Universidad Carlos III de Madrid

Procesadores del Lenguaje

Grupo 83

Traductor de C a GForth

Autores:

Daniel Medina García: 100316850@alumnos.uc3m.es
Diego Vicente Martín: 100317150@alumnos.uc3m.es

4 de mayo de 2016

Índice

1. Descripción de los Ejercicios

A continuación, describimos el razonamiento seguido para resolver cada uno de los ejercicos propuestos en el enunciado. Para evitar añadir fragmentos de código relacionados con ejercicios aún no explicados, nos saltaremos ciertas partes del código fuente que no son relevantes usando la notación `[...]` para delimitar las partes omitidas.

1.1. Variables Locales

Ya que las definiciones de `Forth` deben hacerse antes de que se declare el método principal (`main()`), necesitamos una forma de realizar las declaraciones de variables que suceden dentro de éste método antes de declararlo en `Forth`. Para hacer esto, la solución que encontramos al problema fue posponer en cierta manera la acción semántica que declara el método:

```
program:            vars [...] main         { ; }
                ;
[...]

main:               MAIN '(' ')' '{'        { sprintf (funcion_actual, "main"); [...] }
                    vars codigo '}'         { [...] }
                ;

vars:           /* no variable declaration */
                {
                    if(strcmp(funcion_actual, "global")) {
                        printf (": %s \n", funcion_actual);

                    }
                }
                | INTEGER IDENTIF ';'       { [...] }
                    vars
                | [...]
                ;
```

De esta manera, podemos ver cómo se aceptan declaraciones de variables tanto globales (antes de la declaración del main) como locales. En caso de que se termine de reconocer líneas de código, se pasa a imprimir la sintaxis de `Forth` que indica el inicio de un método antes para que éstas formen parte del método. De momento sólo se contempla la existencia de un método main. La mecánica exacta de esa comprobación se verá más detenidamente en futuros ejercicios.

1.2. Do-While

Para introducir el reconocimiento de bucles do-while, tan solo necesitamos hacer una ligera modificación en el bucle while, cuyo código viene dado con el código inicial:

```
codigo:      /* lambda */                          { ; }
           | sentencia ';' codigo                  { ; }
           | flujo codigo                          { ; }
           ;

flujo:     [...]
           |   DO                                  { printf("begin\n") ; FF; }
                   '{' codigo '}'
                   WHILE expresion ';'             { printf("%s while repeat\n", $7) ; FF; }
           [...]
           ;
```

Puede observarse la distinción entre los no terminales `sentencia` y `flujo`. Ambos son líneas de código, pero esta distinción se hizo necesaria con la introducción que quisimos hacer para los bucles `for` que requerían sentencias únicas en su definición.

Aquí podemos ver que al reconocer el token "do", se traducirá el a `Forth` la declaración del bloque de código asociado al bucle. Junto con las palabras reservadas de `Forth`, se imprimirán los saltos de línea necesarios para la legibilidad del código generado. Una vez se reconozca la última parte (el token "while" y la condición asociada al bloque), se traducirá la última parte del bucle en `Forth`. Por último, se incluye una llamada recursiva al no-terminal `codigo`, para seguir reconociendo el resto de la entrada.

1.3. Operadores Lógicos & de Comparación

Para ser capaces de implementar los operadores lógicos, debemos añadir múltiples reglas de producción con sus correspondientes efectos secundarios. Debemos, de igual manera, tener en cuenta las distintas naturalezas de los operadores: podemos añadir los operadores simples como literales que reconocer, pero esto causará conflictos con los operadores compuestos por más de un caracter. Para solucionar este problema, añadiremos los operadores de más de un caracter a la lista de palabras reservadas.

```
t_reservada pal_reservadas [] = { // define las palabras reservadas y los
    {"main",        MAIN},          // y los token asociados
    [...]
    {"++",          INC},
    {"--",          DEC},
    {"==",          EQUALS},
    {"!=",          NOTEQUALS},
    {">=",          GE},
    {"<=",          LE},
    {"||",          OR},
    {"&&",          AND},
    [...]
} ;
```

