Carga de la tabla sintáctica**: Lee la tabla sintáctica LL(1) generada previamente desde un archivo CSV.
2. **Análisis de tokens**: Recibe una secuencia de tokens generada por el analizador léxico.
3. **Construcción del árbol sintáctico**: Aplica el algoritmo LL(1) para analizar la secuencia de tokens y construir el árbol sintáctico correspondiente.
4. **Manejo de errores**: Detecta y reporta errores sintácticos durante el análisis.
5. **Visualización del árbol**: Genera código Graphviz para visualizar el árbol sintáctico.

El analizador utiliza una estructura de pila para implementar el algoritmo LL(1) y construye el árbol sintáctico de manera incremental a medida que procesa los tokens de entrada.

5.3. Explicación Detallada de los Códigos

5.3.1. Generador de Tabla Sintáctica (TSLL1.py)

El generador de tabla sintáctica implementa el algoritmo LL(1) para la construcción de tablas de análisis sintáctico. Sus componentes principales son:

- **Clase AnalizadorLL1**: Clase principal que encapsula toda la funcionalidad del generador.
- **Método cargar_gramatica**: Lee un archivo de texto que contiene la gramática y la procesa para identificar los símbolos terminales, no terminales y las producciones.
- **Método calcular_first**: Implementa el algoritmo para calcular los conjuntos FIRST para cada símbolo de la gramática.

- **Método calcular_follow**: Implementa el algoritmo para calcular los conjuntos FOLLOW para cada no terminal.
- **Método construir_tabla_ll1**: Construye la tabla de análisis sintáctico LL(1) utilizando los conjuntos FIRST y FOLLOW.
- **Método guardar_tabla_csv**: Guarda la tabla generada en un archivo CSV para su uso posterior por el analizador sintáctico.

El algoritmo opera de manera iterativa, actualizando los conjuntos FIRST y FOLLOW hasta que no se produzcan más cambios, lo que garantiza que se alcance un punto fijo en el cálculo.

5.3.2. Analizador Sintáctico (arbolito.py)

El analizador sintáctico implementa el algoritmo LL(1) para el análisis de código fuente. Sus componentes principales son:

- **Clase Node**: Representa un nodo en el árbol sintáctico, con atributos para el símbolo, los hijos y el token asociado.
- **Clase AnalizadorSintacticoLL**: Clase principal del analizador sintáctico.
- **Método cargar_tabla_csv**: Lee la tabla sintáctica desde un archivo CSV.
- **Método analizar**: Implementa el algoritmo LL(1) para analizar una secuencia de tokens y construir el árbol sintáctico.
- **Método generar_codigo_graphviz**: Genera código DOT para visualizar el árbol sintáctico utilizando Graphviz.
- **Método limpiar_nodos_epsilon**: Elimina los nodos epsilon (producciones vacías) del árbol para mejorar la visualización.

El analizador opera utilizando una pila para realizar el análisis descendente y construye el árbol sintáctico de manera incremental a medida que procesa los tokens de entrada. Además, proporciona información detallada sobre cada paso del proceso de análisis, lo que facilita la depuración y el entendimiento del algoritmo.

6. Ejemplo de Uso

A continuación, se muestra un ejemplo de uso del analizador sintáctico con una entrada simple:

```
# Ejemplo de codigo en el lenguaje propuesto
IMPRIMIR PAR_IZQ CADENA PAR_DER PUNTOYCOMA
```

Listing 2: Ejemplo de Uso del Analizador Sintáctico

Esta entrada representa la instrucción para imprimir una cadena, que es una de las operaciones básicas del lenguaje propuesto. El analizador sintáctico procesará esta entrada y generará el árbol sintáctico correspondiente.

