

UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
UNIOESTE - CAMPUS DE FOZ DO IGUAÇU
CENTRO DE ENGENHARIAS E CIÊNCIAS EXATAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Compiladores
Trabalho 2 - entryC

Juliet Rigoti
Victor Alicino
Professor(a) : Camile Frazao Bordini

2024

1 Funcionamento do Software

Para esse trabalho foram usadas as ferramentas Flex para o analisador léxico, juntamente com o Yacc para o analisador sintático. Para executar o software no terminal, é necessário abrir o terminal na pasta correspondente do software e escrever as seguintes instruções no prompt de comando:

- `yacc -d parser.y`
- `lex lexico.l`
- `gcc *.c`
- `./a.out`

Depois do último comando, o software perguntará se o usuário deseja entrar com o arquivo e os respectivos são `fonte1.txt` e `fonte2.txt`, em que o `fonte1.txt` é o arquivo que não contém nenhum erro de compilação e o `fonte2.txt` com erros léxicos e sintáticos.

1.1 Tokens reconhecidos pelo léxico

Nesta parte os tokens devem se localizar no arquivo com extensão `.l`.

```
/*Operadores aritmeticos*/
SUB "-"
SUM "+"
MULT "*"
DIV "/"
POW "**"
MOD "%"
/*Incremento e decremento*/
INCR "++"
DECR "--"
/*Atribuicao*/
ASSIGNMENT "="
/*Operadores de comparacao*/
COMPARATOR ("<" | ">" | "<=" | ">=" | "!=" | "==")

/*Operadores Logicos*/
AND "&&"
OR "||"
NOT ("~" | "!")
```

```

O_KEY "{"
C_KEY "}"
O_BRAC "["
C_BRAC "]"
O_PAR "("
C_PAR ")"
O_COMENT ("##*")
C_COMENT ("*##")
SEMICOLON ";"
COMMA ","

```

No código acima, é necessário a apresentação desses tokens como forma de quebra de blocos no código, por exemplo os lexemas O-KEY e C-KEY, no software são utilizados como abertura e fechamento para *if*'s e *for*'s. A mesma ideia de abertura e fechamento são usadas para o restante dos tokens.

```

INCLUDE ("#include" [ ]*<.\.h>)
LETTER [a-zA-Z]
STR (\".*\")
NUMBER [0-9]+
ID {LETTER}({LETTER}|{NUMBER})*
ERROID ({LETTER}|{NUMBER})("$"|"@"|{LETTER}|{NUMBER})+

```

O software é capaz de identificar palavras reservadas sem a necessidade de expressar como no código acima, estas palavras são *for*, *while*, *if*, *else*, *return*, *int*, *float*, *double*, *string*, *char* e *void*. Um trecho do código a seguir reflete como são tratadas no arquivo léxico.

```

(while) {
    inserts(HT, yytext, strlen(yytext), flag);
    strcpy(yylval.obj.name, (yytext));
    return WHILE;
}
(if) {
    inserts(HT, yytext, strlen(yytext), flag);
    strcpy(yylval.obj.name, (yytext));
    return IF;
}

```

Possui quatro tipos de erros que são mostrados na tela para o usuário e são os identificadores mal formulados, como é apresentado no trecho do código acima, é nomeado como ERROID e outro para erros que não são tratados, neste quesito é apresentando uma mensagem padrão para o usuário, informando que possui um

token não identificado e linha do código onde isso ocorreu.

```
{ID} {
    if(strlen(yytext) < 10){
        inserts(H, yytext, strlen(yytext), flag);
        strcpy(yylval.obj.name, (yytext));
        return ID;
    }
    else{
        fprintf(stderr, "ERRO: Tamanho maximo de identificador excedido
            na linha %d!\n", flag);
        strcpy(yylval.obj.name, (yytext));
    }
}
```

No trecho acima, o token ID detectará no código, o que encaixa como identificador e caso este token tiver mais de quinze caracteres, é apresentado ao usuário que não é permitido este tamanho.

