# Trabalho de Tradutores

Frederico Dib - 15/0125925

Profª. Cláudia Nalon

1/2020

## 1 Objetivos

A disciplina de tradutores ministrada pela professora Cláudia Nalon tem como seu principal objetivo o estudo de como funciona um tradutor de uma linguagem de programação. Este trabalho[1] consiste na implementação de um tradutor que deve traduzir uma liguagem de programação simplificada para um "Código de Três Endereços"[2]. Essa implementação acontecerá em 5 etapas, sendo elas: "Escolha do tema do projeto", "Implementação do analisador léxico", "Implementação do analisador sintático", "Implementação do analisador semântico", "Implementação do gerador de código intermediário".

## 2 Motivação e Proposta

Atualmente, com avanço das linguagens de programação de alto nível é destacado uma forte facilidade em manipulação de strings. Em Javascript[3], por exemplo, é possível declarar strings de tamanho dinâmico e operar sobre elas operações de concatenação por meio do símbolo '+'.

Essa facilidade na manipulação de strings é muito útil no desenvolvimento de aplicativos móveis, pois é uma forma eficaz de juntar dados vindo do backend por meio de um arquivo .json e strings pré prontas que serão mostradas para o usuário.

Em linguagens de mais baixo nível como C e assembly esse tipo de operação não é possível de forma nativa, para isso o desenvolvedor necessitaria de criar funções ou utilizar bibliotecas que implementam esse funcionamento. A proposta desse trabalho seria criar um tradutor de uma versão simplificada da linguagem C com as funcionalidades mencionadas. Por meio dessa linguagem, seria possível criar códigos como:

```
1 float test(int x, float y) {
2   float value;
3   value = x + y * x + 5.5 + x * 10;
```

[1] https://aprender.ead.unb.br/mod/page/view.php?id=5350
[2] https://github.com/lhsantos/tac/blob/master/doc/tac.pdf
[3] https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/First$_s$teps/*Strings*

```
4   return (value);
5 }
6
7 int main() {
8   /* Isso    um comentario */
9
10  string string1 = "Nice to meet you";
11  string string2;
12  string result;
13  float functionReturn;
14
15  string2 = "do you like blueberries?";
16  result = string1 + ", " + string2 + "\n";
17  print(result);
18
19  functionReturn = test(4, 5.5);
20  print(functionReturn);
21  print("\n");
22
23  if (functionReturn >= 20 && functionReturn <= 100) {
24    print("entre 20 e 100\n");
25  } else {
26    if (functionReturn < 20) {
27      print("menor que 20\n");
28    }
29    if (functionReturn > 100) {
30      print("maior que 100\n");
31    }
32  }
33
34  while (functionReturn < 1000) {
35    functionReturn = functionReturn + 1;
36  }
37
38  print(functionReturn);
39  print("\n");
40
41  int scanNumero;
42  scan(scanNumero, int);
43  print(scanNumero);
44  print("\n");
45
46  return(0);
47 }
```

## 3 Funcionamento do analisador léxico

Análise léxica é a primeira fase de um compilador, na qual um analisador léxico lê o fluxo de caracteres do código fonte e os agrupa em sequências de lexemas [1]. Um lexema é uma unidade básica de significado para uma linguagem. Dessa forma, o analisador léxico utiliza a gramática para identificar lexemas da linguagem. Caso identifique algo

que não faça parte da linguagem, o analisador deve informar a localização da ocorrência.

Nessa implementação primeiro é realizado um parser no código feito pelo programa Flex[4], separando os elementos do código em tokens, sendo eles:

1. ID -> Variáveis e declarações de funções.
2. NUM -> Token numérico.
3. OP -> Token referente a operações aritméticas.
4. COND -> Token referente a operações condicionais.
5. STATEMENT -> if, while, else, return, print, scan.
6. TYPE -> int, float, string.
7. SEP -> Vírgula, parenteses, chaves...

À medida que o parser for sendo realizado o programa imprimirá no terminal os tokens encontrados, seus tipos e valores respectivamente.

### 3.1 Funções adicionadas

Além do flex, o analisador léxico criado conta com as seguintes funções e estrutura de dados para funcionar:

1. Uma variável chamada "line"que armazena a linha atual e uma chamada "word-position"que armazena a posição da palavra dentro dessa linha.
2. Uma função "printError"que imprime no terminal os erros encontrados e encerra o programa.

### 3.2 Tratamento de Erros Léxicos

Ao identificar lexemas que não pertencem à linguagem, o analisador implementado imprime no terminal o lexama incorreto, junto da linha e da posição da palavra. O analisador reconhece os seguintes erros:

1. caracteres não permitidos.

## 4 Gramática

A gramática implementada no analisador léxico sofreu algumas mudanças em relação à apresentada na Escolha do Tema. A mudança foi a adição do tipo INT (inteiro) e DEC (decimal), que são tipos de números restritos. O tipo NUM representa qualquer valor numérico, seja inteiro ou decimal.

