UNIVERSIDAD CARLOS III DE MADRID

GRADO EN INGENIERÍA INFORMÁTICA

PROCESADORES DEL LENGUAJE

Traductor de C a Forth

Autores
ALBERTO VILLANUEVA NIETO 100374691
CRISTIAN CABRERA PINTO 100363778

May 7, 2019

Contents

1 Introducción

La ejecución de dicha práctica consiste en la elaboración de un tradcutor de lenguaje C a lenguaje Forth, para lo cuál partiendo de una grámatica hemos ido modificandola para adaptarla a los diferentes puntos pedidos. En este documento se presenta como se ha progresado en cada punto hasta tener la versión final del código, puesto que durante los primeros puntos hasta llegar al apartado donde hay que implementar el FOR todo había sido hecho sin necesidad de código diferido.

2 Apartados Implementados

4. Variables Locales

Para definir las variables locales de la función *main*, hemos decido poner la semántica de imprimir el nombre de la función después de la parte de la definición de variables y en la semántica de definición de variables se imprimen estas.

```
principal:
        MAIN '(' ')' '{' def_var        { printf (": main \n"); }
                codigo '}'              { printf (";\n"); }
;
```

```
def_var:
        /* lambda */                    { ; }
        | INTEGER IDENTIF ';'           { printf("variable %s \n", $2);}
                def_var
;
```

5. Estructura de control do-while

Para definir el bucle do-while se ha creado una producción en la regla `sentencia` la cual escribe *begin* cuando lee el do, a continuación el no terminal *codigo* escribe el código, después el no terminal *expresion* escribe la condición del bucle y por último se escribe *while repeat*.

```
|DO                                                 { printf("begin\n"); }
        '{' codigo '}' WHILE '(' expresion ')' ';'  { printf("while repeat \n"); }
```

6. Operadores logicos

Para los operadores de más de un caracter se han añadido tokens que permitirán reconocer esos operadores ya que en la estructura de palabras reservadas que se utiliza en el *yylex* se ha añadido la definición de cada caracter y el token que lleva asociado para que cuando el *yylex* detecte el operador pueda devolver el token y la gramática lo pueda reconocer. También se han establecido las precedencias necesarias para evitar que se produzcan conflictos de *shift* y *reduce*. Hay dos tipos de operadores que hemos tenido que añadir, los operadores binarios para los que se ha añadido

una producción a la regla `expresion` de la forma: *expresion operador expresion*, y para la impresión se deja que los no terminales *expresion* impriman las expresiones y luego se imprime el operador correspondiente.

```
| expresion EQUAL expresion          { printf ("= ") ; }
| expresion UNEQUAL expresion        { printf ("= 0= ") ; }
| expresion LESSOREQ expresion       { printf ("<= ") ; }
| expresion '<' expresion            { printf ("< ") ; }
| expresion '>' expresion            { printf ("> ") ; }
| expresion '&' expresion            { printf ("and ") ; }
| expresion '|' expresion            { printf ("or ") ; }
| expresion AND expresion            { printf ("and ") ; }
| expresion OR expresion             { printf ("or ") ; }
| expresion '%' expresion            { printf ("mod ") ; }
```

Los otros operadores son de tipo unario, para estos se ha añadido una producción en la regla `termino` donde se ha dejado que el no terminal *operando* imprima el operando y luego se imprime el operador unario correspondiente.

```
| '!' operando %prec SIGNO_UNARIO          { printf ("0= ") ; }
| operando ADDER %prec POSTFIX              { printf ("1+ ") ; }
| operando SUBSTRACTER %prec POSTFIX        { printf ("1- ") ; }
```

7. Estructura de control If

Para incluir la estructura de control if se ha añadido una producción en la regla `sentencia` donde se deja que se imprima la condición mediante el no terminal *expresion* y después se imprime *if*. Luego se deja que se imprima el código y el resto del if, que puede ser o nada o un *else* que iría seguido de su código correspondiente, y finalmente se imprime el *t*hen para indicar la finalización del código.

```
| IF '(' expresion ')'                    {printf("if\n");}
      '{' codigo '}' restoIf                {printf("then\n");}
```

```
restoIf:
      /* lambda */
      | ELSE                                        {printf("else\n");}
            '{' codigo '}'
;
```

9. Impresión de cadenas mediante puts

Para la realización de este punto se ha añadido un token para identificar la función (también se ha añadido a la estrutura de palabras reservadas la definición y el token generado para que el *yylex* pueda reconocerlo) y una producción a la regla `sentencia` que permite deteminar que hacer

cuando se encuentra con la función *puts* que se encarga de imprimir por pantalla un texto. Para traducirlo a forth lo que se ha hecho es imprimir en la semántica un punto al principio y después entre comillas el token identificado como STRING que es el texto que ha reconocido el *yylex*.

```
| PUTS '(' STRING ')' ';'        { printf(". \" %s\" \n" ,$3); } ;
```