1.2 Regras gramaticas pelo sintático

O que compõe a essência do software são as regras gramaticas que estão no arquivo com extensão .y. Como no léxico, o sintático possui uma estrutura para definições, regras e sub-rotinas. As definições é o bloco que contém as bibliotecas e os arquivos que compartilham informações no arquivo .l, que podem ser reconhecidas por possuir a palavra **extern**.

```
%{
    #include "hashTable.h"
    #include "tree.h"
    extern int yylex();
    extern int flag;
    extern HashTable *H, *HT;
    extern FILE *yyin;
    extern FILE *yyout;
    struct node* no;
    void yyerror();
    int flagError = 0;
}%
```

Antes da utilização devidamente dos tokens, eles devem ser convocados no arquivo sintático com a expressão **%token**, isso ocorre justamente pela questão de pilha do analisador sintático ascendente, para reconhecer o que é terminal e

não terminal.

```
%token <obj> SUB SUM MULT DIV ASSIGNMENT COMPARATOR INCR DECR
%token <obj> AND OR NOT
%token <obj> INT CHAR VOID FLOAT DOUBLE
%token <obj> INTEGER REAL STR
%token <obj> O_KEY O_BRAC O_PAR C_PAR C_BRAC C_KEY O_COMENT C_COMENT
        SEMICOLON COMMA INCLUDE PRINTF SCANF RETURN
%token <obj> WHILE FOR IF ELSE
%token <obj> ID
```

Os tokens apresentados acima fazem parte do reconhecimento da pilha. No entanto, neste software é necessário criar uma árvore sintática com as devidas regras gramaticas e tokens e uma parte importante para isso ocorrer é criar um union, que contém um vetor de char e uma variável para uma estrutura de lista encadeada. Por esta razão que os tokens possuem `<obj>`.

```
%union {
    struct {
        char name[MAX*2]; //
        int type;
        int category;
        char valueInt [MAX*2];
        char valueDouble [MAX*2];
        struct node* tr;
    } obj;
}
```

Com essa estrutura usa-se a variável *tr* para inserir cada não terminal em uma estrutura de árvore binaria.

```
program: headers main {
        $$tr = insertNode($1.tr, $2.tr, "program");
        no = $$tr;
};
```

Cada não terminal recebe o simbolo \$ e um respectivo valor de acordo com a ordem que está a gramatica. No exemplo acima, o não terminal *headers* recebe o valor 1 como parâmetro, por justamente ser a primeira variável a direita na gramatica.

Ao todo existem trinta e uma regras gramaticas que serão usadas para averiguar se existe conexão entre terminais e não terminais.

```
program: headers main
```

```

headers: headers headers | INCLUDE | /*empty*/
dataType: int | char | double | float | void
operator: + | - | * | / | ** | % | =
opLogical: && | ||
comparator: < | > | <= | >= | != | ==
opINCorDEC: ++ | --
PRorSC: printf | scanf
NUMorID: NUMBER | ID
NUMorEMP: NUMBER | /*empty*/
main: dataType ID ( args ) { content ret }
args: dataType * ID | dataType ID [] | dataType ID | /*empty*/
ret : return NUMorID ;
content: content cont | cont
cont: attSTATE | ifSTATE | forSTATE | whileSTATE |
      comentSTATE | PRorSC_STATE | /*empty*/
PRorSC_STATE: PRorSC ( STR bodyPRorSC ) ;
bodyPRorSC: , ID bodyPRorSC | , expCOND bodyPRorSC | /*empty*/
attSTATE: bodyATT ;
bodyATT: dataType ID | bodyATT , ID | bodyATT = NUMorID |
        attSTR | ID = NUMorID | error
attSTR: char ID [ NUMorEMP ] bodySTR
bodySTR: = STR | /*empty*/
ifSTATE: if ( expCOND ) bodyLOOP elseSTATE
expCOND: expCOND comparator expCOND | ( expCOND ) |
        expCOND operator expCOND | expCOND opLogical expCOND | NUMorID
elseSTATE : else { content } | else ifSTATE | /*empty*/
forSTATE: for ( forINIT ; expCOND ; forUpdate ) bodyLOOP
forINIT: bodyATT | /*empty*/
forUpdate: UpdateDF | /*empty*/
UpdateDF: ID opINCorDEC | UpdateDF , UpdateDF
whileSTATE: while ( expCOND ) bodyLOOP
bodyLOOP: { content }
comentSTATE: ### content ###

```

2 Funções da Linguagem entryC

A principal função que auxilia na análise são as ferramentas Flex e Yacc, uma vez que se utiliza a função 'yylex()' e um arquivo 'yyin' 'yyout' no arquivo .y. Isso permite que o software *yacc* interaja com o LEX.

