UNIVERSITY CARLOS III OF MADRID

LANGUAGE PROCESSORS

GROUP 83

# C to GForth translator

*Authors:*

Daniel MEDINA GARC'IA: `100316850@alumnos.uc3m.es`
Diego VICENTE MART'IN: `100317150@alumnos.uc3m.es`

4 May 2016

## Index

# 1. Description of the Exercises

In the following, we describe the reasoning followed to solve each of the exercises proposed in the statement. In order to avoid adding code fragments related to exercises not yet explained, we will skip certain parts of the source code that are not relevant by using the notation `[...]` to delimit the omitted parts.

## 1.1. Variables Local

Since `Forth` definitions must be made before the main method (`main()`) is declared, we need a way to make variable declarations that happen within the
This method before declaring it in `Forth`. In order to do this, the solution we found to the problem was to postpone in a certain way the semantic action that the method declares:

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

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

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

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

In this way, we can see how both global (before the main declaration) and local variable declarations are accepted. In case the recognition of lines of code is finished, the `Forth` syntax indicating the start of a method is printed before so that they become part of the method. For the moment, only the existence of a main method is contemplated. The exact mechanics of such a check will be discussed in more detail in future exercises.

## 1.2. Do- While

To introduce the recognition of do-while loops, we only need to make a slight modification in the while loop, whose code is given with the initial code:

```
code:         /* lambda */                        { ; }
            | sentence ';'  code                  { ; }
            | code flow                           { ; }
            ;

flow:       [...]
            |                                     DO{ printf("begin") ; FF; FF; }
                    '{' code '}'
                    WHILE expression ';'          { printf("%s while repeat", $7) ; FF; }
            [...]
            ;
```

You can see the distinction between the non-terminal `statement` and `flow`. Both are lines of code, but this distinction became necessary with the introduction we wanted to make for `for` loops that required single statements in their definition.

Here we can see that when the "do" token is recognised, the declaration of the code block associated with the loop is translated into `Forth.` Along with the `Forth` reserved words, the line breaks necessary for the readability of the generated code will be printed. Once the last part (the "while" token and the condition associated with the block) is recognised, the last part of the loop is translated into `Forth`. Finally, a recursive call to the non-terminal `code` is included, to continue recognising the rest of the input.

## 1.3. Logical & Comparis Operators on

In order to be able to implement logical operators, we must add multiple production rules with their corresponding side effects. We must also take into account the different natures of operators: we can add simple operators as literals to recognise, but this will cause conflicts with operators composed of more than one character. To solve this problem, we willd operators with more than one character to the list of reserved words.

```
    t_reserved pal_reserved [] = { // defines the reserved words and the
        {"main",       MAIN},          // and the associated
        tokens [...]
        {"++",         INC},
        {"--",         DEC},
        {"==",         EQUALS},
        {"!=",         NOTEQUALS},
        {">=",         GE},
        {"<=",         LE},
        {"||",         OR},
        {"&&",         AND},
        [...]
    } ;
```

With this, we only need to add the new rules that make the sentences with these characters be accepted, associating them to their respective operators in Forth:

```
expression:                                  term{ sprintf($$, "%s", $1) }
        |   expressio '+' expression         { sprintf($$, "%s %s +", $1, $3); }
                n
        |   expressio '-' expression         { sprintf($$, "%s %s -", $1, $3); }
                n
        |   expressio '*' expression         { sprintf($$, "%s %s *", $1, $3); }
                n
        |   expressio '/' expression         { sprintf($$, "%s %s /", $1, $3); }
                n
            |expression '&'                  expression{ sprintf($$, "%s %s and", $1, $3);
        }
            |expression '|'                  expression{ sprintf($$, "%s %s or", $1, $3);
        }
            |expression '%'                  expression{ sprintf($$, "%s %s %s mod", $1,
        $3); }
            |expression '>'                  expression{ sprintf($$, "%s %s >", $1, $3); }
            |expression '<'                  expression{ sprintf($$, "%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));
        }
        |   '!' expression                   { sprintf($$, "%s 0=", $2); }
        |   expression EQUALS expression     { sprintf($$, "%s %s =", $1, $3); }
        |   NOTEQUALS expression             { sprintf($$, "%s %s = 0=", $1, $3); }
            expression
        |   expression LE expression         { sprintf($$, "%s %s <=", $1, $3); }
        |   expression GE expression         { sprintf($$, "%s %s >=", $1, $3); }
        |   expression AND expression        { sprintf($$, "%s %s and", $1, $3); }
        |   expression OR expression         { sprintf($$, "%s %s or", $1, $3); }
        ;
```

