

EA872 Laboratório de Programação de Software Básico

Atividade 3



Vinícius Esperança Mantovani

RA 247395

Entrega (limite): 23/08/2023, às 13 horas

AVISO: OS CÓDIGOS FORAM ENVIADOS JUNTAMENTE COM ESTE RELATÓRIO EM UMA PASTA SEPARADA NO MOODLE.

Exercício 4)

Neste exercício, desenvolvi as seguintes definições para o analisador sintático e o léxico, respectivamente:

```
%{  
    #include <stdio.h>  
    #include <string.h>  
  
    char *names[30];  
    int count = 0;  
    int iniciada = 1;  
  
    int yylex(void);  
    int yyerror(char const *s);  
  
    int linear_search(char *names[], char *searched, int count); //busca  
    linear por uma string em uma lista de strings  
}%  
  
%union {  
    char *string_value;  
    float float_value;
```

[illegible]

```

expressions      :      exe_line expressions
                  |      exe_line
                  ;

exe_line         :      VARNAME EQ operations SEMICOL      {
                                                                if(iniciada ==
1){

names[count] = (char *)malloc(10*sizeof(char)); //adiciona a variavel na
lista de inicializadas caso nao haja nenhuma nao iniciada em "operations"

strcpy(names[count], &$1);

                                                                count++;
                                                                }
                                                                iniciada = 1;

                                                                }

                  ;

operations       :      operations OPERATOR operations
                  |      PAR_ESQ operations PAR_DIR
                  |      VALUE
                  |      VARNAME          {
                                                                if(linear_search(names, &$1,
count) == -1){

                                                                printf("ATENCAO: a variavel
%s foi usada sem ter sido inicializada!\n", &$1); //alerta sobre o uso de
variavel nao inicializada

                                                                iniciada = 0;

                                                                }

                                                                }

                  ;

%%

void main(){
    yyparse();
}

int yyerror (char const *s){
    fprintf (stderr, "%s\n", s);
}

```

```

int linear_search(char *names[], char *searched, int count){    //executa
uma busca conforme explicado na seção de definicoes
    for(int i = 0; i < count; i++){
        if(!strcmp(names[i], searched)){
            return i;
        }
    }
    return -1;
}

```

Neste analisador, foi implementada a gramática recomendada no enunciado do exercício e, para implementar o funcionamento do analisador, foram utilizadas: Uma lista de nomes “names”, o inteiro “count” e, o inteiro “iniciada”; todas definidas na seção de definições do “.y”.

A lista de nomes usada no analisador serve para guardar os nomes das variáveis iniciadas, enquanto que o inteiro “count”, é usado para conter o número de elementos nessa lista. Já quanto ao inteiro restante, “iniciada” é usado para garantir que uma variável que, por ventura, seja igualada à uma variável que não tenha sido inicializada também não seja considerada inicializada.

Ainda nessa seção, temos três tipos possíveis de dados a serem reconhecidos pelo analisador léxico em “%union”, que são: um valor string, um float e, um char. Ademais, temos, por fim, nessa seção, os tokens possíveis, conforme se ve no código.

A seguir, na seção de gramática, implementamos a gramática sugerida no enunciado deste exercício e, implementamos o seguinte:

- 1) Em “identifier”: VARNAME EQ VALUE, adicionamos um nome à lista e somamos um em “count”, pois a lista foi incrementada;
- 2) Em “exe_line”: VARNAME EQ operations SEMICOL, implementamos uma lógica igual à do citado acima, mas com a condição de que “iniciada == 1”, ou seja, de que uma possível variável, que faz parte da operação que está para definir a variável cujo nome está em VARNAME e que não esteja inicializada, não faça com que a variável em VARNAME seja dada por iniciada (adicionada à lista). Satisfeita ou não a condição, “iniciada” passa a valer 1 após o bloco “if”.
- 3) Finalmente, em “operations”: VARNAME, temos uma lógica condicional que verifica se o nome da variável dada é o de uma variável não inicializada e, neste caso, imprime um alerta a respeito disso. Além disso, altera o valor de “iniciada” para 0, para não permitir que uma variável não iniciada cause a inserção de uma variável indesejada na lista.

