MEMORIA

Procesadores de lenguaje

Trad 5

Course 2021/2022

DEGREE IN COMPUTER ENGINEERING

GROUP 84

Carlos Gallego Jiménez 100405937
Andrés Langoyo Martín 100405869

Especificaciones

1. Hemos cambiado el símbolo '#' por '$' en la gramática del no terminal *impresion* y hemos añadido paréntesis para englobar la *expresión* a imprimir*:*

***impresion**: '$' '(' expresion ')' {sprintf (temp, "(print %s), $3,);*
$$ = genera_cadena (temp) ;} ;

2. Para incluir varios parámetros en el print, creamos nuevas reglas en la gramática:
   impresion ::= "(" expresion r_impresión
   r_impresion ::= , expresion r_impresion
   | ")"
   Aquí utilizamos las reglas de resto de expresión para continuar añadiendo parámetros recursivamente o añadir el paréntesis de cierre.

   En cuanto a la acción semántica, para la primera regla metemos en un print la expresión que nos encontramos y lo concatenamos con la siguente llamada a r_expresión.

   {sprintf (temp, "(print %s) %s" , $2, $3);
   $$ = genera_cadena (temp) ;}

   Para la segunda regla, simplemente devolvemos un paréntesis de cierre.

   {sprintf (temp, ") ");
   $$ = genera_cadena (temp) ;}

3. Hemos incluido las variables globales en el documento a partir del no terminal *v_global* que se deriva del no terminal *sentencia* de la siguiente manera y deriva en dos tokens *INTEGER Y INDETIF* :
   sentencia ::= v_global
   v_global ::= INTEGER IDENTIF

   La acción semántica correspondiente es la siguiente:
   { sprintf (temp, "(setq %s 0)", $2);
   $$ = genera_cadena (temp) ; }

4. Para incluir las variables con asignación incluida hemos modificado *v_global* y hemos añadido a la gramática *v_asignación* de manera que cuando no se incluya asignación la semántica de *v_asignación* devuelva "0":

   v_global ::= INTEGER IDENTIF v_asignación
   v_asignación ::= /*lambda*/ { sprintf (temp, "0");
   $$ = genera_cadena (temp);}
   | '=' NUMERO |{ sprintf (temp, "%d", $2);

$$ = genera_cadena (temp);}

5. [1]

En este punto cambiamos la gramática para que, en lugar de reconocer diversas sentencias, tradujera un programa entero formado por declaraciones de variables y definición de funciones. Para ello cambiamos la regla del *axioma*:
axioma: def_variables def_funciones

Y en la acción semántica, imprimimos ambas junto con la ejecución del main:

{ printf ("%s %s\n(main)", $1, $2) ; }

Para las variables, utilizamos el no terminal creado anteriormente, *v_global* y, a continuación usamos un nuevo terminal añadir varias variables recursivamente. Este terminal es *r_variables*.

En la acción semántica, simplemente concatenamos las diversas definiciones que nos meten.

***def_variables**: v_global ';' r_variables { sprintf (temp, "%s \n%s ", $1, $3);*
$$ = genera_cadena (temp) ;}
;

r_variables: def_variables { sprintf (temp, "%s", $1);
$$ = genera_cadena (temp) ;}
| /*lambda*/ { sprintf (temp, "");
$$ = genera_cadena (temp) ;}
;

Para las funciones, como de momento solo nos piden representar el main, incluímos una regla que derive en el no terminal del main, *iniciar.*

def_funciones: iniciar { sprintf (temp, "(defun %s)", $1);
$$ = genera_cadena (temp) ;}
;

Ahora, *iniciar* reconoce la sintaxis típica del main y, dentro de los corchetes, utilizamos un nuevo no terminal para introducir diferentes sentencias (*r_sentencias*).

