Procesadores de Lenguaje

GRADO EN INGENIERÍA INFORMÁTICA

Práctica Final: Grupo 3

Curso 2020/2021

Jorge Rodríguez Fraile, 100405951, Grupo 83, 100405951@alumnos.uc3m.es
Carlos Rubio Olivares, 100405834, Grupo 83, 100405834@alumnos.uc3m.es

Índice

Explicación de lo implementado:

Punto 1

Para el primer paso hemos cambiado en la impresión de expresiones el carácter '#' por '$', para esto, hemos sustituido dicho carácter en las reglas de producción de impresión.

Punto 2

Para poder introducir más de un parámetro en la función de impresión que se llama con '$(parámetros);', hemos creado un nuevo no terminal llamado 'impr' que represente las expresiones impresas. Este nuevo no terminal puede ser una expresión o una expresión, coma e impr.
De esta manera se permite imprimir una sola expresión o generar una secuencia de impresiones manteniendo el orden en el que se introdujeron.

Punto 3

En este caso nos hemos fijado en la regla de producción *INTEGER IDENTIF;* esta regla generará una producción parecida al del comando 'setq', pero en este caso concatenamos un '0' para inicializar la variable. La introducción de este paso ha sido bastante sencilla en este caso.

Punto 4

En este punto hemos decidido extraer las asignaciones en un nuevo no terminal llamado 'setq', en el que hemos metido las anteriores asignaciones creadas y hemos utilizado una estructura muy similar a la del punto anterior, pero en este caso en vez de terminar recibiendo 'IDENTIF;' recibe 'IDENTIF=NUMERO;'.

Punto 5

Para la creación de main hemos utilizado la palabra reservada *MAIN*, y creado un no terminal llamado 'mainfun', este NT se llama en def y puede ser vacío. Dentro de 'mainfun' tenemos la cabecera, un cuerpo que pueden ser asignaciones de variables o cualquier expresión (se explicará en el punto 7) y el cierre de la llave de la función.

Punto 6

Para poder establecer que primero se encuentren solo asignaciones y posteriormente se definan todas las funciones, hemos cambiado el axioma para que produzca la regla 'dcl def'.
'dcl' solo podrá generar asignaciones o nada.
'def' como se ha descrito anteriormente produce las funciones, incluido main. Aunque puede no haber funciones declaradas.

Punto 7

Para esta funcionalidad nos fijamos en el no terminal cuerpo que es el que tiene todas las expresiones permitidas en la función. Este elemento puede derivar en una impresión seguida de cuerpo, una asignación de variable, o una expresión, también tenemos en cuenta que pueda ser vacío.

Punto 8

En esta parte para poder conseguir que se puedan hacer varias asignaciones en la misma sentencia hemos añadido en aquellas sentencias de asignación que empiezan por int la posibilidad de encadenar asignaciones o declaraciones de variables mediante ‘resdecl‘, que puede ser vacío para que solo sea una asignación, una coma y expresión de asignación o una variable a declarar.

Punto 9

Este punto ha sido bastante fácil ya que ha sido eliminar la regla de producción que daba únicamente expresión en el cuerpo de lo que deseemos escribir.

Punto 10

En este caso hemos creado una palabra reservada PUTS, para después aplicarla en nuestro NT ‘cuerpo’, aunque tras varias modificaciones, ha acabado en un nuevo NT denominado ‘sent’, cuya creación se explicará más adelante. En esta regla de producción simplemente leemos el string que se nos pasa, utilizando el token indicado en el enunciado y la ponemos entre comillas precedido de un ‘print‘.

Punto 11

Como se indica en lugar de utilizar el símbolo $ para la impresión lo sustituimos por el token PRINT, que representa la palabra reservada ‘printf‘, también como ‘puts‘ es la sentencia para la impresión de cadenas de caracteres cuando ‘printf‘ se encuentre una no hará nada. De tal manera que ahora se puede imprimir con ‘printf‘ solo expresiones y con ‘puts‘ solo cadenas de caracteres.

