Universidad Carlos III

Procesadores del Lenguaje

Práctica Final

Curso 2021-22

Fecha entrega: **22/04/22**

GRUPO: **83**

Alumnos:

Grupo 1 : Mario Resino Solis, Rafael Zhu Zhou

100428975@alumnos.uc3m.es 100429078@alumnos.uc3m.es

Para que el pdf quede más claro, se van a ir explicando todos los cambios realizados en las diferentes secciones (incluyendo sus puntos). Antes de comenzar, comentar que todos los no terminales que se han añadido nuevos a la gramática han sido añadidos en %type <cadena> para poder usar $$ e ir pasando hacia arriba la cadena que queremos imprimir. Además, también se han creado nuevos tokens para las palabras reservadas.

1. Lo primero que se nos piden en el punto uno es sustituir en la gramática el símbolo '#' por el símbolo '$' en la sentencia para imprimir. Esto se debe a que si no se trataría como un include.

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

2. A continuación nos pedían realizar la impresión de múltiples parámetros. A la gramática le añadimos un nuevo no terminal para poder crear dicha recursividad:

```
Impresión: '$' '(' expresion rest_expresion')' {sprintf(temp, "(print %s)%s", $3,$4)

rest_expresion: ',' expresion rest_impresion {sprintf(temp, "(print %s)%s", $2,$3);
                                              $$ = genera_cadena(temp);}
              |                               {$$="";}
              ;
```

3. En este punto se pide implementar las declaraciones globales. Para ello metemos en la gramática lo siguiente:

```
declaracion_global:   INTEGER IDENTIF                    {sprintf(temp, "(setq %s 0)", $2);
                                                          printf("%s",temp);}
                    ;
```

Este nuevo no terminal se llama desde sentencia.

4. En este punto se pide además poder inicializar una variable global (solamente con números). La gramática queda de la siguiente forma:

```
declaracion_global:   INTEGER IDENTIF                       {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                             printf("%s",temp);}
                    | INTEGER IDENTIF '=' NUMERO            {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
                                                             printf("%s",temp);}
```

Todos los ‘;’ que necesitamos para el final de cada sentencia está en el no terminal sentencia.

5. Para poder incluir la función main hemos tenido que añadir un nuevo no terminal func_main con la estructura adecuada. La gramática queda de la siguiente manera:

```
func_main:          MAIN  '(")"{' cuerpo_funciones '}'      {sprintf(temp, "(defun main () %s) ", $6);
                                                             printf("%s",temp);}
                    ;
cuerpo_funciones:       sentencia_func cuerpo_funciones     {sprintf(temp, "%s %s", $1,$2);
                                                             $$ = genera_cadena(temp);}
                    ;
```

Esta funcion main es llamada desde el axioma. Además se crea también el no terminal cuerpo_funciones para poder poner dentro sentencias.

6. Para seguir la estructura indicada, hemos tenido que modificar el axioma con la llamada a estos dos nuevos terminales : decl_variables y decl_funciones. decl_variables va a tener el no terminal decaracion_global de forma recursiva, para que se puedan declarar más de una variable global, y el no terminal decl_funciones va a tener la llamada a los no terminales de funciones (se explica más adelante) y al no terminal func_main. De esta

forma controlamos que solo existe un único main y que esté al final de todas las funciones.

```
axioma:             decl_variables decl_funciones                    {;}
                    ;
decl_variables:                                                      {;}
                    | declaracion_global ';' decl_variables          {;}
                    ;

decl_funciones:       funciones func_main                            {sprintf(temp, "%s%s", $1,$2);
                                                                     $$ = genera_cadena(temp);}
                    |                                                {$$="";}
                    ;
func_main:          MAIN  '(")"{' cuerpo_funciones '}'               {sprintf(temp, "(defun main () %s) ", $6);
                                                                     printf("%s",temp);}
```

7. Para abordar este punto, la gramática es muy parecida a la de declaración_global. En este caso hemos creado un nuevo no terminal llamado declaracion_local que se va a llamar desde sentencia y sentencia_func. Diferenciamos sentencia y sentencia_func debido al RETURN que se nos pide implementar más adelante. La gramática se queda de la siguiente manera:

```
declaracion_local:  INTEGER IDENTIF                              {sprintf(temp, "(setq %s 0)", $2);
                                                                 $$ = genera_cadena(temp);}
                    | INTEGER IDENTIF '=' expresion              {sprintf(temp, "(setq %s %s)", $2,$4);
                                                                 $$ = genera_cadena(temp);}
                    ;
```