***iniciar**: MAIN '(')' '{' r_sentencias '}' { sprintf (temp, "%s () \n %s", $1, $5);*
$$ = genera_cadena (temp) ;}

La regla de *r_sentencias* concatena diversas sentencias que el usuario pueda introducir:

```
r_sentencias: /*lambda*/                { sprintf (temp, "");
                                          $$ = genera_cadena (temp) ;}
        | sentencia ';' r_sentencias     { sprintf (temp, "\t %s \n %s", $1, $3);
                                           $$ = genera_cadena (temp) ;}
        ;
```

Además, se aprovechan la definición de sentencias que se tenía anteriormente:

```
sentencia:    expresion          { $$ = $1; }
        |  asignacion            { $$ = $1; }
        |  impresion             { $$ = $1; }
        |  v_global              { $$ = $1; }
        ;
```

6. Como comentamos en el punto anterior el axioma deriva en

def_variables *: v_global ';' r_variables*

Y su semántica es la siguiente:

```
{ sprintf (temp, "%s \n%s ", $1, $3);
                          $$ = genera_cadena (temp) ;}
        ;
```

v_global es el no terminal usado para las variables globales y deriva en:

INTEGER IDENTIF v_asignacion v_multiple

y su semántica es la siguiente:

```
 { sprintf (temp, "( setq %s %s ) %s", $2, $3, $4);
                                  $$ = genera_cadena (temp) ; }
```

v_asignacion es el no terminal usado para inicializar variables globales ya que deriva en :

```
/*lambda */                     { sprintf (temp, "0");
                                  $$ = genera_cadena (temp);}
        | '=' NUMERO                        { sprintf (temp, "%d", $2);
                                  $$ = genera_cadena (temp);}
        ;
```

Podemos observar que si no inicializamos la variable esta se inicializará con el predeterminado, que es 0.

También hemos incluido el no terminal r_variables para añadir recursividad con el objetivo de agregar varias variables una después de otra.
def_funciones ha sido explicado en el punto anterior.

7. En el apartado siete nos piden incluir variables locales pero por la estructura de nuestra gramática ya están incluidas debido a que el no terminal *sentencia* deriva en *v_global* explicada en la especificación 6.

8. Para incluir la múltiple definición de variables utilizamos el no terminal *v_múltiple* mencionado en la especificación 6.
   Este no terminal deriva en *',' IDENTIF v_asignacion*
   y su semántica es la siguiente:

```
   { sprintf (temp, "( setq %s %s )", $2, $3);
                                  $$ = genera_cadena (temp) ; }
```

De esta manera conseguimos una definición de variables múltiple con asignación o sin ella.

[1] A partir de la especificación 5 se modificó la gramática de manera significativa.

9. Eliminamos de la gramática la regla que deriva *sentencia* a *expresión*.

10. Para incluir la impresión de los literales en nuestra gramática incluimos un nuevo no terminal en el que deriva *sentencia*, como se puede observar en el siguiente fragmento de código :

***sentencia**: imprimir_cadena { $$ = $1; }*

***imprimir_cadena**: PUTS '('STRING')' {sprintf (temp, "(print \"%s\")" ,$3);*
$$ = genera_cadena (temp) ;}

Para reconocer la secuencia "*puts*", añadimos "puts" a las palabras reservadas y creamos un token de cadena para *PUTS.*

11. Como en el apartado anterior primeramente añadimos *printf* a las palabras reservadas y seguido creamos un token *PRINTF* de tipo cadena.

Para llevar a cabo en la gramática estos cambios creamos el nuevo no-terminal *argumento.* El proceso ha sido el siguiente: hemos reemplazado la *expresion* y el *r_mpresio*n que utilizabamos por el nuevo no terminal que nos permite intercalar argumentos tanto cadenas como las expresiones con la diferencia de que cuando traducimos un argumento que sea cadena no realizamos ninguna acción semántica para su traducción. Además volvemos a reemplazar *expresion y r_impresion* por *argumento* para garantizar la recursividad que nos permite incluir múltiples argumentos. Adjuntamos el código modificado, gramática y acciones semánticas:

***impresion**: PRINTF '(' argumento {sprintf (temp, "%s" , $3);*
$$ = genera_cadena (temp) ;}
;

***argumento**: expresion r_impresion {sprintf (temp, "(print %s) %s" , $1, $2);*
$$ = genera_cadena (temp) ;}

| STRING r_impresion {sprintf (temp, "%s" , $2);
$$ = genera_cadena (temp) ;}
;

***r_impresion**: ',' argumento {sprintf (temp, "%s" , $2);*
$$ = genera_cadena (temp) ;}
| ')' {sprintf (temp, "");
$$ = genera_cadena (temp) ;}
;