Conforme mencionado no arquivo de referência (NIEMANN, 2015), 'yyin' é uma variável do tipo 'FILE*' que aponta para o arquivo de entrada. O LEX

define 'yyin' automaticamente. Se o programador atribuir um arquivo de entrada a 'yyin' na seção de funções auxiliares, então 'yyin' será definido para apontar para esse arquivo. Além disso, o 'yytext' é uma variável do tipo 'char*' que contém o lexema encontrado atualmente. A cada invocação da função 'yylex()', 'yytext' carrega um ponteiro para o lexema encontrado no fluxo de entrada por 'yylex()'. O valor de 'yytext' será substituído após a próxima invocação de 'yylex()'. A função 'yylex()' é do tipo de retorno 'int' e é definida automaticamente pelo LEX em 'lex.yy.c', mas não é chamada automaticamente. O programador deve invocar 'yylex()' na função principal. O LEX gera o código para a definição de 'yylex()' com base nas regras criadas no software.

No entanto apenas as ferramentas não são o suficiente para alcançar o resultado esperado, foi necessário a criação de dois tipos estruturados de dados, para o analisador léxico usa uma tabela Hash com encadeamento para a tabela de palavras reservadas e outra para de símbolos.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAM 127 // para tabela hash
#define MAX 100 // para vetor de char

struct cell {
    char name[MAX];
    int lineno;
    int len;
    int iVal; //variavel para quando for inteiro
    float fVal; //variavel para quando for float/double
    char cVal[MAX]; //usar tanto para char quanto para string
    int value; //variavel para guardar valores de expressao ou variavel
    int type; //para guardar se eh inteiro/float/char
    int category;
    struct cell *prox;
};

typedef struct {
    struct cell **table;
} HashTable;
```

No entanto, é necessário o complemento de funções para que as tabelas tenham conteúdo.

No trecho de código abaixo, é necessário que a tabela Hash seja limpa, o que quer dizer que atribui-se o valor **NULL** para que ela não possua conteúdo que atrapalhe na inserção de valores nas células de memória.

```
HashTable *initialization(){
    int i;
    HashTable *aux = (HashTable *)malloc(sizeof(HashTable)); // aloca o
        primeiro ponteiro
    aux->table = (struct cell **)malloc(TAM * sizeof(struct cell *));
    for (i = 0; i < TAM; i++) aux->table[i] = NULL;
    return aux;
}
```

No entanto, como possuí um vetor com encadeamento, precisasse de uma função que calcule o index de cada bloco desse mesmo vetor.

```
int hash(char *name){
    int sum = 0;
    for (int i = 0; name[i] != '\0'; i++){
        sum += name[i];
        sum *= 2;
    }
    return sum % TAM;
}
```

Ainda se faz necessário uma função de busca para ter certeza se o conteúdo que o LEX quer inserir já existe, caso exista esta função retornará o index do vetor que contém a informação.

```
int SearchToken(struct cell *aux, char *name){
    int index = hash(name);
    while (aux != NULL && strcmp(aux->name, name)) aux = aux->prox;

    if (aux == NULL)
        return -1;
    else
        return index;
}
```

A função principal que é chamada para inserir os tokens nas células do vetor.

```
void inserts(HashTable *h, char *name, int len, int line, int type, int
cat){
    int index = hash(name);
```



```

        printf("| TYPE_KEYWORD\t|");
        break;
case 6:
    printf("| TYPE_CHAR\t|");
    break;
case 7:
    printf("| TYPE_VOID\t|");
    break;
default:
    printf("          \t|");
    break;
}
switch (aux->category){
case 8:
    printf(" NUMBERS\t|");
    break;
case 9:
    printf(" VARIABLE\t|");
    break;
case 10:
    printf(" DATA_TYPE\t|");
    break;
case 11:
    printf(" CHARACTER\t|");
    break;
case 12:
    printf(" LIBRARIES\t|");
    break;
case 13:
    printf(" OPERATOR\t|");
    break;
case 14:
    printf(" PARAMETER\t|");
    break;
case 15:
    printf(" FUNCTION\t|");
    break;
default:
    printf("          \t|");
    break;
}

switch (aux->type){

```

```

        case 1:
            printf("%-10d\t|", aux->iVal);
            break;
        case 2:
            printf("%-10.2f\t|", aux->fVal);
        case 3:
            printf("%-10s\t|", aux->cVal);
        default:
            printf("          \t|");
            break;
    }

    printf("%2d", aux->lineno);
    aux = aux->prox;
    while (aux != NULL){
        printf(" -> %2d", aux->lineno);
        aux = aux->prox;
    }
    printf("\t\n");
}
}
finalization(H);
}

```

Outra estrutura de dados que foi utilizada é a árvore binária, foi utilizada para a criação da árvore sintática.

