Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Tradutor de BASH Script para C

Felipe Carvalho Gules
08/29137

Revisão do trabalho prático de Tradutores.

Professora
Profª Drª Cláudia Nalon

Brasília, 25 de Julho de 2014

1.Introdução

O trabalho consiste em traduzir códigos escritos em linguagem *BASH Script,* que é uma linguagem interpretada, em um código escrito em linguagem C, podendo assim ser compilada para linguagem de máquina. Tornando o programa mais rápido de ser executado e mais seguro (já que é difícil a tradução de linguagem de máquina). Essa tradução pode assim ser útil para administradores de sistemas GNU/Linux e administradores de redes.

O trabalho esta dividido em três módulos, cada um tratando das diferentes etapas da compilação. O primeiro módulo, o analisador léxico é a etapa do processo de tradução que reconhece *lexemas* e forma *tokens*, nesta etapa foi utilizada a ferramenta *FLEX*.

O segundo módulo, realiza a análise sintática, e construção de uma árvore sintática (*parsing tree*). O Analisador sintático é a etapa da tradução em que verifica-se a estrutura gramatical da sequência de *tokens* fornecida pelo analisador léxico. Foi utilizada a ferramenta *Bison.*

O terceiro e último módulo trata da geração de código intermediário, que consiste em gerar um código a partir da árvore sintática anotada(árvore sintática com regras). Essa árvore normalmente é composta por três endereços, uma vez que esta característica facilita a geração do código em C.

Dentre uma variedade de justificativas para a tradução realizada com um tradutor de *Bash* para C, como a apresentada neste trabalho, destaca-se a vantagem de que se torna mais prático para os programadores de *Bash Script* e usuários leigos traduzirem seus códigos rapidamente.

2.Implementação e funcionamento do programa

O programa utiliza as ferramentas *FLEX* e *BISON* que são implementações em software livre das antigas ferramentas LEX e YACC que facilitam e auxiliam o programador no processo de tradução.

Está dividido em três arquivos que são compilados juntos, utilizando o FLEX, BISON e GCC. para facilitar esse processo de compilação, basta executar o arquivo "compila.sh":

```
$./compila.sh
```

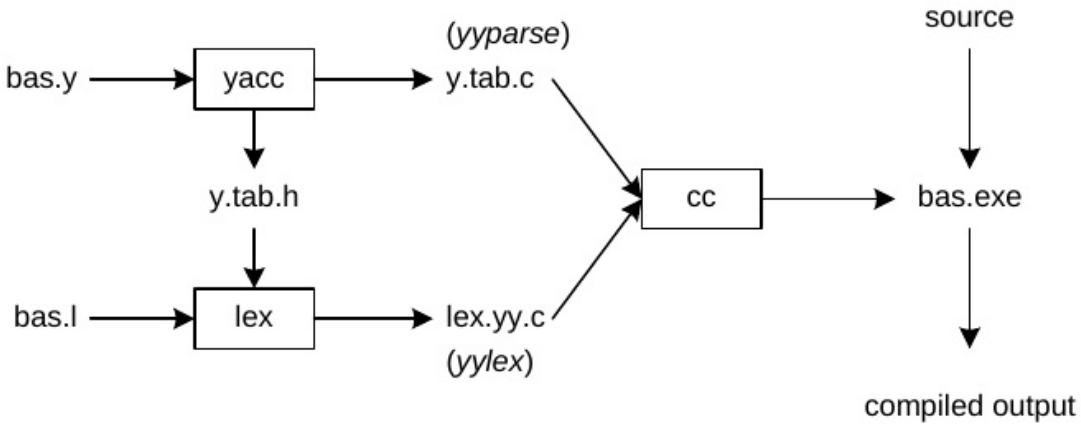
E separadamente utilizar os seguintes comandos:

```
$ lex lexico.l
$ yacc sintatico.y -d -y
$ gcc -c y.tab.c lex.yy.c
$ gcc y.tab.o lex.yy.o btoc.c -o btoc
```

para execução, basta digitar:

```
$ ./btoc < entrada.sh
$ ./btoc < entrada_com_erros.sh
```

Sendo que o objetivo do programa é traduzir um código em BASH script em outro código equivalente em *C*. Ele recebe como entrada um arquivo de *BASH script* simples, separa os *tokens* reconhecidos pelo analisador *léxico*, e depois é gerado uma árvore sintática anotada pelo analisador sintático/semântico, e ao fim, essa árvore é impressa em tela estruturada como um código em C.



Processo de compilação [5]

No caso desse trabalho, a entrada é um arquivo de BASH script e o compilador deve retornar um arquivo texto com o código equivalente em C.