12. Para implementar los operadores lógicos y aritméticos y de comparación hemos primeramente implementado tokens de tipo cadena para los nuevos operadores además de añadirlos a las palabras reservadas.

```
%token <cadena> OR          // identifica el operador '||'
%token <cadena> AND         // identifica el operador '&&'
%token <cadena> COMPARISON  // identifica el operador '=='
%token <cadena> DIFFERENCE  // identifica el operador '!='
%token <cadena> LESS_EQUAL  // identifica el operador '<='
%token <cadena> GREATER_EQUAL // identifica el operador '>='
```

Después de llevar a cabo este procedimiento, los hemos implementado en la gramática de manera que derivaran de *expresion* junto al resto de operadores y hemos traducido a Lisp en la acción semántica:

```
expresion:   termino                                    { $$ = $1 ;}
        | expresion '+' expresion          { sprintf (temp, "( + %s %s )", $1, $3);
                                           $$ = genera_cadena (temp) ; }
        | expresion '-' expresion          { sprintf (temp, "( - %s %s )" , $1, $3);
                                           $$ = genera_cadena (temp) ; }
        | expresion '*' expresion          { sprintf (temp, "( * %s %s )" , $1, $3);
                                           $$ = genera_cadena (temp) ; }
        | expresion '/' expresion          { sprintf (temp, "( / %s %s )" , $1, $3);
                                           $$ = genera_cadena (temp) ; }
        | expresion '<' expresion      { sprintf (temp, "( < %s %s )", $1, $3);
                                                $$ = genera_cadena (temp) ; }
        | expresion '>' expresion      { sprintf (temp, "( > %s %s )" , $1, $3);
                                           $$ = genera_cadena (temp) ; }
        | expresion LESS_EQUAL expresion { sprintf (temp, "( <= %s %s )" , $1,
$3);                                          $$ = genera_cadena (temp) ; }
        | expresion GREATER_EQUAL expresion { sprintf (temp, "( >= %s %s )" ,
$1, $3);                                  $$ = genera_cadena (temp) ; }

        | expresion COMPARISON expresion { sprintf (temp, "( = %s %s )", $1, $3);
                                  $$ = genera_cadena (temp) ; }
        | expresion DIFFERENCE expresion { sprintf (temp, "( //= %s %s )" , $1,
$3);
                                  $$ = genera_cadena (temp) ; }
        | expresion AND expresion          { sprintf (temp, "( AND %s %s )" , $1,
$3);
                                  $$ = genera_cadena (temp) ; }
        | expresion OR expresion           { sprintf (temp, "( OR %s %s )" , $1,
$3);
                                  $$ = genera_cadena (temp) ; }
        ;
```

Por último dado que debíamos de tener en cuenta la asociatividad y la preferencia de los operadores hemos incluido este fragmento en el código:

%right '=' // es la ultima operacion que se debe realizar
%left OR
%left AND
%left COMPARISON DIFFERENCE
%left LESS_EQUAL GREATER_EQUAL'>' '<'
%left '+' '-' // menor orden de precedencia
%left '' '/' // orden de precedencia intermedio*
%left SIGNO_UNARIO // mayor orden de precedencia

Es importante mencionar que al incluir los operadores de mayor y menor como tokens y palabras reservadas generan un conflicto en la gramática por tanto debemos incluirlos de manera literal en las preferencias.

Al probar los tests, incluimos los operadores de módulo, con la misma preferencia que la multiplicación y división.

expresion: expresion '%' expresion
expresion_aritmetica: expresion '%' expresion
%left '*' '/' '%'

13. Para implementar el bucle *while* solo hemos tenido que añadir a las palabras reservadas *while* ya que el token venía creado.
    En la gramática lo hemos implementado derivandose de *r_sentencias* para evitar el problema con el punto y coma. De esta manera podemos incluir código dentro del *while* como ya hacíamos sin llevar a cabo ningún cambio significativo. Además en la acción semántica incluimos la traducción a Lisp:

    ***iniciar**: MAIN '(')' '{' r_sentencias '}' { sprintf (temp, "%s () \n %s", $1, $5);*
    $$ = genera_cadena (temp) ;}

    ***r_sentencias**: WHILE '('expresion')' '{' r_sentencias '}' r_sentencias*

    { sprintf (temp, "(loop while %s do \n%s)", $3, $6);
    $$ = genera_cadena (temp) ;}
14. Lo primero que llevamos a cabo es reservar las palabras *if* y *else* además de añadir sus respectivos tokens.