Con esto, tan solo debemos añadir las nuevas reglas que hagan que se acepten las sentencias con dichos caracteres, asociándolos a su respectivos operadores en `Forth`:

```
expresion:      termino                         { sprintf($$, "%s", $1) }
        |   expresion '+' expresion             { sprintf($$, "%s %s +", $1, $3); }
        |   expresion '-' expresion             { sprintf($$, "%s %s -", $1, $3); }
        |   expresion '*' expresion             { sprintf($$, "%s %s *", $1, $3); }
        |   expresion '/' expresion             { sprintf($$, "%s %s /", $1, $3); }
        |   expresion '&' expresion             { sprintf($$, "%s %s and", $1, $3); }
        |   expresion '|' expresion             { sprintf($$, "%s %s or", $1, $3); }
        |   expresion '%' expresion             { sprintf($$, "%s %s mod", $1, $3); }
        |   expresion '>' expresion             { sprintf($$, "%s %s >", $1, $3); }
        |   expresion '<' expresion             { sprintf($$, "%s %s <", $1, $3); }
        |   IDENTIF INC                         { sprintf($$, "%s @", [...]($1)); }
        |   IDENTIF DEC                         { sprintf($$, "%s @", [...]($1)); }
        |   INC IDENTIF                         { sprintf($$, "%s @ 1 +", [...]($2)); }
        |   DEC IDENTIF                         { sprintf($$, "%s @ 1 -", [...]($2)); }
        |   '!' expresion                       { sprintf($$, "%s 0=", $2); }
        |   expresion EQUALS expresion          { sprintf($$, "%s %s =", $1, $3); }
        |   expresion NOTEQUALS expresion       { sprintf($$, "%s %s = 0=", $1, $3); }
        |   expresion LE expresion              { sprintf($$, "%s %s <=", $1, $3); }
        |   expresion GE expresion              { sprintf($$, "%s %s >=", $1, $3); }
        |   expresion AND expresion             { sprintf($$, "%s %s and", $1, $3); }
        |   expresion OR expresion              { sprintf($$, "%s %s or", $1, $3); }
        ;
```

Destacamos aquí que la expresión no se imprime directamente como hacía el código inicial, sino que se pasa internamente como cadena de caracteres al nivel superior del árbol de derivación. Esto permite una mayor flexibilidad, legibilidad y facilita la depuración del código.

Sin embargo, si intentamos compilar con bison este código, obtendremos un número de conflictos shift/reduce. Para solucionarlos, debemos añadir a bison de forma explícita el orden de precedencia los operadores y la asociatividad. Para ello, escogimos la opción de añadir las siguientes reglas al archivo tras la declaración de tokens en lugar de indicarlo explícitamente en cada línea de la sección 2:

```
%right '='                      // menor orden de precedencia
%left OR
%left AND
%left '|'
%left '&'
%left NOTEQUALS EQUALS
%left GE LE '<' '>'
%left '+' '-'
%left '*' '/' '%'
%left INC DEC '!'
%left SIGNO_UNARIO              // mayor orden de precedencia
```

Con todo esto, ya tenemos listo el reconocimiento de operadores, tanto simples como compuestos

1.4. Estructura If-Else

Para añadir la estructura de control lógico if-else, se han declarado las siguientes reglas:

```
flujo:      [...]
            |   IF expresion                    { printf("%s if\n", $2) ; FF; }
                    '{' codigo '}'
                    else_block
            [...]
            ;

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

De esta forma, podemos asegurarnos de que si se detecta la palabra reservada "if" en alguna línea de código, se usará dicha regla de producción. Como conidición, se acepta tanto una expresión aritmética como de comparación lógica, así como operandos independientes (e.g. `if (0)`). Después de un bloque de código, se espera el no-terminal `else_block`: puede estar vacío (en caso de que el "if" que se está evaluando no tenga una cláusula de else) o reconocerse la palabra reservada "else" y un bloque de código seguido de corchetes.

1.5. Funciones de Entrada/Salida

Para implementar las funciones de entrada y salida, añadiremos "printf" como palabra reservada, y añadiremos las siguientes reglas:

```
sentencia:  [...]
            |   PRINTF '(' STRING formatting ')'
            ;

formatting:     /* no more parameters */
            |   ',' expresion                   { printf ("%s .\n", $2) ; FF; }
                    formatting
            ;
```

Con estas reglas, ya nos aseguramos que todos los prints son correctamente traducidos: La primera string es ignorada: en caso de que se quiera imprimir una string, se debe usar el método `puts(str)`, que es traducido de esa manera. Sin embargo, si se usa la primera string para imprimir con formato otras variables que se pasan como argumento, estas serán imprimidas en pantalla de forma correcta.

Nótese que la implementación completa de "printf" es sencilla, haciendo falta tan sólo añadir una línea en la semántica de `sentencia` para imprimir la cadena `STRING` dividida en tokens (mediante el método `strtok` dentro de `<string.h>`) sustituyendo cualquier %[type] por cada uno de los parámetros adicionales recopilados en `formatting`. Sin embargo, para pasar los test propuestos por el profesorado dejamos la función tal cual planteaba el enunciado.

1.6. Vectores & Matrices

La declaración de vectores en `Forth` incluye la reserva de memoria correspondiente al tamaño del vector en la misma línea, guardando con el identificador de la variable el puntero correspondiente a la primera posición del vector. Esto es sencillo para la primera dimensión, y algo más enrevesado para más de ellas. Para traducir las declaraciones desde `C`, hemos usado la siguiente regla dentro de las definiciones de posibles variables:

```
vars:           [...]
                | INTEGER IDENTIF '[' NUMERO ']' ';'
                    { createArray($2, atoi($4)); }
                    vars
                | INTEGER IDENTIF '[' NUMERO ']' '[' NUMERO ']' ';'
                    { createMatrix($2, atoi($4), atoi($7)); }
                    vars
                ;
```

Ahora podemos ver cómo las reglas que expanden el no-terminal `vars` han sido ampliadas para aceptar tanto vectores de una dimensión como de dos. De la misma manera, podemos ver que en estas reglas se hace referencia a dos métodos:

- `createArray(char *variable_id, int size)`: que genera un vector de tamaño indicado por la sentencia en `C`, usando la declaración en `Forth variable <nombre><tamaño>cells allot`.
- `createMatrix(char *id, int x, int y)`: en el que se genera la matriz con las dimensiones definidas. Este método declara una función completa en `Forth`, que usaremos para crear matrices:

  ```
  : matrix-n-m ( #rows #cols -- )
  CREATE DUP , * ALLOT
  DOES> ( member: row col -- addr )
  ROT OVER @ * + +  CELL+ ;
  ```

  Esta función solo se declará la primera vez que sea necesaria para la definición de matrices, algo que sabremos usando una variable global definida en el traductor. Teniendo esta función definida, la declaración de matrices se realiza con la sentencia `<x><y>matrix-n-m <nombre>`. Consideramos esta solución la más elegante para resolver el problema de las matrices, creando la estructura de datos pertinente junto a la implementación del acceso a sus elementos en una función puramente de `Forth`, en lugar de realizar operaciones manuales y propensas a errores cada vez que se quiera crear una matriz o acceder a sus elementos. Esta estructura de datos guarda la dimensión que necesita para calcular el desplazamiento en la primera celda, y reserva tras ella el número necesario de celdas extra para guardar todos los elementos restantes de la matriz. Esta solución evita la necesidad de elaborar una tabla de símbolos en nuestro código de `C`, ya que la crea en el denominado "disco" de `Forth`.

Usando estas reglas para declarar vectores matrices, tan solo nos queda añadir las reglas necesarias para reconocerlas como operandos:

```
operando:           IDENTIF                 { sprintf($$, "%s @", [...]($1)); }
                |   IDENTIF '[' expresion ']'
                    {
                        char* aux = malloc(64*sizeof(char));
                        if (aux == NULL) { perror("Error at malloc\n");}
                        sprintf(aux, "%s", $3);
                        sprintf($$, "%s %s CELLS + @", [...]($1), aux);
                        free(aux);
                    }
                |   IDENTIF '[' expresion ']' '[' expresion ']'
                    { sprintf($$, "%s %s %s @", $3, $6, [...]($1)); }
```

Como vemos, para acceder a los elementos de una matriz tan sólo hace falta introducir las coordenadas, el identificador y la palabra reservada que obtiene el elemento de memoria en `Forth`.

1.7. Funciones

Para permitir el uso de funciones distintas al main en nuestro programa en `C` a traducir, añadimos las siguientes líneas:

```
program:            vars functions main    { ; }
                ;

functions:          /* no functions  */     { variable_cnt = 0; }
                |   IDENTIF '('
                    {
                        strcpy (funcion_actual, $1) ;
                        variable_cnt = 0;
                    }
                    parameters ')' '{'
                    vars
                    {
                        if(strcmp("", $4)) {
                            char * id;
                            sprintf(id, "%s", $4);
                            printf("%s !\n", [...](id));
                        }
                    }
                    codigo '}'                { printf (" ;\n"); }
                    functions
                ;

parameters:       /* no parameters declaration */   { sprintf($$, ""); }
                | INTEGER IDENTIF
                    {
                        sprintf($$, "%s", $2);
```

```
                    createVariable($2);
                }
            ;

[...]