```

struct node{ /*arvore binaria*/
    struct node *left;
    struct node *right;
    char *token;
};

```

Utilizada no analisador sintático, pois é a que insere as variáveis não terminais na árvore binária.

```

struct node *insertNode(struct node *L, struct node *R, char *token){
    struct node *r = (struct node*) malloc (sizeof(struct node));
    char *str = (char *) malloc (sizeof(char));
    strcpy(str, token);
    r->left = L;
    r->right = R;
    r->token = str;
    return(r);
}

```

}

Abaixo é apresentada a função que mostra para o usuário em arquivo *.txt* o resultado da árvore sintática do arquivo fonte.

```
void printAux(FILE* of, struct node* node, int n, int child){
    int i;
    for(i = 0; i < n; i++){
        if(i == n-1 && child == 0){
            fprintf(of, " \u251c");
        }
        else if(i == n-1 && child == 1){
            fprintf(of, " \u2515");
        }
        else{
            fprintf(of, " | ");
        }
    }

    fprintf(of,"%s\n", node->token);

    if(node->left != NULL)
        printAux(of, node->left, n+1, 0);
    if(node->right != NULL)
        printAux(of, node->right, n+1, 1);
}

void printTree(FILE* of, struct node* node){
    fprintf(of,"| | | ARVORE SINTATICA | | |\n");
    printAux(of, node, 0, 0);
}
```

3 Construção do Software

O software não possui interface gráfica, então para utiliza-lo é necessário um terminal. Para iniciar o programa basta iniciar o executável resultante da compilação do código-fonte apresentada no capítulo 1

Figura 1: Instruções no terminal

```
(base) juliet@juliet-Asus:~/Documents/CC/Compiladores/analizadorSintatico-Compiladores$ yacc -d parser.y
parser.y: warning: 147 shift/reduce conflicts [-Wconflicts-sr]
parser.y: warning: 7 reduce/reduce conflicts [-Wconflicts-rr]
parser.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
(base) juliet@juliet-Asus:~/Documents/CC/Compiladores/analizadorSintatico-Compiladores$ lex lexico.l
(base) juliet@juliet-Asus:~/Documents/CC/Compiladores/analizadorSintatico-Compiladores$ gcc *.c
(base) juliet@juliet-Asus:~/Documents/CC/Compiladores/analizadorSintatico-Compiladores$ ./a.out
Deseja entrar com um arquivo? 1-SIM/2-NAO
>> █
```

Fonte: De autoria própria.

A seguir o usuário deverá entrar com o arquivo texto sem erros léxicos e sintáticos, que é o *fonte1.txt*.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(char argv[]){
    int ix, iix;
    float ffx = 10.51;
    double dx, ddx;
    char cx, ccx;
    void vx, vvix;

    ix = 2;

    int lx = 3 - 2;
    lx = 6 * 35;
    return 1;
}
```

Em seguida quando se encerra o software, as árvores sintáticas são geradas no arquivo texto.

Para o arquivo que contém erros léxicos e sintáticos, é gerado a tabela de símbolos e reservadas, mas não a árvore sintática.

```
#include <stdio.h>
int main (){
    float x = 1, y = 1, 9XX;
    variavelmuitogrande2023;

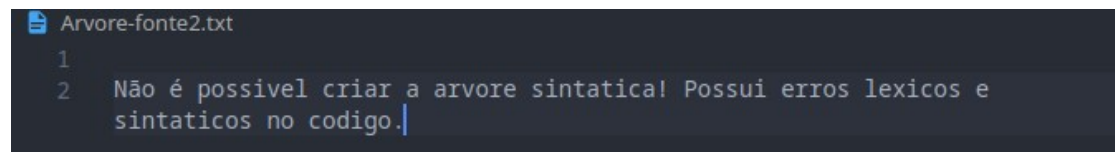
    if(x == 0) y = 2;
    else y = 1;
    for (int i = 0; ;i--) printf("testee\n")
}
```

```
    return sum;  
}
```

Em seguida será mostrado no terminal a tabela de simbolo e palavras reservadas.

No arquivo texto que possui erro, não é criado uma árvore sintática então caso o usuário não tenha conhecimento desse detalhe, é mostrado uma mensagem no arquivo texto informando.

