Procesadores del Lenguaje

Descendente recursivo

Autores: Álvaro Marco Pérez (100383382), David Rico Menéndez (100384036)
Datos: Grupo de trabajo 2

Índice

1. Diseñad una gramática que represente las expresiones aritméticas anteriormente definidas.

Para el desarrollo de una gramática que acepte operaciones aritméticas tales como: **(* (+ 1 2) 3) \n** se itera sobre varias idea iniciales para esta gramática, resolviendo aquellos casos en los que la gramática impide la generalización de los casos base, obteniendo finalmente la siguiente gramática:

A::= Expresión

Expresión ::= (- Término Término) |

(+ Término Término) |

(* Término Término) |

(/ Término Término) |

Término

Término ::= Número | Expresión

2. Determinad si es necesario transformar la gramática anterior para que cumpla con las condiciones LL(1). Utilizad la herramienta jflap.

El primer paso para hacer uso de la herramienta JFLAP ha sido simplificar la gramática, resultando:

A ::= E

E ::= (oTT) |

T

T ::= n |

E

Como se puede apreciar, se ha sustituido el términal *Número* por *n*, con el objetivo de que JFLAP reconociese la sintaxis. Del mismo modo, se ha simplificado lo símbolos *+, -, *, /* por un terminal *o*, que representa cada uno de ellos.

La conversión de esta gramática a LL(1) ha producido los siguientes resultados:

No Terminal	Primero	Siguiente
A	{ (, n }	{$}

E	{ (, n }	{$, (,), n}
T	{ (, n }	{$, (,), n}

	(	)	n	o	$
A	E		E		
E	(oTT), T		T		
T	E		E, n		

Se puede apreciar que el *Parser* LL(1) se ha realizado correctamente y, consecuentemente, la gramática puede ser trasladada a código.

3. Desarrollad un Parser Descendente Recursivo conforme a la gramática del punto 2 para procesar y evaluar expresiones en notación prefija.

4. Ampliad la gramática para manejar variables simples (un carácter, mayúsculas o minúsculas).

a. Una Variable estará representada por un Token específico, y podrán intervenir en una expresión, al igual que un Numero.

Para ello, se añade en **Término**, la derivación **Variable**

b. Se puede asignar a una Variable el valor de un Numero o de una Expresion mediante la expresión (! Variable Parametro). En este caso, el Operador ! almacena el valor del último Parametro en la Variable especificada.

Para ello se añade a **Expresión** la derivación **(! Variable Expresión)**

c. La entrada A \n debe generar el valor de la variable A.

Ya se contempló e implementó en el apartado a.

A ::= E

E ::= (oTT) |

(avT) |

T

T ::= n |

v |

E

Correspondiendo el término *v* a *Variable* y *a* al operador *!*:

La construcción del *Parser* LL(1) sobre la simplificación descrita de la gramática original resulta en las siguientes tablas:

No Terminal	Primero	Siguiente
A	{ v, (, n }	{$}
E	{ v, (, n }	{$, v, (,), n}
T	{ v, (, n }	{$, v, (,), n}

	(	)	a	n	o	v	$
A	E			E		E	
E	(avT), (oTT), T			T		T	
T	E			E, n		E, v	

5. Ampliad la gramática para permitir dos o más Parametros en las Expresiones. Por ejemplo, (- 1 1 1 1) debe generar -2 en la salida.

Para realizar la ampliación de la gramática, se rehace **Expresión** añadiendo un término **restoExpresión**, que permite la escritura de más de dos operandos en la operación.

La gramatica resultante y final despues de todos estos pasos es la siguiente:

A::= Expresión

Expresión ::= (- Expresión Resto) |

(+ Expresión Resto) |

(* Expresión Resto) |

(/ Expresión Resto) |

(! Variable Expresión) |

Término

Resto ::= Expresión Resto | lambda

Término ::= Número | Variable

Del mismo modo que en los anteriores apartados, se realiza una simplificación de la gramática para su transformación en el *Parser* LL(1), haciendo uso de la herrramienta JFLAP.

La gramática resultante es:

A ::= E

E ::= (oTR) |

(avE) |

T

R ::= ER |

λ

T ::= n |

v |

E

Que produce en su conversión los siguientes resultados:

No Terminal	Primero	Siguiente
A	{ v, (, n }	{$}
E	{ v, (, n }	{$, v, (,), n}
R	{ λ, v, (, n }	{) }
T	{ v, (, n }	{$, v, (,), n}

	(	)	a	n	o	v	$
A	E			E		E	
E	(avT), (oTT), T			T		T	
R	ER	λ		ER		ER	
T	E			E, n		E, v	

Para la comprobación de la gramática se ha hecho un fichero de pruebas con entradas de diversa índole. Dicho fichero contiene cincuenta entradas distintas a la gramática en la que se comprueban las diferentes características añadidas al *Parser*. Las entradas se pueden agrupar según la intencionalidad de la comprobación en los siguientes puntos:

Entrada	Comprobación
(* 1 4)	La gramática acepta operaciones prefijas

(+ 1 2 1)	La gramática acepta operaciones prefijas con más de dos operandos
(+ 1 2 1 2)	La gramática acepta operaciones prefijas con múltiples operandos
(+ 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2)	La gramática acepta operaciones prefijas con múltiples operandos
(+ (+ 1 2)(+ 1 2))	La gramática acepta operaciones prefijas concatenadas
(+(+ (+ 1 2)(+ 1 2))(+ (+ 1 2)(+ 1 2)))	La gramática acepta operaciones prefijas concatenadas y jerarquizadas
(+(+ (+ 1 2)(+ 1 2))(+ (+ 1 2)(+ (+ 1 2)(+ (+ 1 2)(+ (+ 1 2)(+ 1 2))))))	La gramática acepta operaciones prefijas concatenadas y con jerarquía concatenada
(! A 2)	La gramática acepta asignaciones de variables
(+ 1 A)	La gramática acepta operaciones con variables
(- A 3)	La gramática acepta operaciones con variables
(! A (+ 1 2))	La gramática acepta operaciones con variables y operaciones concatenadas
(! A (-(+ 1 2)(+ 1 2)))	La gramática acepta operaciones con variables y multiples operaciones concatenadas
(! A B)	La gramática acepta operaciones con múltiples variables
(! A (! b (! c (! D (! e (! F (! G (! h 3))))))))	La gramática acepta asignaciones concatenadas
A	La gramática acepta mostrar variables
1123456	La gramática acepta mostrar números

La gramática diseñada acepta todos las pruebas realizadas.

7.Conclusiones

La práctica ha resultado sencilla y ha servido para complementar los conocimientos adquiridos con los conceptos teóricos de la asignatura.