3.Analisador Léxico

O arquivo "lexico.l" possui toda a especificação para retorno dos *tokens* para o analisador sintático. Seguem as regras de reconhecimento dos *tokens* e o tratamento dado:

```
%}

digits      [0-9]
ltr         [a-zA-Z]
alphanum    [a-zA-Z0-9]

%%
 /*Cabecalho e comentarios*/
#[^\n]*         line++;

 /*Argumentos e Variaveis*/
"$"{alphanum}+  {yylval.sValue = strdup(yytext+1); return(VAR);}
{digits}+       {yylval.iValue = atoi(yytext); return(INT);}

 /*Strings*/
{ltr}?\"(\\.|[^\\"])*\"   {yylval.sValue = strdup(yytext); return(STRING);}

 /*Funcoes*/
"case"                return(CASE);
"do"                  return(DO);
"done"                return(DONE);
"echo"      return(ECHO);
"else"                return(ELSE);
"elif"                return(ELIF);
"esac"                return(ESAC);
"fi"        return(FI);
"for"       return(FOR);
"function"  return(FUNCTION);
"if"                  return(IF);
"in"        return(IN);
"read"      return(READ);
"then"                return(THEN);
"until"     return(UNTIL);
"while"     return(WHILE);

[-(){}<>=+*/;&|]    return(*yytext);
"["         return(*yytext);
"]"         return(*yytext);
".."        return(LOOPFOR);
";;"        return(SEMI_SEMI);

"++"                  return(INC_OP);
"--"                  return(DEC_OP);
"&&"                  return(AND_OP);
"||"                  return(OR_OP);
"<="                  return(LE_OP);
"-le"                 return(LE_OP);
"-lt"                 return(LT_OP);
">="                  return(GE_OP);
```

```
"-ge"                   return(GE_OP);
"-gt"                   return(GT_OP);
"=="                    return(EQ_OP);
"-eq"                   return(EQ_OP);
"!="                    return(NE_OP);
"-ne"                   return(NE_OP);

{ltr}+          {yylval.sValue = strdup(yytext); return(VAR);}

[\n]            {line++; return(*yytext);}

[ \t\v\f]+ { /* consome espaco em branco */ }

.               {printf("\n// ERRO LEXICO: Letra ou simbolo desconhecido:%s
na linha %d\n",yytext, line);}
%%
```

Observação: o BASH não faz distinção entre tipos das variáveis [1], e não reconhece números reais(*float*), apenas inteiros não negativos. Contudo, é possível fazer declaração de variáveis de caracteres, ou de números inteiros, ainda que sem a verificação de tipos.

Durante a análise léxica, os caracteres não reconhecidos retornam o seguinte erro:

```
\\ ERRO LEXICO: Letra ou simbolo desconhecido:@ na linha 12
```

Esse retorno indica que o caractere '@' não foi reconhecido e estava na linha 12 do código em BASH. As duas contra-barras adicionadas, servem para que no arquivo de saída (o código em C) essa linha fique no formato de comentário. A política adotada é continuar o processo de compilação, assim, um caractere não identificado não interrompe o processo de compilação, apenas gera um alerta.

4.Analisador Sintático

A gramática do analisador sintático, foi reduzida e simplificada para as seguintes estruturas:

- Estruturas condicionais "IF, ELIF, ELSE" e "CASE";
- Estrutura de repetição "WHILE", UNTIL e FOR;
- Expressões de comparação e aritimética;
- Comando para ler entrada do teclado e imprimir em tela: "READ" e "ECHO".