It should be noted here that the expression is not printed directly as in the initial code, but is passed internally as a string at the top level of the derivation tree. This allows for greater flexibility, readability and easier debugging of the code.

However, if we try to compile this code with bison, we will get a number of shift/reduce conflicts. To solve them, we must explicitly add to bison the order of precedence of operators and associativity. To do so, we chose the option of adding the following rules to the file after the token declaration instead of explicitly stating them in each line of section 2:

```
%right '='                    // lower order of precedence
%left OR
%left AND
%left '|'
%left '&'
%left NOTEQUALS EQUALS EQUALS
```

```
%left GE LE '<' '>'
%left '+' '-'
%left '*' '/' '%' '%'
%left INC DEC '!
                                %left_UNARY_SIGN// highest order of precedence
```

With all this, we are ready to recognise both simple and compound operators.

## 1.4. Structure If- Else

In order to add the logical if-else control structure, the following rules have been declared:

```
flow:        [...]
             |   IF expression                      printf("%s if", $2) ; FF; }
                     '{' code '}'
                     else_block
             [...]
             ;

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

In this way, we can ensure that if the reserved word "if" is detected in any line of code, that production rule will be used. As a conidiction, both an arithmetic and a logical comparison expression are accepted, as well as independent operands (e.g. `if (0)`). After a code block, the non-terminal `else block` is expected: it may be empty (in case the "if" being evaluated does not have an else clause) or the reserved word "else" and a code block followed by square brackets are recognized.

## 1.5. Functions of Input/Output

To implement the input and output functions, we add "printf" as a reserved word, and add the following rules:

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

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

With these rules, we make sure that all prints are correctly translated: The first string is ignored: in case you want to print a string, you must use the `puts(str)` method, which is translated in this way. However, if you use the first string to format print other variables passed as arguments, they will be printed correctly on the screen.

Note that the full implementation of "printf" is straightforward, requiring only the additional line in the `statement` semantics to print the `STRING` string split into tokens (using the `strtok` method inside <`string.h`>) replacing any %[type] by each of the additional parameters collected in `formatting`. However, in order to pass the tests proposed by the teachers, we left the function as stated in the statement.

## 1.6. Vectors & Matrices

The vector declaration in Forth includes the memory reservation corresponding to the size of the vector on the same line, storing with the variable identifier the pointer corresponding to the first position of the vector. This is simple for the first dimension, and somewhat more convoluted for more of them. To translate the declarations from C, we have used the following rule within the definitions of possible variables:

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

We can now see how the rules that expand the non-terminal `vars` have been extended to accept both one- and two-dimensional vectors. Likewise, we can see that two methods are referred to in these rules:

- `createArray(char *variable id, int size)`: which generates a vector of size indicated by the C statement, using the declaration in Forth `variable <name><size~no>cells allot`.
- `createMatrix(char *id, int x, int y)`: where the matrix with the defined dimensions is generated. This method declares a complete function in Forth, which we will use to create matrices:

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

  This function is only declared the first time it is needed for the definition of matrices, which we will know by using a global variable defined in the translator. Having this function defined, the declaration of matrices is done with the statement `<x><y>matrix-n-m <name>`. We consider this the most elegant solution to solve the matrix problem, creating the relevant data structure together with the implementation of the access to its elements in a purely Forth function, instead of performing manual and error-prone operations every time we want to create a matrix or access its elements. This data structure stores the dimension needed to calculate the displacement in the first cell, and reserves after it the necessary number of extra cells to store all the remaining elements of the matrix. This solution avoids the need to build a symbol table in our C code, since it creates it in the so-called Forth "disk".

Using these rules for declaring matrix vectors, all that remains is to add the necessary rules for recognising them as operands:

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

As we can see, to access the elements of an array, it is only necessary to enter the coordinates, the identifier and the reserved word that obtains the element from memory in Forth.

## 1.7. Functions

To allow the use of functions other than main in our C program to be translated, we add the following lines:

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

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

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