Para finalizar, na seção de código, temos, além dos elementos básicos necessários, uma função implementada para fazer uma busca linear entre os componentes da lista de nomes. Essa função serve para ajudar a discernir se uma variável está ou não inicializada.

Agora, ao analisador léxico:

```

%{
    #include "nondef.tab.h"
}%

main_start      main\(\) [ ]+\{

```

```

types          (int) | (float)
namevar        [A-Za-z]+[0-9]*
value          [0-9]+(\.[0-9]+)*
main_end       \}
equal          =
operator       \+|\-|\*|\/
par_dir        \)
par_esq        \(
semicolon      ;
comma          ,

%%

{main_start}   {
                sscanf(yytext, "%s", &yylval.string_value);
                return MAIN_START;
            }

{main_end}     {
                sscanf(yytext, "%s", &yylval.string_value);
                return MAIN_END;
            }

{types}        {
                sscanf(yytext, "%s", &yylval.string_value);
                return TYPE;
            }

{namevar}      {
                sscanf(yytext, "%s", &yylval.string_value);
                return VARNAME;
            }

{value}        {
                sscanf(yytext, "%f", &yylval.float_value);
                return VALUE;
            }

{equal}        {
                sscanf(yytext, "%s", &yylval.string_value);
                return EQ;
            }

```

```

{operator}      {
                  sscanf(yytext, "%s", &yylval.string_value);
                  return OPERATOR;
                }

{par_dir}       {
                  sscanf(yytext, "%s", &yylval.string_value);
                  return PAR_DIR;
                }

{par_esq}       {
                  sscanf(yytext, "%s", &yylval.string_value);
                  return PAR_ESQ;
                }

{semicolon}     {
                  sscanf(yytext, "%s", &yylval.string_value);
                  return SEMICOL;
                }

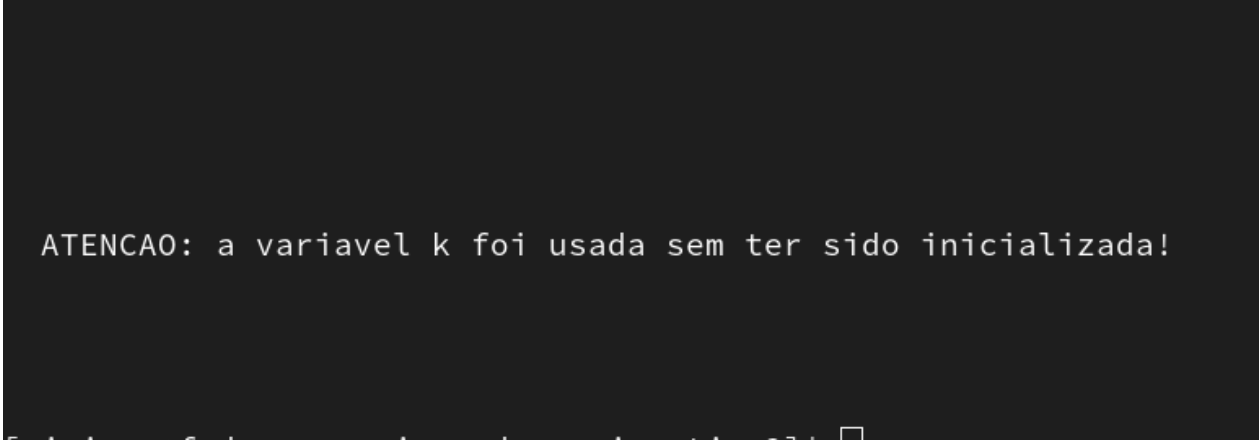
{comma}         {
                  sscanf(yytext, "%s", &yylval.string_value);
                  return COMMA;
                }

%%

```

Para evitar explicações excessivas, aqui está uma explicação minimamente simplória. Na seção de regras, define-se várias expressões regulares, responsáveis por capturar os diversos tokens necessários para a análise sintática. Sendo assim, na seção de regras, para cada expressão definida na seção anterior, pega-se o que foi capturado e se emite esse resultado para o analisador sintático, retornando-se o token referente à expressão regular definidora da regra em análise. Desse modo, o analisador léxico serve apenas para capturar os tokens e emití-los de maneira adequada para o analisador sintático, de forma a permiti-lo atuar conforme é necessário para aquilo que se pede no exercício.