A gramática utilizada nessa parte do trabalho é reduzida [7]:

```
inputunit
        : inputunit pipeline
        | /* NULL */
        ;

number
        : INT
        ;

word
        : VAR
        | STRING
        ;

command
        : shell_command
        | function_def
        | list_terminator
        ;

shell_command
        : for_command
        | if_command
        | case_command
        | expr linebreak
        | ECHO linebreak expr list_terminator
        | READ linebreak expr list_terminator
        | VAR '=' expr
        | '(' '(' VAR '=' expr ')' ')'
        | WHILE '[' expr ']' list_terminator DO compound_list DONE
        | UNTIL '[' expr ']' list_terminator DO compound_list DONE
        ;

for_command
        : FOR '(''(' word '=' number ';' expr ';' expr ')'')'
list_terminator DO compound_list DONE
        | FOR '(''(' word '=' number ';' expr ';' expr ')'')' group_command
        | FOR word IN '' list_terminator DO compound_list DONE
```

```
        | FOR word IN '' list_terminator group_command
        ;

if_command
        : IF '[' expr ']' list_terminator THEN compound_list FI %prec IFX

        | IF '[' expr ']' list_terminator THEN compound_list ELSE
compound_list FI

        | IF '[' expr ']' list_terminator THEN compound_list elif_clause FI

        ;

elif_clause
        : ELIF '[' expr ']' list_terminator THEN compound_list %prec IFX

        | ELIF '[' expr ']' list_terminator THEN compound_list ELSE
compound_list

        | ELIF '[' expr ']' list_terminator THEN compound_list elif_clause

        ;

case_command
        : CASE word linebreak IN case_clause_sequence linebreak ESAC

        | CASE word linebreak IN case_clause ESAC

        ;

case_clause
        :  pattern_list
        |  case_clause_sequence pattern_list
        ;

pattern_list
        : linebreak pattern ')' compound_list
        | linebreak pattern ')' linebreak
        | linebreak '(' pattern ')' compound_list
        | linebreak '(' pattern ')' linebreak
        ;

case_clause_sequence
        :  pattern_list SEMI_SEMI
        |  case_clause_sequence pattern_list SEMI_SEMI
        ;

pattern
        :  number
        |  '*'
        |  pattern '|' number
        ;


function_def
        : FUNCTION VAR linebreak group_command
```

```
        ;

group_command
        : linebreak '
        ;

compound_list
        : linebreak list1
        ;

list1
        : list1 linebreak pipeline
        | pipeline
        ;

list_terminator
        : '\n'
        | ';'
        ;

newline_list
        : '\n'
        | newline_list '\n'
        ;

linebreak
        : newline_list %prec NX
        | %prec NX
        ;

expr
        : word
        | number
        | expr '+' expr
        | expr INC_OP
        | expr '-' expr
        | expr DEC_OP
        | expr '*' expr
        | expr '/' expr
        | expr '<' expr
        | expr '>' expr
        | expr GE_OP expr
        | expr GT_OP expr
        | expr LE_OP expr
        | expr LT_OP expr
        | expr NE_OP expr
        | expr EQ_OP expr
        | expr AND_OP expr
        | expr OR_OP expr
        | '(''(' expr ')'')'
        ;

pipeline
        : pipeline '|' command
        | command linebreak
        ;
```

Nesse estudo fez-se necessário introduzir estruturas e funções para gerar a árvore sintática. Foram introduzidas três estruturas de dados, uma para armazenar números, a segunda para armazenar identificadores e a terceira para armazenar operadores. A terceira estrurura possui um campo para indicar quantos filhos o nó possui e outro campo para ligar as estruturas anteriores:

```
typedef struct nodeTypeTag {
    nodeEnum type;              /* tipo do noh */

    union {
        numNodeType num;        /* numeros */
        idNodeType id;          /* identificadores */
        oprNodeType opr;        /* operadores */
    };
} nodeType;
```

A construção da árvore sintática, é feita ligando as estruturas em memória: quando um *token* é reconhecido pelo analisador sintático, uma ação relacionada a esse *token* é realizada. Neste caso é alocado em memória a estrutura contendo os valores.

Também é utilizada uma tabela de símbolos que armazena os identificadores distinguindo-os entre "variável" ou "função". Quando uma variável é recebida do analisador léxico, há a verificação de sua possível existência na tabela de símbolos. Caso a variável não esteja presente na tabela, então é inserido o novo símbolo.

Segue a estrutura tabela de símbolos:

```
struct tabela_simbolos;
typedef struct tabela_simbolos {
        int type;              /* tipo da variavel*/
        char* value;            /* valor da variavel */
        int hasValue;          /* inicializacao */
        struct tabela_simbolos* next;
} tabela_simbolos;
```

Caso um número inteiro seja recebido pelo analisador sintático a ação da regra relacionada grava o valor em memória na estrutura correspondente. Caso seja um operador, a estrutura grava em memória seu valor e os ponteiros dos valores de seus respectivos filhos.

5.Gerador de código

Ao final do processo, a função contida no arquivo "btoc.c" percorre a árvore sintática imprimindo em tela o código equivalente em C.

A medida que a função gera_codigo() percorre a árvore em profundidade verificando os nós, ela imprime as estruturas no formato da linguagem em C. Por exemplo a estrutura em *bash script*, e sua estrutura equivalente em C:

`while [ $x -le 5 ]` `do` `  echo x` `done`	`while (x <= 5)` `{` `  printf("%d\n", x);` `}`

6.Arquivos de testes

Junto com os arquivo do compilador, dois arquivos são referentes aos testes, um contendo um código correto e outro código incorreto. Utilizei as principais estruturas que podem ser traduzidas no código em *Bash*. Seguem suas respectivas execuções:

entrada sem erros:

(entrada)	`#/bin/bash` `read x` `y=5` `for ((i=0;i<=10;i++))` `do` `  echo "estou no for"` `done`