```
                        createVariable($2);
                    }
                ;

[...]

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

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

arguments:                                        { sprintf($<string>$, ""); }
                |more_arguments_expression
                    { sprintf($<string>$, "%s%s ", $<string>1, $<string>2); }
                ;

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

The first non-terminal, `functions`, recognises the functions declared before the main method, as well as the body of the function being recognised. Looking at the semantic actions, we can see how when recognising the start of the function, the variable count is reset to 0 again to distinguish local variables from global variables in the name. Afterwards, the structure of a C function is recognised: name, parameters, variable declaration (as we have already explained previously in the document) and the appropriate code block. The parameters to be recognised are done through a non-terminal of its own, which can be a lambda (the function does not receive parameters). We can also see that the rules for accepting return statements and function calls, which lead to their own non-terminal, are appended to the `statement.`

It is worth mentioning the implementation of function calls, which, due to `Bison`'s limitations, are not always possible.
are only possible with functions that have one parameter: by repeating a syntax similar to that of the function call (which works with as many attributes as necessary) for the definition of the function with more than one parameter, the code would produce errors outside our working environment, so we limit ourselves to defining functions with up to one parameter.

### 1.8. Variable Suffixes Locals

Because in Forth all variables coexist in the same scope, we must have some mechanism to avoid that two variables have the same name when they are in different functions (something that is perfectly valid in C). To avoid this, we will suffix the variable name with the name of the function. This is done using the global variable current function, which stores the name of the function whose scope is being analysed; and the variables variable and variable cnt, which are in charge of storing and counting the variables that are declared. The methods of creating and storing variables, used in the semantic actions that make use of variables (declarations, expressions and assignments), are used for this purpose.

After that, the method used to ensure that the correct name is used in a reference to a variable is the method *getScopeVariableId(char *variable id), which receives the name of the variable that has been referenced in the code and iterates through all the stored variables that have been declared in the same function. If a match is obtained, the name is returned with the correct suffix ( <current function>) and, if not, it is assumed to be a global variable and printed with the appropriate suffix ( global).

## 2. Modification of the proposed tests

The only modification necessary for the correct execution of the tests is, in dowhile.c, to modify the strange expression structure 2*(a/2) != a by (a/2)*2 != a or by the of course more sensible a % 2. For some reason that we do not understand, the right hand side of an operator (which leads to an expression exactly the same as the left hand side) generates a segment violation if
We attribute the problem to Bison and have not been able to replicate it in other environments than the one proposed by this if control statement. We attribute the problem to Bison and have not been able to replicate it in other environments than the one proposed by this if control statement. All the other tests are passed successfully, and we add our own test file where we evaluate all the optional points implemented, such as the functional function calls and the use of our array function.

We include a script (run_tests.sh) for the execution of all the tests that will leave the code in Forth in the folder test results and the output of its execution in GForth in test output.

## 3. Conclusions, problems encountered and personal opinion

The working environment offered by Bison is quite clear for the implementation of the different sections of a translator. However, it is the Bison compiler itself that has caused many errors in C code that is perfectly correct in a normal environment, slowing down our work considerably.

On the other hand, the Forth language was a challenge to understand at first, but upon further study this practice has served to introduce us to what could almost be considered a new form of programming for many of us, postfix notation being another way of thinking about operations.

So, we are happy with the development of the internship. We believe that it has helped us to understand how a compiler works inside and that we have learnt a lot of knowledge related to other aspects that we would not have had contact with otherwise (or only with theory).