sentencia:                              { ; }
            | funcall                  { printf("%s\n", $1) }
            | [...]
            | RETURN expresion         { printf("%s\n", $2); }
            ;

funcall:      IDENTIF '(' arguments ')'
                {
                    char* aux = malloc(64*sizeof(char));
                    if (aux == NULL) { perror("Error at malloc\n"); }
                    sprintf(aux, "%s", $1);
                    sprintf($<cadena>$, "%s%s", $<cadena>3, aux);
                }
            ;

arguments:                                  { sprintf($<cadena>$, ""); }
            |   expresion more_arguments
                { sprintf($<cadena>$, "%s%s ", $<cadena>1, $<cadena>2); }
            ;

more_arguments:                             { sprintf($<cadena>$, ""); }
            |   ',' expresion more_arguments
                { sprintf($<cadena>$, " %s%s", $<cadena>2, $<cadena>3); }
            ;
```

El primer no terminal, `functions`, reconoce las funciones declaradas antes del método main, así como el cuerpo de la función que se está reconociendo. Viendo las acciones semánticas, podemos observar cómo al reconocer el inicio de la función, la cuenta de variables se inica a 0 de nuevo para distinguir las variables locales de las globales en el nombre. Después, se reconoce la estructura propia de una función en C: nombre, parámetros, declaración de variables (como ya hemos explicado previamente en el documento) y el bloque de código apropiado. Los parámetros a reconocer se hacen a través de un no-terminal propio, que puede ser lambda (la función no recibe parámetros). Podemos ver también que en `sentencia` se añaden las reglas para aceptar sentencias de return y de llamadas a funciones, que llevan a su propio no-terminal.

Cabe destacar la implementación de las llamadas a funciones, que por limitaciones de `Bison` tan sólo son posibles con funciones que tienen un parámetro: repitiendo una sintaxis parecida a la de la llamada a la función (que funciona con tantos atributos como sea necesario) para la definición de la función con más de un parámetro, el código producía errores fuera de nuestro entorno de trabajo, por lo que nos limitamos a definir funciones con hasta un parámetro.

1.8. Sufijos en Variables Locales

Debido a que en `Forth` todas las variables conviven en el mismo ámbito, debemos tener algún mecanismo que evite que dos variables tengan el mismo nombre estando en funciones distintas (algo que es perfectamente válido en `C`). Para evitarlo, pondremos como sufijo al nombre de la variable el nombre de la función. Esto se hace usando la variable global `funcion_actual`, que almacena el nombre de la función cuyo ámbito está siendo analizado; y las variables `variable` y `variable_cnt`, que se encargan de almacenar y contar las variables que se declaran. Se utilizan para ello los métodos de crear y almacenar variables, utilizadas en las acciones semánticas que hacen uso de las variables (declaraciones, expresiones y asignaciones).

Tras ello, el método que se usa para asegurar que en una referencia a un variable se usa nombre correcto es el método `*getScopeVariableId(char *variable_id)`, que recibe el nombre de la variable a la que se ha hecho referencia en el código e itera por todas las variables almacenadas que han sido declaradas en su misma función. En caso de que se obtenga una coincidencia, se devuelve el nombre con el sufijo correcto (`_<funcion_actual>`) y, en caso de que no, se asume que es una variable global y se imprime con el sufijo adecuado (`_global`).

2. Modificación de los test propuestos

La única modificación necesaria para la correcta ejecución de los test es, en `dowhile.c`, modificar la rara estructura de expresión `2*(a/2) != a` por `(a/2)*2 != a` o por la desde luego más sensata `a% 2`. Por algún motivo que no logramos entender, la parte derecha a un operador (que deriva a una expresión exactamente igual que la del lado izquierdo) genera una violación de segmento si ésta está entre paréntesis. Atribuimos el problema a `Bison` y no hemos sido capaz de replicarlo en más ambientes que el propuesto por esta sentencia de control if. Todos los demás test son pasados con éxito, y añadimos nuestro propio archivo de test donde evaluamos todos los puntos opcionales implementados, como son las llamadas funcionales a funciones y el uso de nuestra función de matrices.

Incluimos un script (`run_tests.sh`) para la ejecución de todos los tests que dejarán el código en `Forth` en la carpeta `test_results` y el output de su ejecución en `GForth` en `test_output`.

3. Conclusiones, problemas encontrados y opinión personal

El entorno de trabajo que ofrece `Bison` es bastante claro para la implmentación de los distintos apartados de un traductor. Sin embargo, es el propio compilador de `Bison` el que ha provocado muchos errores de código de `C` que es perfectamente correcto en un entorno normal, retrasando bastante nuestro trabajo.

Por otra parte, el lenguaje de `Forth` supuso un reto para su comprensión de primeras, pero tras un estudio más en profundidad esta práctica ha servido para conocer lo que casi se podría considerar una nueva forma de programación para muchos de nosotros, siendo la notación postfija otra forma de pensar las operaciones.

Así, estamos contentos con el desarrollo de la práctica. Creemos que nos ha servido para entender cómo funciona por dentro un compilador y que hemos aprendido bastante conocimiento relacionado con otros aspectos con los que, de otra forma (o tan sólo con la teoría), no hubiésemos tenido contacto.