10. Imprimir expresiones (printf)

Para conseguir imprimir expresiones lo que se ha hecho es añadir un token que permita identificar la función (también se ha añadido a la estrutura de palabras reservadas la definición y el token generado para que el *yylex* pueda reconocerlo) y una producción a la regla `sentencia` donde no se imprime nada sino que es la regla `expresiones` la cual está formada por dos producciones donde el no terminal *expresion* es el encargado de imprimir la exprsion y después se imprime un punto que indica que se tiene que imprimir el valor de la expresión.

```
| PRINTF '(' STRING ',' expresiones ')' ';' ;
```

```
expresiones:
        | expresion                              { printf(" .\n"); }
        | expresion                              { printf(" . "); }
              ',' expresiones
;
```

La regla `expresiones` se usa para permitir que a la función *printf* se le puedan pasar una o más expresiones que imprimir.

11. Operador ternario

Para el funcionamiento de este punto hemos agregado una producción a la regla `expresion` en la cual el primer no terminal *expresion* se encarga de imprimir la expresión correspondiente, después cuando el operador ternario es leído se imprime *if*, el siguiente no terminal *expresion* otra expresión, después al leer los dos puntos se imprime *else* tras el cual el último no terminal de *expresion* imprime su expresión y para cerra se imprime *then*.

```
| expresion '?'              {printf("if\n");}
      expresion ':'          {printf("\nelse\n");}
      expresion              {printf("\nthen\n");}
```

Además para evitar conflictos de *shit/reduce* se ha tenido que añadir precedencia al operando '?'.

12. Adición de vectores

Para tratar los vectores tenemos que tener en cuenta 2 aspectos el primero de ellos es la declaración de variables para lo cual a la producción de la regla `def_var` se ha añadido un no terminal después

de *variable* y del identificador que es *restoDef_var* cuya regla es la encargada de imprimir lo necesario para reservar suficiente memoria para el vector.

```
def_var:
        /* lambda */                    { ; }
        | INTEGER IDENTIF               { printf ("variable %s ", $2);}
                restoDef_var';' def_var
;
restoDef_var:
        /* lambda */                    { printf("\n"); }
        |'[' expresion ']'              { printf("1 - cells allot\n");}
;
```

El segundo aspecto que hemos tenido que tratar para la inclusión de vectores en el traductor ha sido la asignación de una variable tipo vector a una expresión. Para ello primero hemos creado una regla nueva que contiene todas las asignaciones y la hemos denominado `asignacion`. En esa regla hemos añadido tanto la asignación para variable de tipo entero como la asignación de variables tipo vector. Y lo que se hace es imprimir en primer lugar la expresión que se corresponde a lo de dentro del corchete del vector (de ello se encarga el no terminal *expresion*), después se imprime la expresión de la derecha del igual (de lo que también se encarga el no terminal *expresion*) y por último como tenemos que intercambiar los valores anteriores en la pila se escribe *swap cells identificador + !* para que este primero en la pila la posicion del vector, eso se convierte en celdas y se le suma al vector para desplazarse el número de posiciones que se pide y al estar ya en esa posicion se asigna el valor.

```
asignacion:
        IDENTIF '=' expresion                           {printf ("%s !\n", $1);}
        | IDENTIF '[' expresion ']' '=' expresion       {printf ("swap cells %s + !\n", $1}
;
```

Para llevar a cabo la asignación se ha agregado una produccion a la regla `sentencia` que permite que en el codigo haya asignaciones. También para que se cumpla que se produce una asignación hay que tener en cuenta que el acceder a una posicion del vector nos deposita un valor en la pila luego hay que tratarlo como un operando por lo que haria falta incluir en la regla `operando` una producción que permita identificar que esa expresion es un vector y cuya semántica imprime, después de la impresión del no terminal *expresion*, *cells identificador* + @ para acceder al valor de esa posición determinada del vector.

```
sentencia:
        asignacion';'
        ...
;
operando:
        IDENTIF                         { printf ("%s @ ", $1) ; }
        | IDENTIF '[' expresion ']'     { printf ("cells %s + @ ", $1) ; }
;
```