    Para construir la gramática hemos introducido 3 nuevos no terminales *r_sentencia, r_if* y *r_else.*

    r_sentencia y r_sentencias se utilizan para detectar cuando estamos añadiendo una sentencia o más en los *if* y los *else* para añadir la función *progn* de esta manera si *r_sentencias* nos devuelve una string vacía entendemos que solo hay una sentencia en el *if.*.

r_sentencias**: IF '('expresion')' '{'r_sentencia r_sentencias '}' **r_if

acción semántica:

```
{ if (strcmp("", $7) == 0){

sprintf (temp, "( if %s \n\t%s %s", $3, $6, $9);
 }
else{
sprintf (temp, "( if %s \n\t( progn \n %s %s \t)\n %s", $3, $6, $7, $9);

}

$$ = genera_cadena (temp) ;}
```

El no terminal *r_sentencia* fue creado con la intención de eliminar la recursividad en *r_sentencias* para poder detectar cuantas sentencias se incluían en el *if* y el *else.*

***r_sentencia**:*

```
        sentencia ';'                                   { sprintf (temp, "\t %s \n", $1);
                                                $$ = genera_cadena (temp) ;}
        | WHILE '('expresion')' '{' r_sentencias '}'        { sprintf (temp, "( loop
while %s do \n\t%s )", $3, $6);
                                                $$ = genera_cadena (temp) ;}
        | IF '('expresion')' '{'r_sentencia r_sentencias '}' r_else     { if (strcmp("", $7)
== 0){
                                                        sprintf (temp, "( if %s \n\t%s", $3,
$6);

                                                }
                                                else{
                                                        sprintf (temp, "( if %s \n\t( progn
\n %s %s \t)\n %s", $3, $6, $7, $9);

                                                        }

                                                        $$ = genera_cadena (temp) ;}
        ;
```

r_if nos permite escribir la parte del *else* en el caso de que sea necesario. Si no lo es, derivamos en *r_sentencias* otra vez, para poder poner más sentencias. En el caso de que si se introduzca la parte del *else,* introducimos una estructura similar a la parte del *if* y añadimos la posibilidad de escribir más sentencias a continuación con *r_sentencias.*

r_if:

```
        r_sentencias                                    { sprintf (temp, ") \n%s",$1);
                                                $$ = genera_cadena (temp) ;}
        | ELSE '{'r_sentencia r_sentencias'}' r_sentencias

                                                        { if (strcmp("", $4) == 0){
                                                sprintf (temp, " %s ) \n%s", $3, $6);
```

```
                                        }
                                        else{

                                          sprintf (temp, "\n\t( progn \n %s %s \t)\n
                                          \n%s", $3, $4, $6);

                                        }
                                        $$ = genera_cadena (temp) ;}
```

El no terminal r_elseI nos permite escribir la parte del else cuando solo se incluye una sentencia .

```
r_else:     /*lambda*/                         { sprintf (temp, ")");
                                        $$ = genera_cadena (temp) ;}

        | ELSE '{'r_sentencia r_sentencias'}'       { if (strcmp("", $4) == 0){
                                              sprintf (temp, " %s )", $3);

                                        }
                                        else{

                                  sprintf (temp, "\n\t( progn \n %s %s \t)\n", $3, $4);

                                        }
                                        $$ = genera_cadena (temp) ;}
```

15. Para empezar a implementar el bucle *for* como hemos hecho en anteriores pasos introducimos esta palabra entre las reservadas y creamos su token. Después, procedemos a implementarlo en la gramática teniendo en cuenta que debemos incluir una versión de *for* sin recursividad para cuando se incluya dentro de un *if* con una sola sentencia. Para traducir utilizamos el bucle while como está indicado en la documentación.

```
r_sentencias: FOR '('IDENTIF '=' expresion_aritm ';' expresion_cond ';' IDENTIF '='
expresion_aritm ')'
 '{' r_sentencias '}' r_sentencias                  { sprintf (temp, "( setq %s
%s )\n( loop while %s do \n%s \n( setq %s %s )\n)\n%s ",$3, $5, $7, $14, $9, $11, $16);
                                        $$ = genera_cadena (temp) ;}
```