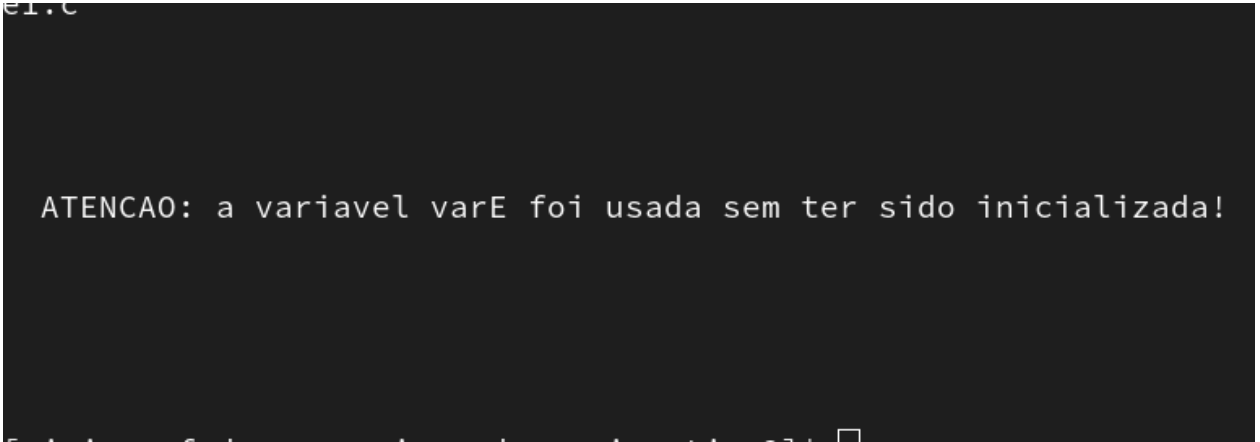
A seguir estão os outputs para cada teste disponibilizado de apoio:



ATENCAO: a variavel k foi usada sem ter sido inicializada!

This is a screenshot of a terminal window with a dark background. The text is displayed in a light-colored, monospaced font. The message is a warning about an uninitialized variable 'k'.

Figura 1: Teste 0



ATENCAO: a variavel varE foi usada sem ter sido inicializada!

This is a screenshot of a terminal window with a dark background. The text is displayed in a light-colored, monospaced font. The message is a warning about an uninitialized variable 'varE'.

Figura 2: Teste 1



ATENCAO: a variavel ponto foi usada sem ter sido inicializada!

This is a screenshot of a terminal window with a dark background. The text is displayed in a light-colored, monospaced font. The message is a warning about an uninitialized variable 'ponto'.

Figura 3: Teste 2

Figura 4: Teste 3

```
ATENCAO: a variavel ccccc foi usada sem ter sido inicializada!  
ATENCAO: a variavel aaaaa foi usada sem ter sido inicializada!
```

Figura 5: Teste 4

```
ATENCAO: a variavel tempo foi usada sem ter sido inicializada!
```

Figura 6: Teste 5

```
ATENCAO: a variavel tres foi usada sem ter sido inicializada!
```

Figura 7: Teste 6

```
ATENCAO: a variavel e foi usada sem ter sido inicializada!  
ATENCAO: a variavel e foi usada sem ter sido inicializada!  
ATENCAO: a variavel c foi usada sem ter sido inicializada!
```

Figura 9: Teste 8


```
ATENCAO: a variavel tres foi usada sem ter sido inicializada!

ATENCAO: a variavel sete foi usada sem ter sido inicializada!

ATENCAO: a variavel dois foi usada sem ter sido inicializada!
ATENCAO: a variavel sete foi usada sem ter sido inicializada!
ATENCAO: a variavel oito foi usada sem ter sido inicializada!
```

Figura 10: Teste 9

Exercício 5)