Punto 12

Para este punto hemos creado una palabra reservada para And, or, !=, >=,<= y ==, después de esto hemos creado un nuevo NT denominado ‘cond’ para poder separarlas del resto de expresiones y facilitar futuras expansiones de la gramática. En cada una de las reglas de producción de este nuevo NT introducimos un modelo de *expresión símbolo expresión* donde símbolo es cada una de las nuevas palabras reservadas creadas. Para conectarla al resto de la gramática, el NT ‘expresión’ tiene una nueva regla de producción que da ‘cond’.

Punto 13

Para que fuera posible crear el bucle ‘while‘ primero tuvimos que crear la palabra reservada correspondiente, de tal manera que si detecta ‘while‘ a continuación esperará una condición, dada por ‘cond‘, entre paréntesis seguido de una serie de secuencias, que son de cuerpo, entre llaves. Entre la recepción de la condición y la espera de las sentencias se pone el do de ‘lisp‘. De esta manera dentro de la condición puede haber cualquier clase de condición y en el cuerpo del bucle cualquier sentencia incluido otro bucle.

Punto 14

La estructura del bucle if es bastante parecida al del ‘while‘, el problema ha surgido al implementar la expansión de ‘else’, ya que daba problema de conflicto, para esto, hemos creado un nuevo ‘NT’ denominado ‘els’, que puede dar la rama de else o bien nada, esto nos ha resuelto los problemas de conflicto que teníamos.

Punto 15

En este punto hemos desarrollado el bucle for, para esto hemos creado un nuevo no terminal llamado ‘decliter‘, y otro llamado ‘iter‘. Su función es bastante autoexplicativa; la primera se encarga de la inicialización del iterador en la declaración del bucle, y la segunda en su iteración. Por último, dentro del bucle se puede generar el NT ‘cuerpo‘ y también después de este, para poder proseguir con las instrucciones.

Punto 16

Para la creación de vectores añadimos a nuestros NT ‘setq‘ y ‘decl‘ una nueva regla que reconozca el tipo INTEGER, nombre como IDENTIF y tamaño NUMBER entre llaves, que se traduce al correspondiente ‘(setq nombre (make-array tamaño))‘.

En cuanto a la posibilidad de acceder a los valores de un array añadimos al NT termino la regla que produce ‘vec‘, de tal manera que ‘vec‘ es el no terminal que reconoce la secuencia nombre como IDENTIF y el correspondiente índice al que se desea acceder, para traducirlo como ‘(aref nombre indice)‘.

Por último, nos queda que se pueda asignar valor a las posiciones del vector, al igual que la creación, en los NT ‘setq‘ y ‘decl‘ se añade una nueva regla que llama al ‘vec‘ definido antes seguido de un símbolo igual y el valor a asignar. El valor que asignar será una ‘expresion‘ o NUMERO. Este devolverá ‘(setf vec valor)‘ terminado de ‘;’ y con la posibilidad de escribir más sentencias.

Punto 17

Para este apartado se han creado dos NT principales, ‘fun‘ y ‘callfun’. ‘fun‘ se encarga de crear la estructura de la función y es llamada en def (esto lo hemos hecho para evitar problemas). A su vez, ‘fun‘ llama a ‘inipar‘, que como su nombre indica es el que reconoce los parámetros de la función (de 0 a n). Para conseguir la iteración de ‘inipar‘, es decir, que consiga aceptar más de un parámetro, hemos creado ‘resinipar‘, utilizando una recursividad a derechas. La estructura de los parámetros viene dada por el NT par, se podría haber incluido en ‘inipar‘, pero así la gramática queda estructurada de una manera más jerárquica. Por último, ‘callfun‘ se utiliza para llamadas a funciones, que utiliza la llamada del NT ‘par‘.

En cuanto a los returns, hemos creído conveniente crear una variable llamada ‘nombre_fun‘ que almacene el nombre de la función que se ha llamado para luego incluirla en la sintaxis de ‘return-from‘, lo que nos permite hacer varios returns dentro de la función. La regla de producción de returns ha sido introducida en el NT cuerpo.