Por otra parte también hemos modificado las expresiones al dividirlas en arimeticas y condicionales con dos nuevos no terminales:

```
expresion_aritm:   termino
        | expresion '+' expresion
        |  expresion '-' expresion
        |  expresion '*' expresion
        ;

expresion_cond:  expresion '<' expresion
```

| *expresion '>' expresion*
| *expresion LESS_EQUAL expresion*
| *expresion GREATER_EQUAL expresion*
| *expresion COMPARISON expresion*
| *expresion DIFFERENCE expresion*
| *expresion AND expresion*
| *expresion OR expresion*
;

16. Para implementar los vectores nos hemos servido de la gramática ya creada por las variables, evitando tener que añadir más no terminales.
    Para declarar los vectores hemos utilizado el no terminal *v_asignacion* y *v_global* de esta manera nuestros vectores adquieren recursividad. Se pueden intercalar varias definiciones de vectores con variables en una misma línea.

***v_global**: INTEGER IDENTIF v_asignacion v_multiple*

***v_asignacion:** '['expresion_aritm']'* *{ sprintf (temp, "(make-array %s)", $2);*
$$ = genera_cadena (temp);}

impresion: *PRINTF '(' argumento*

Para imprimir nos servimos de *argumento* un no terminal que hemos utilizado previamente para imprimir diferentes expresiones, esto también le concede recursividad a la impresión de vectores y se pueden intercalar con otras impresiones de variables o expresiones.

***argumento:** IDENTIF '['expresion_aritm']' r_impresion*
{sprintf (temp, "(print (aref %s %s)) %s" , $1, $3, $5);
$$ = genera_cadena (temp) ;}

La asignación de variables la implementamos con el no terminal *asignación.* Para los vectores, hemos añadido una nueva regla similar:

***asignacion:** IDENTIF '['expresion_aritm']' '=' expresion*

{ sprintf (temp, "(setf (aref %s %s) %s) " , $1, $3, $6);
$$ = genera_cadena (temp) ; }

17. Para reconocer las funciones añadimos un nuevo *no_terminal*, funciones, delante de *iniciar* que servía para reconocer la parte del main

    def_funciones: funciones iniciar

    Con *funciones* ponemos la sintaxis típica de una función con la declaración del nombre, los parámetros entre paréntesis y las sentencias junto con el return dentro de los corchetes. Al final, se incluye *funciones* de nuevo para poder definir varias más funciones.

    funciones: /*lambda*/ { sprintf (temp, "");
    $$ = genera_cadena (temp) ;}

```
| IDENTIF '('parametros')' '{'
                                { sprintf (funcion_actual, "%s", $1);}
                                variables_ambito ';' cuerpo_funcion '}' funciones

        { sprintf (temp, "( defun %s (%s) \n%s \n%s))\n%s",$1,$3,$7,$9,$11);
                                $$ = genera_cadena (temp) ;}
;
```

Intercalamos una acción semántica que nos permite guardar en una variable el nombre de la función que se está definiendo para luego poder escribirlo en el retorno de la función.

El no-terminal *cuerpo_funcion* es utilizado para escribir varias sentencias o varios retornos:

```
cuerpo_funcion: r_sentencia cuerpo_funcion          { sprintf (temp, "%s \n%s",
$1, $2);
                                $$ = genera_cadena (temp) ;}


    | RETURN  expresion retorno_funcion ';' cuerpo_funcion
                            { if (strcmp("", $3) == 0){

                              sprintf (temp, "( return-from %s %s
                              )\n%s",funcion_actual,$2,$5);
                             }else{

                              sprintf (temp, "( return-from %s ( values %s %s
                              ) )\n%s",funcion_actual,$2,$3,$5);
                             }
                              $$ = genera_cadena (temp) ;}
    |/*lambda*/                 { sprintf (temp, "");
                              $$ = genera_cadena (temp) ;}
    ;
```