13. Estructura de control FOR

Para definir esta estructura de control hemos tenido que pasar las asignaciones a codigo diferido para que se pueda imprimir la tercera asignación despues de que el código se imprima. Para ello hemos tenido que pasar expresion, termino y operando a codigo diferido para que todo de lo que depende asignación también este en codigo diferido.
Lo primero que hemos hecho ha sido pasar de usar Union a usar una estructura para poder hacer referencia al atributo en el que se guardara, para ello definimos la estructura t_atributos y definimos YYSTYPE como esa estructura. Además también tuvimos que quitar <cadena> y <valor> de los tokens.

```
typedef struct s_atributos {
        int valor;
        char *cadena ;
} t_atributos ;
#define YYSTYPE t_atributos
```

En la semantica de cada producción en diferido se ha usado una variable temporal temp y con sprintf y genera_cadena para escribir en $$.cadena. También, se han realizado los cambios necesario para que en todas las producciones en las que se usaba expresión se imprima desde ahi.
De esta forma, finalmente el for quedaria definido asi añadido como producción de la regla sentencia:

```
| FOR '(' asignacion ';' expresion ';'    { printf("%sbegin %swhile\n",$3.cadena,$5.cadena); }
      asignacion ')' '{' codigo '}'         { printf("%srepeat\n",$8.cadena); }
```

14. Variables tipo matriz

Para las matrices se ha añadido la variable t_simbolos_matrices que es una tabla de simbolos para almacenar el nombre de la matriz y el el tamaño de la segunda dimension. Para la definicion de las matrices se ha añadido la regla matrix que se añade al final de un vector y en la semantica de la definicion de la variable se ha hecho la creación de la matriz en la tabla de simbolos dependiendo de si esa variable es una matriz, esto ultimo se ha hecho con la variable global isMatrix.

```
def_var:
    /* lambda */              { ; }
    | INTEGER IDENTIF         { printf ("variable %s ", $2.cadena);}
          restoDef_var';'     { if(isMatrix)crearMatriz($1.cadena); }
          def_var
;
restoDef_var:
    /* lambda */              { printf("\n"); }
    |'[' expresion ']'        { printf("%s",$2.cadena);}
          matrix
;
matrix:
    /*lambda*/                { printf("1 - cells allot\n"); isMatrix=0; }
    | '[' expresion ']'       {
                                  isMatrix=1;
                                  t_simbolos_matrices[num_matrices].expresion=
                                      genera_cadena($2.cadena);
                                  printf("%s* cells allot\n",$2.cadena);
                              }
;
```

Para la asignación usamos la tabla de simbolos para hacer el desplacamiento adecuado asi que añadimos la siguiente producción a la regla asignación.

```
| IDENTIF '[' expresion ']' '[' expresion ']' '=' expresion
      {
          i = findMatriz($1.cadena);
          sprintf(temp, "%s%s%s* %s+ cells %s + !\n", $9.cadena, $3.cadena,
              t_simbolos_matrices[i].expresion, $6.cadena, $1.cadena);
          $$.cadena=genera_cadena(temp);
      }
```

Para usarlo como termino, es muy parecido a la asignación ya que solo se tiene que quitar en la producción = termino y en la semantica se quita el primer %s para no añadir la expresion despues del = y se quita ! ya que ya no estas haciendo una asignación.

15. Funciones

Renombramiento de variables

Para las funciones, lo primero que se ha hecho es que el nombre de las variables sea nombreDeFuncion_nombreDeVariable. Para ello se ha usado la variable global para guardar el nombre de la funcion en la que se esta junto con una barra baja (en la funcion factorial, la variable funcion tendria "factorial_"). En cuanto se empezaba una funcion se le ponia ese nombre y en cuanto se terminaba se ponia el string vacío, y en el yylex se ponia el nombre de la variable como una concatenacion del nombre de la funcion y el nombre de la variable.
Ademas de esto, se han hecho otras dos tablas de simbolos, una para las variables globales y otra

para las variables locales. Ambas se usan en el yylex a la hora de crear las variables. La de las variables globales se usa para poder llamar a una variable global desde dentro de una funcion y que reconozca que esa variable es global, ademas se ha usado un campo para saber si existe una variable global y otra local con el mismo nombre para que en ese caso use el nombre de la variable local. La de las variables locales se usa para que en las funciones recursivas se guarden el estado actual en la pila de retorno y despues se recojan.