Neste exercício, os códigos implementados para os analisadores foram os seguintes:

sepcomm.l:

```
%{
#include "sepcomm.tab.h"
#include <stdio.h>
#include <string.h>

struct Lista{ //define a estrutura da lista ligada
    struct Lista *proximo;
    char params[50][50];
    char comando[50];
};
int count_com = 0; //conta o numero de comandos na lista ate o momento
int count_par = 0; //conta o numero de parametros na lista componente
do comando sendo analisado no momento

    struct Lista Comandos; //declaracao do primeiro elemento da lista de
comandos
    struct Lista *Atual_Comandos = &Comandos; //declaracao do ponteiro que
sempre aponta para o ultimo elemento adicionado na lista de comandos

    int verifica_comando = 0; //verifica se algum comando foi passado na
linha (1 se tem, 0 cc)
}%

%START          param
```

```

comments      ^#+.+$
command       ^([^:#+])+
doublepoints  :
parameter     [^\n# ]+
space         [ ]+
comma         ,

%%

{comments}    {
                sscanf(yytext, "%s", &yylval.string_value);
                return COMMENT;
            };

{space}       ;

{doublepoints} {
                BEGIN param; //entra no modo de recepcao de parametros
                sscanf(yytext, "%s", &yylval.string_value);
                return DOUBLEPNTS;
            }

\n            {
                verifica_comando = 0;
                BEGIN 0;      //volta para o modo de recepcao de comandos
                return LINEBREAK;
            }

<param>{parameter}    {
                        sscanf(yytext, "%s", &yylval.string_value);

                        if(verifica_comando == 1){ //verifica se a
linha contem realmente um comando
                            strcpy(Atual_Comandos->params[count_par],
yytext);

                            //printf("parametro adicionado: %s\n",
Atual_Comandos->params[count_par]); //teste para ver se o parametro foi
adicionado corretamente

                            count_par++;
                        }

                        return PARAMETER;
                    }

```

```

{comma}      {
                sscanf(yytext, "%s", &yyval.string_value);
                return COMMA;
            }

{command}    {
                count_par = 0; //comeca a contar o numero de parametros
deste comando

                struct Lista *i = &Comandos;
                // while(i->proximo != NULL){      //      teste para
ver se foram armazenados dcorretamente os comandos anteriores e o primeiro
de seus parametros
                //      printf("comando anterior: %s\n", i->comando);
                //      printf("param 1: %s\n", i->params[0]);
                //      i = i->proximo;
                // }

                sscanf(yytext, "%s\n", &yyval.string_value);

                if(count_com == 0){ //caso seja o primeiro comando, nao
cria pula para novo espaco da lista
                    strcpy(Atual_Comandos->comando, yytext);
                    Atual_Comandos->proximo = (struct Lista
*)malloc(sizeof(struct Lista));
                    count_com++;
                } else{ //nao eh o primeiro comando, logo, pula para o
proximo espaco

                    Atual_Comandos = Atual_Comandos->proximo;
                    strcpy(Atual_Comandos->comando, yytext);
                    Atual_Comandos->proximo = (struct Lista
*)malloc(sizeof(struct Lista));
                    count_com++;
                }
                // printf("comando atual: %s\n",
Atual_Comandos->comando); //teste para ver se o comando foi devidamente
adicionado

                verifica_comando = 1;
                return COMMAND;
            }

```

%%

É basicamente neste analisador em que a lógica de armazenamento de comandos e parâmetros ocorre. Inicialmente, cria-se uma estrutura chama lista, que funciona como a lista ligada desejada. Os elementos dessa estrutura contém um ponteiro que aponta para o próximo nó (elemento), uma lista de parâmetros e, uma string, que armazena o nome do comando. Temos ainda, dois inteiros “count_com” e “count_par”, responsáveis, respectivamente, por contar o número de comandos armazenados até o momento e, o número de parâmetros correspondentes ao comando atual que já foram armazenados. Por fim, define-se, também, um elemento do tipo referente à estrutura criada e, um ponteiro que aponta para um elemento dessa estrutura. Esse ponteiro, aponta, inicialmente, para o elemento criado “Comandos”, tem como nome “Atual_Comandos” e, será utilizado para armazenar o endereço do último comando registrado.

Após tais definições, temos as definições das expressões regulares, conforme se percebe no código. Seguindo, na seção de regras, é possível perceber que existe um segundo modo de funcionamento do analisador léxico além do padrão, o modo “param”, que é usado para evitar que um comando seja capturado pelo analisador como um parâmetro. Ainda nesta seção, temos, em quase todas o envio de tokens para o analisador sintático, porém, em algumas regras, há também uma lógica para armazenar o nome do comando na lista de comandos e, os nomes dos parâmetros nas listas de parâmetros referentes a cada comando. São essas regras: “parameter”, na qual é capturado o parâmetro e armazenado na lista de parâmetros de seu respectivo comando; a segunda é “command”, na qual é capturado um comando e, este é armazenado na lista de comandos.