Utilizamos el no terminal *parametros* para reconocer los parámetros dentro de los paréntesis. Con *r_parametros* permitimos que se escriban más de uno y con la derivación *parametros -> lambda,* contemplamos el caso en que no se introduce ninguno.

```
parametros: INTEGER IDENTIF r_parametros      { if (strcmp("", $3) == 0){

                        sprintf (temp, " %s ",$2);
                        }
                        else{
                           sprintf (temp, " %s %s ",$2, $3);
                        }
```

```
                                $$ = genera_cadena (temp) ;}

        | /*lambda*/                { sprintf (temp, "");
                                $$ = genera_cadena (temp) ;}
        ;

r_parametros: ',' parametros            { sprintf (temp, "%s",$2);
                                $$ = genera_cadena (temp) ;}
        | /*lambda*/                { sprintf (temp, "");
                                $$ = genera_cadena (temp) ;}
```

También incluimos las llamadas a las funciones dentro de otras funciones o del main:

sentencia: llamada_funcion

llamada_funcion: IDENTIF '('parametros_llamada')'
r_parametros_llamada: ',' parametros_llamada

| /*lambda*/

```
parametros_llamada: expresion r_parametros_llamada { if (strcmp("", $2) == 0){

                                sprintf (temp, " %s ",$1);
                                }
                                else{
                                   sprintf (temp, " %s %s ",$1, $2);
                                }
                                 $$ = genera_cadena (temp) ;}

        | /*lambda*/                { sprintf (temp, "");
                                $$ = genera_cadena (temp) ;}
        ;
```

18. Para incluir las asignaciones múltiples añadimos una nueva regla al no terminal *asignacion*.

    asignacion: IDENTIF',' IDENTIF asignacion_multiple expresion ',' expresion

    Añadimos dos identificadores y dos expresiones para diferenciarla de la asignación que ya teníamos antes.

    Diseñamos esta regla en forma de cebolla para obligar a que se escriban el mismo número de terminales a la izquierda y a la derecha.

    asignacion_multiple: '='

```
                                { sprintf (temp, " ) (values ");
                               $$ = genera_cadena (temp) ; }
```

| ',' IDENTIF asignacion_multiple expresion ','

{ sprintf (temp, "%s %s %s",$2, $3, $4);
$$ = genera_cadena (temp) ; }

;

La semántica va construyendo una cadena que se inicia con el =. A partir de ahí, se van añadiendo a la izquierda y derecha identificadores y expresiones hasta llegar a asignación, donde se cierra.

Para los retornos múltiples, modificamos el retorno previamente definido:

cuerpo_funcion: RETURN expresion retorno_funcion ';' cuerpo_funcion

Incluímos el no termial retorno_funcion para permitir devolver varias expresiones:

retorno_funcion: ','expresion retorno_funcion
| /*lambda*/

19. Para los ámbitos, decidimos forzar que las variables del ámbito de la función se declarase en una primera parte de la función y, luego, el resto de las sentencias:

funciones: IDENTIF '('parametros')' '{' { sprintf (funcion_actual, "%s", $1);}
variables_ambito ';' cuerpo_funcion '}' funciones

variables_ambito:INTEGER IDENTIF variables_ambito_asignacion variables_ambito_multiple

{sprintf (temp, "(let ((%s %s) %s",$2,$3,$4);
$$ = genera_cadena (temp) ;
}

;

variables_ambito_asignacion: /*lambda */

{ sprintf (temp, "0");
$$ = genera_cadena (temp);}

| '=' expresion_aritm

{ sprintf (temp, "%s", $2);
$$ = genera_cadena (temp);}

;

variables_ambito_multiple: /*lambda*/

{ sprintf (temp, "");
$$ = genera_cadena (temp) ; }
| ',' IDENTIF variables_ambito_asignacion variables_ambito_multiple

```
                                        { sprintf (temp, "( %s %s ) %s", $2, $3, $4);
                                    $$ = genera_cadena (temp) ; }
        ;
```

Para los ámbitos anidados, añadimos a r_sentencias un nuevo no terminal, ámbito anidado. De la misma manera que en el caso anterior, forzamos la declaración de las variables locales al principio del nuevo ámbito.

r_sentencias: ambito_anidado r_sentencias

ambito_anidado: '{'variables_ambito ';' r_sentencias'}'