Figura 2: Informação na árvore sintática do arquivo fonte2.txt



```
Arvore-fonte2.txt  
1  
2  Não é possível criar a árvore sintática! Possui erros lexicos e  
   sintaticos no código. |
```

Fonte: De autoria própria.

4 Análise Semântica

Na última etapa do processamento do código em si temos a análise semântica que é a verificação de consistência do código, ou seja, se as variáveis estão sendo usadas corretamente, se os tipos de dados estão sendo usados corretamente, se as variáveis foram declaradas antes de serem usadas, entre outras verificações.

4.1 Regras Semânticas

As regras semânticas são regras que definem o que é permitido e o que não é permitido em um programa de computador. O compilador entryC possui 9 regras semânticas, que são:

- args
- PRorSC_STATE
- bodyPRorSC
- attSTATE
- bodyATT
- stateATTEXP
- attSTR

- expCOND

Decompondo uma por uma as regras semânticas, temos:

4.2 args

Essa regra confere a sequência de caracteres que segue a declaração da função principal main, exemplos que essa regra confere são "int main(char *argv)", ou "int main(char argv[])".

A verificação pode se dar através das seguintes regras:

4.2.1 dataType MULT ID

```
dataType MULT ID {
    struct cell *temp = SearchParser(H, $3.name);
    if(temp->type == 4){
        setType(H, $3.name, $1.type);
        setCategory(H, $3.name, 9);
        $$tr = insertNode($1.tr, $3.tr, "dataType *ID");
    } else {
        printf("Erro Semantico: Tipo de variavel redefinida na linha
              %d\n", flag);
    }
}
```

Aqui é verificado se o tipo de dado é um ponteiro (exemplo: char *argv); O ID será buscado na tabela de símbolos e seu tipo será checado, se o tipo for 4 (que significa "dado desconhecido") a cadeia é validada e seu tipo é definido com o tipo de dataType, do contrário a cadeia é inválida e um erro semântico é gerado.

4.2.2 dataType ID O_BRAC C_BRAC

```
dataType ID O_BRAC C_BRAC {
    struct cell *temp = SearchParser(H, $2.name);
    if(temp->type == 4){
        setType(H, $2.name, $1.type);
        setCategory(H, $2.name, 9);
        $$tr = insertNode($1.tr, $2.tr, "dataType ID[]");
    } else {
        printf("Erro Semantico: Tipo de variavel redefinida na linha
              %d\n", flag);
    }
}
```

```
}
```

Aqui é verificado se o tipo de dado é um vetor (exemplo: `char argv[]`); O *ID* será buscado na tabela de símbolos e seu tipo será checado, se o tipo for 4 (que significa "dado desconhecido") a cadeia é validada e seu tipo é definido com o tipo de `dataType`, do contrário a cadeia é inválida e um erro semântico é gerado.

4.2.3 dataType ID

```
dataType ID {  
    struct cell *temp = SearchParser(H, $2.name);  
    if(temp->type == 4){  
        setType(H, $2.name, $1.type);  
        setCategory(H, $2.name, 14);  
        $$tr = insertNode($1.tr, $2.tr, "dataType ID");  
    } else {  
        printf("Erro Semantico: Tipo de variavel redefinida na linha  
              %d\n", flag);  
    }  
}
```

Aqui é verificado se o tipo de dado é um tipo primitivo (exemplo: `int x`); O *ID* será buscado na tabela de símbolos e seu tipo será checado, se o tipo for 4 (que significa "dado desconhecido") a cadeia é validada e seu tipo é definido com o tipo de `dataType`, do contrário a cadeia é inválida e um erro semântico é gerado.

4.3 attSTATE e bodyATT

Essas regras conferem a sequência de caracteres que segue a declaração de uma atribuição de valor a uma variável, exemplos que essas regras conferem são "`int x`", ou "`int x, y`".

```
attSTATE: bodyATT SEMICOLON {  
    $$tr = insertNode($1.tr, NULL, "bodyATT;");  
};
```

4.3.1 dataType ID

```
bodyATT: dataType ID {  
    struct cell *temp = SearchParser(H, $2.name);  
    if(temp->type == 4){
```



```

        setType(H, $2.name, $1.type);
        setCategory(H, $2.name, 9);
        $$ .type = $1.type;
        strcpy($$.name, $2.name);
        $$ .tr = insertNode($1.tr, NULL, "dataType ID");
    }
    else {
        printf("Erro Semantico: Tipo de variavel redefinida na linha
               %d\n", flag);
    }
}

```

A regra `bodyATT` confere se a linha de código é uma declaração de variável, checando se o corpo segue o padrão "dataType ID" (exemplo: `int x`). O *ID* será buscado na tabela de símbolos e seu tipo será checado, se o tipo for 4 (que significa "dado desconhecido") a cadeia é validada e seu tipo é definido com o tipo de `dataType`, do contrário a cadeia é inválida e um erro semântico é gerado.