8. Este punto se aborda añadiendo tanto en declaracion_local como en

declaracion_global un nuevo terminal en cada caso, rest_decl_local y rest_decl_global respectivamente. La gramática queda de la siguiente forma:

```
declaracion_global:   INTEGER IDENTIF rest_decl_global   {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                          printf("%s",temp);}
                    | INTEGER IDENTIF '=' NUMERO rest_decl_global
                                                         {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
                                                         printf("%s",temp);}
                    ;

rest_decl_global:   ',' IDENTIF rest_decl_global          {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                           $$ = genera_cadena(temp);}
                    | ',' IDENTIF '=' NUMERO rest_decl_global
                                                         {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
                                                          $$ = genera_cadena(temp);}
                    |                                                {$$="";}
                  ;
declaracion_local:  INTEGER IDENTIF rest_decl_local       {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                           $$ = genera_cadena(temp);}
                  |   INTEGER IDENTIF '=' expresion rest_decl_local
                                                         {sprintf(temp, "(setq %s %s)%s", $2,$4,$5);
                                                          $$ = genera_cadena(temp);}
                    ;

rest_decl_local:    ',' IDENTIF '=' expresion rest_decl_local   {sprintf(temp, "(setq %s %s)%s", $2,$4,$5);
                                                                 $$ = genera_cadena(temp);}
                  |   ',' IDENTIF rest_decl_local               {sprintf(temp, "(setq %s 0)%s", $2,$3);
```

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

            |                                       {$$="";}

            ;
```

9. Para este punto simplemente hemos quitado expresion de sentencia
10. Para este punto , se ha implementado una nueva palabra reservada ‘puts’ al cual se le asigna el token PUTS. Dentro de impresión se ha añadido una nueva derivación que permite llevar a cabo la funcionalidad mencionada. La gramática queda de la siguiente manera:

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

           ;
```

11. Para este punto, se ha implementado una nueva palabra reservada ‘printf’ al cual se le asigna el token PRINTF. Dentro de impresión se ha añadido una nueva derivación que permite llevar a cabo la funcionalidad. La gramática queda de la siguiente manera:

```
impresion:  PRINTF '(' STRING rest_impresion ')'    {$$=$4;}

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

           ;

rest_impresion: ',' expresion rest_impresion        {sprintf(temp, "(print %s)%s", $2,$3);

                                                     $$ = genera_cadena(temp);}

            |                                       {$$="";}

            ;
```

Como se puede observar, la sentencia tiene STRING para ignorar los string que se introduzcan al principio de la sentencia, para omitir cosas como %d.

12. Para este punto lo primero que hemos hecho ha sido crear un token para cada operador lógico y de comparación y los hemos añadido en la sección de palabras

reservadas para su reconocimiento. Una vez hecho esto, hemos añadido su precedencia usando la declaración de bison de %left, quedando así las declaraciones de bison:

```
%right '='                  // es la ultima operacion que se debe realizar
%left OR
%left AND
%left NOTEQ EQ
%left '<' '>' LESSEQ GREATEREQ
%left '+' '-'               // menor orden de precedencia
%left '*' '/' '%'           // orden de precedencia intermedio
%left SIGNO_UNARIO          // mayor orden de precedencia
```

Estos operadores lógicos y comparadores los hemos metido en expresion, quedando la gramática de la siguiente manera:

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

```
| expresion NOTEQ expresion          {sprintf(temp, "(/= %s %s)", $1,$3);
                                     $$ = genera_cadena(temp);}
| expresion EQ expresion             {sprintf(temp, "(= %s %s)", $1,$3);
                                      $$ = 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 LESSEQ expresion        {sprintf(temp, "(<= %s %s)", $1,$3);
                                     $$ = genera_cadena(temp);}
| expresion GREATEREQ expresion     {sprintf(temp, "(>= %s %s)", $1,$3);
                                     $$ = genera_cadena(temp);}
| expresion '%' expresion           {sprintf(temp, "(mod %s %s)", $1,$3);
                                     $$ = genera_cadena(temp);}
        ;
```

13. Para esta función hemos añadido un nuevo no terminal llamado bucles. Este nuevo no terminal se llama desde sentencia y desde sentencia_func. La gramática de bucle queda de la siguiente forma:

```
bucles: WHILE '('expresion')"{' cuerpo '}'      {sprintf(temp, "(loop while %s do %s)",$3,$6)
                                                 $$ = genera_cadena(temp);}
        ;
```

Como se menciona en la práctica, bucle no necesita de ‘;’ al final y se llama tanto en sentencia como en sentencia_func.