Aprofundando na regra “command”, vale destacar que existem condicionais, uma para o caso em que não há comando nenhum na lista, portanto não é necessário pular para o próximo espaço da lista, enquanto que, a outra condicional, contrária a essa, é satisfeita quando há ao menos um elemento na lista de comandos, o que significa que é necessário pular para o próximo espaço para adicionar o comando atual na lista.

Ademais, foi usada uma variável chamada “verifica_comando”, que tem como função verificar se foi passado algum comando na linha atual, caso não tenha, os parâmetros passados na linha não são armazenados em lugar nenhum.

sepcomm.y:

```
%{
#include <stdio.h>
#include <string.h>

int yylex(void);
int yyerror(char const *s);
}%

%union {
    char *string_value;
    float float_value;
    char char_value;
}

%token <string_value> DOUBLEPNTS
%token <string_value> PARAMETER
```

```

%token <string_value> COMMA
%token <string_value> COMMAND
%token <string_value> LINEBREAK
%token <string_value> COMMENT
%%

total      :      line
           |      total line
           ;

line       :      COMMAND DOUBLEPNTS parameters LINEBREAK
           |      DOUBLEPNTS parameters LINEBREAK {
                                                    printf("ERRO: não foi
passado nenhum comando nesta linha\n");
                                                    }
           |      COMMAND DOUBLEPNTS LINEBREAK
           |      COMMENT LINEBREAK
           ;

parameters :      parameters COMMA PARAMETER
           |      PARAMETER
           ;

%%

void main(){
    yyparse();
}

int yyerror (char const *s){
    fprintf (stderr, "%s\n", s);
}

```

Neste código, definem-se, inicialmente, tipos possíveis de retornos do analisador léxico para o sintático, que são “char”, “string” e “float”. Como tokens, o analisador sintático recebe aqueles que se vê no código acima e, suas produções são simples, compondo: “parameters” (define os parâmetros recebidos), “line” (define uma linha recebida) e “total” (define todo o conteúdo recebido).

Abaixo está a execução do parser para o input de exemplo do exercício. Nele estão impressas respostas do parser cujas linhas de códigos correspondentes estão comentadas, ou seja, para que o parser tenha output, é necessário descomentá-las, caso contrário ele apenas armazena os elementos em listas

conforme é pedido no exercício. Essas impressões são feitas apenas para apresentar o funcionamento do programa.

```
[pininsu@MiWiFi-RA72-srv arquivos_de_apoio_ativ_3]$ bash make.sh
[pininsu@MiWiFi-RA72-srv arquivos_de_apoio_ativ_3]$ ./sepcomm
# Arquivo de comandos e parâmetros para o exercício ex5.
comando_a:parâmetro_a1,parâmetro_a2,opção_a1, 324234, x3294549
----->comando atual: comando_a
----->parametro adicionado: parâmetro_a1
----->parametro adicionado: parâmetro_a2
----->parametro adicionado: opção_a1
----->parametro adicionado: 324234
----->parametro adicionado: x3294549
comando_b : opção_b1, opção_b2, parametro_xyz
----->comando anterior: comando_a
----->param 1: parâmetro_a1
----->comando atual: comando_b
----->parametro adicionado: opção_b1
----->parametro adicionado: opção_b2
----->parametro adicionado: parametro_xyz
comando_c :
----->comando anterior: comando_a
----->param 1: parâmetro_a1
----->comando anterior: comando_b
----->param 1: opção_b1
----->comando atual: comando_c
: parâmetro_inócuo_1, parâmetro_inútil_1
ERRO: não foi passado nenhum comando nesta linha
comando_d: parâmetro_d, 22, 1, opção_inexistente
----->comando anterior: comando_a
----->param 1: parâmetro_a1
----->comando anterior: comando_b
----->param 1: opção_b1
----->comando anterior: comando_c
----->param 1:
----->comando atual: comando_d
----->parametro adicionado: parâmetro_d
----->parametro adicionado: 22
----->parametro adicionado: 1
----->parametro adicionado: opção_inexistente
```