Definicion de funciones

Para la definicion de las funciones se ha puesto en definicion de variables la funcion void, y la funcion int se ha factorizado junto con la variable int. Para la función se imprimen primero los argumentos y las variables locales, que son definiciones de variables locales, luego se imprime : y el nombre de la funcion y luego se recogen los argumentos de la pila. Después de esto se imprime el codigo que lo imprime el no terminal codigo_funcion que puede ser vacio o codigo para que pueda haber funciones con solo return. Y finalmente si es de tipo int, se tiene return expresión que imprime la expresion para guardarla en pila.

```
def_var:
    /* lambda */{ ; }
    | INTEGER { declarando = 1; }
       restoVariable_funcion def_var
    |VOID IDENTIF                          {
                                                if(funcion[0] != 0){
                                                   perror("No se permite funciones
                                                       dentro de funciones\n");
                                                   exit(-1);
                                                   }
                                                sprintf(funcion,"%s_",$2.cadena);
                                           }
       '(' argumentos ')' '{' def_var      {
                                                printf (": %s\n",$2.cadena);
                                                printArgumentos();
                                           }
       codigo '}'                          {
                                                printf (";\n");
                                                salirFuncion();
                                           }
       def_var
;
```

```
restoVariable_funcion:
    IDENTIF                                 {
                                                printf ("variable %s\n",$1.cadena);
                                                declarando = 0;
                                            }
    restoDef_var ';'                        { if(isMatrix)crearMatriz($1.cadena); }
|IDENTIF '('                                {
                                                if(funcion[0] != 0){
                                                    perror("No se permite funciones
                                                        dentro de funciones\n");
                                                    exit(-1);
                                            }
                                                sprintf(funcion,"%s_",$1.cadena);
                                                declarando = 0;
                                            }
        argumentos ')' '{' def_var          {
                                                printf (": %s\n",$1.cadena);
                                                printArgumentos();
                                            }
codigo_funcion RETURN expresion ';' '}' {
                                                printf ("\n%s\n;\n",$11.cadena);
                                                salirFuncion();
                                            }
;
```

```
codigo_funcion:
    /*lambda*/
    |codigo
;
```

Llamadas a funciones

Para las llamadas a funciones se tienen que añadir producciones en sentencia (para las llamadas a funciones de tipo void) y en operando (para llamadas a funciones tipo int), las dos son iguales, solo cambia que en la de operandoº se hace con codigo en diferido.
Lo primero que se hace, es que los argumentos se guardan en pila, de esto se encarga el no terminal funcion_args, despues se mira si la funcion es una llamada recursiva, si lo es, se guardan todas las variables locales en la pila de retorno se imprime recurse y luego se recuperan todas las variables locales, y si no es una funcion recursiva, simplemente se escribe el nombre de la funcion para hacer la llamada.

```
| IDENTIF '(' funcion_args ')' ';'          {
                                                strcpy(temp,$1.cadena);
                                                strcat(temp,"_");
                                                if(strcmp(funcion,temp)==0){
                                                    printf("%s\n",$3.cadena);
                                                    for(i=0 ; i<64; i++){
                                                        if(var_locales[i] != NULL){
                                                            printf("%s @ >r\n",var_locales[i]);
                                                        }
                                                    }
                                                    printf("recurse\n");
                                                    for(i=63 ; i>=0; i–){
                                                        if(var_locales[i] != NULL){
                                                            printf("r> %s !\n",var_locales[i]);
                                                        }
                                                    }
                                                }else{
                                                    printf("%s\n%s ",$3.cadena,$1.cadena);
                                                }
                                            }
```

3 Cambios en los ficheros de pruebas

Los únicos ficheros de pruebas que han tenido que ser modificados han sido factorial.c y potencias.c ya que incluian un comentario que usaba variables las cuales no se corresponden con las variables que utliza Forth de modo que hemos tenido que cambiar el comentario y poner el nombre de la variable correspondiente.
Además se ha añadido un fichero de pruebas avanzadas para ver el funcionamiento de las llamadas recursivas a funciones ya que este no venía incluido en el directorio de pruebas proporcionado en aula global.

4 Conclusión

La realización de esta práctica, junto con las prácticas guiadas que se han llevado a cabo antes de la práctica final, no han servido para conocer otro lenguaje que desconociamos y para asentar los conocimientos de la asignatura, sobre todo en temas del uso de semántica en diferido. También nos ha permito conocer otras herramientas como es bison.
Durante la realización de la práctica nos hemos encontrado con varios problemas. Entre ellos el más frecuente ha sido la aparición de conflictos *shift/reduce* que surgian principalmente por el tema de precedencias, lo cual también nos ha costado entender en un principio. Otro problema grande que nos encontramos fue al llegar a la parte del FOR ya que es ahí donde hay que llevar a cabo la parte de diferido, ya que en un principio tratamos de hacerlo a través de *union* pero finalmente tuvimos que quitarlo y crear una estructura para los atributos de los tokens. Y el último gran problema que nos hemos encontrado ha sido conseguir que las funciones recursivas llegasen a

funcionar puesto que no connociamos la existencia de la pila de retorno ni la función que hacia el *recurse*.