14. Para abordar este punto hemos creado dos nuevos no terminales, condicionales para añadir la funcionalidad y cuerpo_cond, por si un if tiene más de dos sentencias dentro y así poder añadir progn. La sentencia queda

de la siguiente manera:

```
condicionales: IF '('expresion')''{' cuerpo_cond '}'

                                          {sprintf(temp, "(if %s %s)",$3,$6);

                                           $$ = genera_cadena(temp);}

             | IF '('expresion')''{' cuerpo_cond '}' ELSE'{' cuerpo_cond '}'
                                        {sprintf(temp, "(if %s %s %s)",$3,$6,$10);

                                         $$ = genera_cadena(temp);}

              ;


cuerpo_cond: sentencia                                          {$$=$1;}

            |sentencia sentencia cuerpo

                                      {sprintf(temp, "(progn %s %s %s)",$1,$2,$3);

                                       $$ = genera_cadena(temp);}


            |                                                   {$$="";}

            ;
```

Los if tampoco necesitan al final el símbolo ‘;’ y se llama tanto en sentencia como en sentencia_func.

15. Para el bucle for, se ha usado la misma filosofía que en el bucle while a la hora de aplicar la semántica (como se explica en la práctica). Se crea un nuevo no terminal llamado bucle_for y la gramática queda de la siguiente forma:

```
bucles_for:  FOR '(' declaracion_local ';' expresion ';' asignacion ')"{' cuerpo '}'

                                        {sprintf(temp, "%s\n(loop while %s do %s%s)",$3,$5,$10,$7);
                                         $$ = genera_cadena(temp);
            ;
```

Al igual que el bucle while no requiere de ‘;’ al final y se llama tanto en sentencia como en sentencia_func.

16. En este punto se pide implementar la creación de vectores y su utilización en operaciones. Para la implementación de la creación simplemente se ha introducido su traducción:

```
declaracion_global:   INTEGER IDENTIF rest_decl_global   {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                          printf("%s",temp);}
                    | INTEGER IDENTIF '=' NUMERO rest_decl_global
                                                         {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
                                                         printf("%s",temp);}
                    | INTEGER IDENTIF '[' NUMERO ']' rest_decl_global
                                                  {sprintf(temp, "(setq %s (make-array %d))%s", $2,$4,$6);
                                                  printf("%s",temp);}
                  ;
rest_decl_global:   ',' IDENTIF rest_decl_global                  {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                                   $$ = genera_cadena(temp);}
                    | ',' IDENTIF '=' NUMERO rest_decl_global
                                                         {sprintf(temp, "(setq %s %d)%s", $2,$4,$5);
                                                          $$ = genera_cadena(temp);}
                    | ',' IDENTIF '[' NUMERO ']' rest_decl_global
                                                    {sprintf(temp, "(setq %s (make-array %d))%s", $2,$4,$6);
                                                    $$ = genera_cadena(temp);}
                     |                                                          {$$="";}
                  ;
declaracion_local:  INTEGER IDENTIF rest_decl_local          {sprintf(temp, "(setq %s 0)%s", $2,$3);
                                                             $$ = genera_cadena(temp);}
                  |   INTEGER IDENTIF '=' expresion rest_decl_local
                                                         {sprintf(temp, "(setq %s %s)%s", $2,$4,$5);
```

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

                |INTEGER IDENTIF '[' NUMERO ']' rest_decl_local

                                                {sprintf(temp, "(setq %s (make-array %d))%s", $2,$4,$6);

                                                 $$ = genera_cadena(temp);}

            ;


                    ;


rest_decl_local:   ',' IDENTIF '=' expresion rest_decl_local     {sprintf(temp, "(setq %s %s)%s", $2,$4,$5);

                                                                 $$ = genera_cadena(temp);}

                |   ',' IDENTIF rest_decl_local                   {sprintf(temp, "(setq %s 0)%s", $2,$3);

                                                                  $$ = genera_cadena(temp);}

                | ',' IDENTIF '[' NUMERO ']' rest_decl_local

                                        {sprintf(temp, "(setq %s (make-array %s))%s", $2,int_to_string($4),$6);

                                        $$ = genera_cadena(temp);}

                |                                                  {$$="";}

                ;
```

También se pide poder acceder a su valor, en el enunciado pone como ejemplo el uso de un printf, sin embargo nosotros hemos permitido en cualquier sitio donde se pueda poner un número o variable poder poner también un vector con su índice. De esta forma se modifica únicamente el no terminal operando:

```
operando: IDENTIF                       {$$ = genera_cadena($1);}
          | IDENTIF'['expresion']'       {sprintf(temp, "(aref %s %s)",$1, $3);
                                         $$ = genera_cadena(temp);}
          | NUMERO                      {$$ = int_to_string($1);}
          | '(' expresion ')'            {$$ = $2;}
          ;
```

Como último subapartado se pide poder cambiar los valores de los

elementos del vector, para ello en asignación se ha introducido el cambio de los elementos del vector:

```
asignacion: IDENTIF '=' expresion                    {sprintf(temp, "(setq %s %s)", $1,$3);
                                                      $$ = genera_cadena(temp);}
          | IDENTIF '[' expresion ']'"=' expresion
                                                     {sprintf(temp, "(setf (aref %s %s) %s)",$1,$3, $6);
                                                      $$ = genera_cadena(temp);}
;
```

17. Para realizar este punto, se pide realizar la traducción de operadores ternarios. No existe como tal una función en Lisp que lo haga así que se ha simulado usando las declaraciones ifs-then-else. De esta forma, dentro del no terminal asignacion añadimos una nueva derivación:

```
asignacion: IDENTIF '=' expresion                          {sprintf(temp, "(setq %s %s)", $1,$3);
                                                            $$ = genera_cadena(temp);}

          | IDENTIF '[' expresion ']'"=' expresion
                                                          {sprintf(temp, "(setf (aref %s %s) %s)", $1,$3, $6);
                                                           $$ = genera_cadena(temp);}

          | IDENTIF '=' expresion '?' operando ':' operando
                                                  {sprintf(temp, "(setq %s (if %s %s %s))",$1, $3, $5, $7);
                                                   $$ = genera_cadena(temp);}

          | IDENTIF '[' expresion ']' '=' expresion '?' operando ':' operando
                              {sprintf(temp,"(setf (aref %s %s) (if %s  %s %s))",$1, $3, $6, $8, $10);
                              $$ = genera_cadena(temp);}

           ;
```

18. La primera implementación a las funciones es permitir argumentos, tanto opcionales como obligatorios. Para esto, tuvimos que investigar por internet cómo incluir los argumentos. Una vez aprendido únicamente se tradujo con la forma apropiada quedando así la siguiente gramática:

```
argumentos: rest_argumentos                {$$=$1;}
          |                                {$$="";}
;

rest_argumentos: INTEGER IDENTIF                        {$$=genera_cadena($2);}
          | INTEGER IDENTIF ',' rest_argumentos          {sprintf(temp,"%s %s",$2,$4);
                                                         $$=genera_cadena(temp);}
```

```
            | rest_arg_opc                                    {sprintf(temp,"&optional %s",$1);
                                                              $$ = genera_cadena(temp);}
;

rest_arg_opc: INTEGER IDENTIF '=' NUMERO                      {sprintf(temp,"(%s %d)",$2,$4);
                                                              $$=genera_cadena(temp);}
| INTEGER IDENTIF '=' NUMERO ',' rest_arg_opc                 {sprintf(temp,"(%s %d)%s",$2,$4,$6);
                                                              $$=genera_cadena(temp);}
;
```

Tras esto, se pide implementar los retornos en las funciones, para ello tuvimos que duplicar los no terminales cuerpo y sentencia, creando así cuerpo_func y sentencia_func. Esto nos permitió diferenciar los retornos finales de los retornos intermedios.

```
cuerpo_funciones: sentencia_func cuerpo_funciones             {sprintf(temp, "%s %s", $1,$2);
                                                              $$ = genera_cadena(temp);}

                | RETURN expresion ';'                        {$$=$2;}
                |                                             {$$="";}
;
sentencia_func: asignacion ';'                                {$$=$1;}
              | impresion ';'                                 {$$=$1;}
              | declaracion_local ';'                         {$$=$1;}
              | bucles                                        {$$=$1;}
              | condicionales                                 {$$=$1;}
              | bucles_for                                    {$$=$1;}
              | llamada_func ';'                              {$$=$1;}
              | RETURN expresion ';' sentencia_func
                                {sprintf(temp, "(return-from %s %s) %s",nombres_funciones, $2,$4);
                                 $$ = genera_cadena(temp);}
;
sentencia: asignacion ';'                                     {$$=$1;}
         | impresion ';'                                      {$$=$1;}
         | declaracion_local ';'                              {$$=$1;}
         | bucles                                             {$$=$1;}
         | condicionales                                      {$$=$1;}
         | bucles_for                                         {$$=$1;}
         | llamada_func ';'                                   {$$=$1;}
         | RETURN expresion ';'          {sprintf(temp, "(return-from %s %s)",nombres_funciones,$2);
                                          $$ = genera_cadena(temp);}
;
```

Por último, falta implementar las llamadas a las funciones, que pueden aparecer en cualquier sitio donde pudiese aparecer un número además de aparecer como una simple sentencia.

```
llamada_func: IDENTIF '(' parametros ')'          {sprintf(temp,"(%s %s)",$1,$3);
                                                   $$=genera_cadena(temp);}
;
```

```
parametros: rest_parametros                    {$$=$1;}
          |                                    {$$="";}
;

rest_parametros: expresion                     {$$=$1;}

               | expresion ',' rest_parametros  {sprintf(temp,"%s %s",$1,$3);
                                                $$=genera_cadena(temp);}

;
```