4.4 bodyATT COMMA ID

```

bodyATT COMMA ID {
    struct cell *temp = SearchParser(H, $3.name);
    if(temp->type == 4){
        setCategory(H, $3.name, 9);
        setType(H, $3.name, $1.type);
        $$ .type = $1.type;
        strcpy($$.name, $3.name);
        $$ .tr = insertNode($1.tr, NULL, "bodyATT, ID");
    }
    else {
        printf("Erro Semantico: Tipo de variavel redefinida na linha
               %d\n", flag);
    }
}

```

Essa regra é similar a regra `dataType ID`, porém ela checa por mais de uma variável declarada na mesma linha (exemplo: `int x, y`). Se o *ID* for encontrado na tabela de símbolos e seu tipo for 4, toda a cadeia é validada e seu tipo é definido com o tipo de `dataType`,

4.5 attSTR

Por fim é feita a verificação de atribuição de uma string a uma variável. Mais detalhes sobre a regra attSTR mais abaixo.

4.6 stateATTEXP

A regra stateATTEXP é responsável por verificar se a atribuição de uma variável é válida.

4.6.1 dataType ID ASSIGNMENT expCOND SEMICOLON

```
dataType ID ASSIGNMENT expCOND SEMICOLON{

    struct cell *temp = SearchParser(H, $2.name);
    if(temp->type == 4){
        setType(H, $2.name, $1.type);
        setCategory(H, $2.name, 9);
        if($1.type == $4.type){
            $$type = $4.type;
            getValue(H, $2.name, $4.name);
            $$tr = insertNode(NULL, $4.tr, "dataType ID = expCOND;");
            $$tr = insertNode($1.tr, $2.tr, "dataType ID");

        } else {
            printf("Erro Semantico: Tipo de dados imcopativeis na linha
                    %d\n", flag);
        }

    } else {
        printf("Erro Semantico: Tipo de variavel redefinida na linha
                %d\n", flag);
    }
}
```

Essa regra confere se a atribuição de uma variável é válida, checando se o corpo segue o padrão "dataType ID = expCOND;"(exemplo: `int x = 10`, `int x = (10 + 5)`). O ID será buscado na tabela de símbolos e seu tipo será checado, se o tipo for 4 (que significa "dado desconhecido") a cadeia é validada e seu tipo é definido com o tipo de dataType, do contrário a cadeia é inválida e um erro semântico é gerado.

4.6.2 ID ASSIGNMENT expCOND SEMICOLON

```
ID ASSIGNMENT expCOND SEMICOLON{
    struct cell *temp = SearchParser(H, $1.name);
    if(temp->type == $3.type){
        $$ .type = $3.type;
        setType(H, temp->name, $3.type);
        getValue(H, $1.name, $3.name);
        $$ .tr = insertNode($1.tr, $3.tr, "ID = expCOND;");
    } else if(temp->type == 4){
        printf("Erro Semantico: Variavel nao foi declarada na linha
              %d\n", flag);
    }
    else {
        printf("Erro Semantico: Tipo de dados imcopativeis na linha
              %d\n", flag);
    }
}
```

A atribuição de variável verificada nessa regra é similar a regra `dataType ID ASSIGNMENT expCOND SEMICOLON`, porém sem a definição de tipo, uma atribuição sem declaração de tipo (exemplo: `x = 10, x = (10 + 5)`).

4.7 attSTR

Por fim é feita a verificação de atribuição de uma string a uma variável.