BASH

```
function hello {
    echo "Funcao Hello"
    (( y = x / 2 ))
    echo $y
}

COUNTER=20

until [ $COUNTER -lt 10 ]
do
    echo $COUNTER
    (( COUNTER-- ))
done

while [ $x -le 10 ]
do
    if [ $x -le 2 ] ; then
        (( y = y - 1 ))
        echo "entrou no if"
    elif [ $x -ge 5 ] ; then
        echo "entrou no elif"
        (( y = y - 1 ))
    else
        echo "entrou no else"
    fi
    (( x++ ))
done

hello

echo "Switch case, (1) case simples, (2) case | case, qualquer
outro numero, DEFAULT "
read DISTR

case $DISTR in
    1)
        echo "Case Simples."
        ;;
    2|3)
        echo "case | case)"
        ;;
    *)
        echo "Default."
        ;;
esac
```

(saída)

```
#include<stdio.h>

int main() {


    unsigned int x;
    scanf("%d",&x);
```

C

```c
    unsigned int y;
    y = 5;

    unsigned int i;
    for (i = 0;i <= 10;i++)    {
        printf("estou no for");
    }

    void hello(){
        printf("Funcao Hello");
        y = x / 2;
        printf("%d\n", y);
    }


    unsigned int COUNTER;
    COUNTER = 20;
    while (!(COUNTER < 10))
    {
        printf("%d\n", COUNTER);
        COUNTER--;
    }
    while (x <= 10)
    {
        if (x <= 2)
        {
            y = y - 1;
            printf("entrou no if");
        }
        else if (x >= 5)
        {
            printf("entrou no elif");
            y = y - 1;
        }
        else {
            printf("entrou no else");
        }
        x++;
    }
    hello();
    printf("Switch case, (1) case simples, (2) case | case,
qualquer outro numero, DEFAULT ");

    unsigned int DISTR;
    scanf("%d",&DISTR);
    switch (DISTR) {
        case 1:
            printf("Case Simples.");
            break;
        case 2: case 3:
            printf("case | case)");
            break;
        default:
            printf("Default.");
            break;
    }
```

	`` return 0; } ```

entrada com erros:

(entrada) BASH	```#/bin/bash read x y=5 @ while [$x -le 10] do if { $x -le 2 } ; then ((y = y - 1)) echo "entrou no IF" elif [$x -ge 5] ; then echo "entrou no elif" ((y = y - 1)) else echo "entrou no else" fi ((x++)) done while ((y = x / 2)) echo $y echo $x```
(saída) C	```#include<stdio.h> int main() { unsigned int x; scanf("%d",&x); // ERRO LEXICO: Letra ou simbolo desconhecido:@ na linha 5 unsigned int y; y = 5; // ERRO SINTATICO: linha: 8: syntax error, unexpected '{', expecting '['```

```
    return 0;
}
```

Os erros introduzidos foram o caractere "@" na linha 5, e o comando "if { $x -le 2 }" na linha 8. Como visto antes, quando o analisador léxico encontra um erro, ele apenas reporta o erro e ignora o caractere não reconhecido sem passar esse caractere para o analisador semântico. O analisador sintático reporta um erro indicando a linha, em que estão contidos, informa qual o *token* era esperado receber do analisador léxico, e interrompe a compilação.

7.Resultados

O tradutor de Bash Script para C funciona corretamente para diversos tipos de procedimentos. O código de saída em C, já possui declarações e identação, deixando o usuário pronto para revisar rapidamente o código impresso em tela e compilar utilizando o *GCC* ou compilador de sua preferência.

8.Dificuldades Encontradas

Para o desenvolvimento deste estudo foi necessária intensa pesquisa haja visto a dificuldade de encontrar bibliografia referente aos problemas encontrados. Bibliografia esta que foi, não obstante, imprescindível para a execução trabalho por se tratar de um problema de grande complexidade, abstração teórica envolvendo ferramentas
de difícil depuração e tratamento de erros. Além disso a adequação do trabalho ao propósito foi outro item que dispendiou bastante tempo.

Referências

[1] The GNU Bash Reference Manual, (Acesso em 2 de Junho 2014)

[2] ANSI C grammar, Lex specification, (Acesso em 2 de Junho 2014)

[3] BNF for BASH, (Acesso em 2 de Junho 2014)

[4] Lex & Yacc Tutorial, (Acesso em 2 de Julho 2014)

[5] ANSI C Yacc Grammar, (Acesso em 2 de Julho 2014)

[6] Bison bad if structure, (Acesso em 2 de julho 2014)

[7] Basic Grammar for BASH, (Acesso em 2 de julho 2014)

[8] Bison Error Recovery, (Acesso em 5 de julho 2014)

[9] Error Reporting Recovery, (Acesso em 6 de julho 2014)