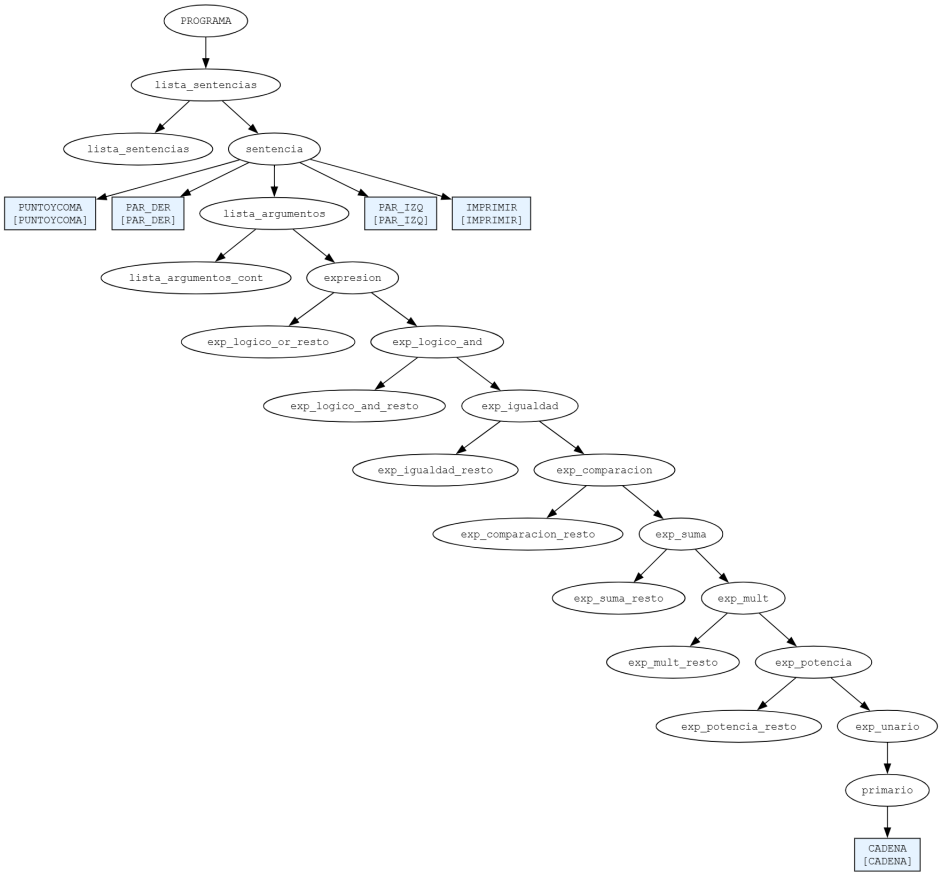

Figura 1: Árbol Sintáctico para IMPRIMIR PAR_IZQ CADENA PAR_DER PUNTOYCOMA

El analizador sintáctico genera un código Graphviz similar al siguiente para visualizar el árbol sintáctico:

```
digraph G {
  rankdir="TB";
  node [shape=ellipse, fontname="Courier", fontsize=12];
  edge [fontname="Courier", fontsize=10];
  2611667668816 [label="PROGRAMA"];
  2611667763280 [label="lista_sentencias"];
  2611667763472 [label="lista_sentencias"];
  2611667763280 -> 2611667763472;
  2611667764496 [label="sentencia"];
  2611667764560 [label="PUNTOYCOMA\n[PUNTOYCOMA]", shape=box,
    style=filled, fillcolor="#e6f3ff"];
  2611667764496 -> 2611667764560;
  2611667765584 [label="PAR_DER\n[PAR_DER]", shape=box, style
   =filled, fillcolor="#e6f3ff"];
  2611667764496 -> 2611667765584;
  2611667765712 [label="lista_argumentos"];
  2611667769936 [label="lista_argumentos_cont"];
  2611667765712 -> 2611667769936;
  2611667770320 [label="expresion"];
  2611667770448 [label="exp_logico_or_resto"];
  2611667770320 -> 2611667770448;
  2611667771856 [label="exp_logico_and"];
  2611667771984 [label="exp_logico_and_resto"];
  2611667771856 -> 2611667771984;
  2611667773520 [label="exp_igualdad"];
  2611667773648 [label="exp_igualdad_resto"];
  2611667773520 -> 2611667773648;
  2611667764368 [label="exp_comparacion"];
  2611667597264 [label="exp_comparacion_resto"];
  2611667764368 -> 2611667597264;
  2611667592464 [label="exp_suma"];
  2611667593104 [label="exp_suma_resto"];
  2611667592464 -> 2611667593104;
  2611667765072 [label="exp_mult"];
  2611667765200 [label="exp_mult_resto"];
  2611667765072 -> 2611667765200;
  2611667767632 [label="exp_potencia"];
  2611667767760 [label="exp_potencia_resto"];
  2611667767632 -> 2611667767760;
  2611667770064 [label="exp_unario"];
  2611667770192 [label="primario"];
  2611667772752 [label="CADENA\n[CADENA]", shape=box, style=
   filled, fillcolor="#e6f3ff"];
  2611667770192 -> 2611667772752;
  2611667770064 -> 2611667770192;
  2611667767632 -> 2611667770064;
  2611667765072 -> 2611667767632;
  2611667592464 -> 2611667765072;
  2611667764368 -> 2611667592464;
```

```
  2611667773520 -> 2611667764368;
  2611667771856 -> 2611667773520;
  2611667770320 -> 2611667771856;
  2611667765712 -> 2611667770320;
  2611667764496 -> 2611667765712;
  2611667765840 [label="PAR_IZQ\n[PAR_IZQ]", shape=box, style
   =filled, fillcolor="#e6f3ff"];
  2611667764496 -> 2611667765840;
  2611667765968 [label="IMPRIMIR\n[IMPRIMIR]", shape=box,
   style=filled, fillcolor="#e6f3ff"];
  2611667764496 -> 2611667765968;
  2611667763280 -> 2611667764496;
  2611667668816 -> 2611667763280;
}
```

Listing 3: Código Graphviz Generado para el Árbol Sintáctico

7. Conclusiones

La implementación del analizador sintáctico para el lenguaje propuesto demuestra la aplicación práctica de los conceptos teóricos de compiladores, específicamente las fases de análisis léxico y sintáctico. El generador de tabla sintáctica y el analizador sintáctico trabajan juntos para procesar el código fuente del lenguaje y generar una representación estructurada en forma de árbol sintáctico.

Las principales conclusiones son:

- El método LL(1) proporciona un enfoque sistemático y eficiente para el análisis sintáctico descendente.
- La transformación de la gramática para eliminar la recursividad izquierda y factorizar por la izquierda es esencial para la aplicación del análisis LL(1).
- La visualización del árbol sintáctico mediante herramientas como Graphviz facilita la comprensión de la estructura del código analizado.
- La implementación modular permite una clara separación de responsabilidades entre la generación de la tabla sintáctica y el proceso de análisis.

Este proyecto sirve como base para futuras expansiones, como la implementación de un analizador semántico y un generador de código, que completarían el proceso de compilación para el lenguaje propuesto.

8. Conclusión

SERPY se inspira en la simplicidad y la estructura de lenguajes como Python y JavaScript, lo que lo convierte en una herramienta efectiva para la enseñanza de la programación en español. Esta relación con lenguajes conocidos facilita la transición y el aprendizaje para los nuevos programadores.