4.7.1 attSTR: dataType ID O_BRAC NUMorEMP C_BRAC bodySTR

```
dataType ID O_BRAC NUMorEMP C_BRAC bodySTR{
    struct cell *temp = SearchParser(H, $1.name);
    if(temp->type == 4){
        setType(H, $2.name, $1.type);
        setCategory(H, $2.name, 9);
        struct node *a = insertNode($1.tr, $2.tr, $2.name);
        struct node *b = insertNode($4.tr, $6.tr, "NUMorEMP
              bodySTR");
        $$ .tr = insertNode(a, b, "CHAR ID NUMorEMP bodySTR ");
    }
    else {
        printf("Erro Semantico: Tipo de variavel redefinida na
              linha %d\n", flag);
    }
}
```

```
    }  
};
```

Essa regra confere se a atribuição de uma string a uma variável é válida, checando se o corpo segue o padrão "dataType ID O_BRAC NUMorEMP C_BRAC bodySTR", exemplo (exemplo: `char x[10] = "Hello, World!"` ou `char x[] = "Hello, World!"`).

4.8 expCOND

Essa regra abrange as expressões matemáticas soma, subtração, multiplicação e divisão, também checa se a expressão possui parênteses.

4.8.1 expCOND SUM expCOND

```
expCOND SUM expCOND {  
    if($1.type == $3.type){  
        $$tr = insertNode($1.tr, $2.tr, "expCOND + expCOND");  
        $$type = $1.type;  
        calculateTest(H, $1.name, $3.name, "+", $$name);  
        //sprintf( $$name, "%d", $$valueInt);  
        inserts(H, $$name, strlen($$name), flag, 1, 8);  
    }  
    else {  
        printf("Erro Semantico: Tipo incompativel na linha %d\n",  
            flag);  
        $$type = 4;  
    }  
}
```

Essa regra confere se a soma matemática será válida, para ser válida ambos os operadores precisam ser do mesmo tipo, se ambos possuem o mesmo tipo, nessa regra não é necessario verificar o tipo de variavel, a conta será executada, é na função calculateTest que verifica o tipo de dado.

As demais regras de expCOND são similares a regra expCOND SUM expCOND, porém com operadores diferentes.

5 Update na HashTable

Com a análise semântica, a HashTable precisou ser atualizada com funções auxiliares, são elas:

- getValue

- calculateFloat
- calculateInt

5.1 getValue

A função `getValue` é responsável por obter o valor de uma variável na HashTable com base no seu nome e o operador especificado.

```
void getValue(HashTable *h, char *name, char *op){
    int index = hash(name);
    int index2 = hash(op);
    struct cell *aux = h->table[index];
    struct cell *aux2 = h->table[index2];

    switch (aux2->type){
    case 1:
        aux->iVal = atoi(aux2->name);
        break;
    case 2:
        aux->fVal = atof(aux2->name);
        break;
    case 3: case 6:
        strcpy(aux->cVal, aux2->name);
        break;
    default:
        break;
    }
}
```

5.2 calculateFloat

A função `calculateTest` calcula um valor a partir de dois valores da HashTable e o operador especificado.

```
void calculateTest(HashTable *h, char *name, char *name2, char*
operator, char *destino){
    int index = hash(name);
    int index2 = hash(name2);
    struct cell *aux = h->table[index];
    struct cell *aux2 = h->table[index2];
```

```

if (strcmp(operator, "+") == 0){
    if(aux->type == 1){
        sprintf(destino, "%d", aux->iVal + aux2->iVal);
    } else {
        sprintf(destino, "%.2f", aux->fVal + aux2->fVal);
    }
}
else if (strcmp(operator, "-") == 0){
    if(aux->type == 1){
        sprintf(destino, "%d", aux->iVal - aux2->iVal);
    } else {
        sprintf(destino, "%.2f", aux->fVal - aux2->fVal);
    }
}
else if (strcmp(operator, "*") == 0){
    if(aux->type == 1){
        sprintf(destino, "%d", aux->iVal * aux2->iVal);
    } else {
        sprintf(destino, "%.2f", aux->fVal * aux2->fVal);
    }
}
else if (strcmp(operator, "/") == 0){
    if(aux2->iVal != 0 && aux->iVal != 0 ){
        if(aux->type == 1){
            sprintf(destino, "%d", aux->iVal / aux2->iVal);
        } else {
            sprintf(destino, "%.2f", aux->fVal / aux2->fVal);
        }
    }
    else printf("No    possivel efetuar a operao de diviso\n");
}
}

```

6 Referências

NIEMANN, T. *Lex & yacc tutorial*. [S.l.]: epaperpress. Accesible en: [http://epaperpress.com/lexandyacc/download ...](http://epaperpress.com/lexandyacc/download...), 2015. 6