[4]https://westes.github.io/flex/manual/

1. *initializer* → *global-list*
2. *global-list* → *global-list var* | *var* | *global-list func* | *func* | *comment* | *global-list comment*
3. *comment* → /[∗]. ∗ [∗]/
4. *var* → *string* **ID**; | *string* **ID** = **STRING**; | *int* **ID**; |
   *int* **ID** = **INT**; | *float* **ID**; |
   *float* **ID** = **DEC**;
5. *func* → *string* **ID** ( *params-list* ) { *content-list* } |
   *int* **ID** ( *params-list* ) { *content-list* } |
   *float* **ID** ( *params-list* ) { *content-list* }
6. *params-list* → *params* | $\varepsilon$
7. *params* → *params*, *param* | *param*
8. *param* → *int* **ID** | *float* **ID** | *string* **ID**
9. *content-list* → *content-list content* | $\varepsilon$
10. *content* → *var* | *add-value* | *comand* | *print* | *scan* | *return* | *call-func*
11. *add-value* → **ID** = *expression*
12. *expression* → **ID** | **STRING** | **INT** | **DEC** |
    *expression* **OP** *expression*
13. *comand* → *comand-if* | *comand-ifelse* |*comand-while*
14. *comand-if* → *if* ( *condition* ) { *content-list* }
15. *comand-ifelse* → *if* ( *condition* ) { *content-list* } *else* { *content-list* }
16. *comand-while* → *while* ( *condition* ) { *content-list* }
17. *condition* → **ID COND ID** | **ID COND NUM** |
    **NUM COND ID** | **NUM COND NUM**
18. *print* → *print*(**ID**) | *print*(**NUM**) | *print*(**STRING**)
19. *scan* → *scan*(**ID**, **NUM**) | *scan*(**ID**, **STRING**)
20. *return* → *return*(**ID**) | *return*(**NUM**) | *return*(**STRING**)
21. *call-func* → **ID**(*call-func-params*) |
    **ID** = **ID**(*call-func-params*)
22. *call-func-params* → *call-func-params*, **ID** | **ID** | $\varepsilon$

**ID** = *letter* (*letter*|*digit*)$^*$

**INT** = $digit^+$

**DEC** = $digit^+.digit^+$

**NUM** = **INT** | **DEC**

**STRING** = "$(all\ characters)^*$"

**OP** = + | **-** | ***** | /

**COND** = > | >= | < | <= | == | **!=**

$letter = a \mid ... \mid z \mid A \mid ... \mid Z$

$digit = 0 \mid ... \mid 9$

Palavras reservadas: **print scan int float string if else while return**
Símbolos: = |, |; |( |) |{ |} |" |+ |- |* |/ |< |<= |> |>= |== |**!**= |

# 5 Sintático

Para realizar a análise sintática da linguagem descrita, utilizei o programa Bison[5] da linguagem C, junto do programa Flex, previamente usada na construção do analisador léxico. Após a passagem do Flex, é enviado ao Bison uma lista de tokens, em que, por meio delas, é realizado uma análise sintática da linguagem. A medida que essa análise for sendo realizado, será montado uma árvore sintática que será utilizada nos passos posteriores.

## 5.1 Tratamento de Erros Sintáticos

Para lidar com o tratamento de erros no analisador sintático, foi utilizado, por questões de eficiência e qualidade, a função "yyerror", implementada pelo próprio Bison. Dentro dessa função, o erro é impresso no console e o programma é encerrado por meio de um "exit(1)". Para deixar os erros mais descritivos, foi adicionado no topo do arquivo os seguintes enunciados: "%error-verbose", "%debug", "%locations".

# 6 Árvore Sintática

Durante a execução da analise sintática a árvore sintática vai sendo construída, possuindo a seguinte estrutura:

```
1 struct node {
2   char *value;
3   char node_type;
4   int type;
5   char *addr;
```

[5] https://www.gnu.org/software/bison/manual/

```
6   struct node *node1;
7   struct node *node2;
8   struct node *node3;
9 };
```

O campo "value"se refere ao valor daquele nó, o que poderia ser o nome de uma variável, nome de uma função, um número, entre outros. O campo "type"se refere à tipagem de dado daquele nó caso tenha, sendo "0"um valor inteiro, "1"um valor decimal, "2"uma string, "3"tipo não definido. Posteriormente essa tipagem será usada para análise semântica. O campo "node_type"Se refere ao tipo de nó, especificando que tipo de nó está sendo definido, entre eles temos os seguinte valores:

1. F - Declaração de função;
2. v - Declaração de variável;
3. V - Variável;
4. P - Parâmetros de função;
5. R - Regra de ligação;
6. C - IF ou IF-ELSE;
7. c - Expressões condicionais;
8. W - While;
9. I - Inteiro;
10. D - Decimal;
11. S - String;
12. B - Simbolo;
13. E - Expressão aritmética;
14. A - Atribuição de valor;
15. L - Chamada de função;

Após a criação da árvore será impresso no terminal uma representação da mesma. Essa é a saída referente ao código de exemplo dado na sessão "Motivação e Proposta":

```
########## Arvore Sintatica ##########
<Função> test, tipo: 1
INICIO PARAMETROS
-<Parametro> x, tipo: 0
-<Parametro> y, tipo: 1
```

```
FIM PARAMETROS
INICIO FUNÇÃO -  test
-<Declaração de Variavel> value, tipo: 1, addr: 4
-<Atribuição> (tipo: 1) value =
--<EXPRESSION> tipo: 1
--<Variavel> x, tipo: 0, addr: 2
---<Simbolo> +
---<EXPRESSION> tipo: 1
---<Variavel> y, tipo: 1, addr: 3
----<Simbolo> *
----<EXPRESSION> tipo: 1
----<Variavel> x, tipo: 0, addr: 2
-----<Simbolo> +
-----<Decimal> 5.5
-<RETURN> tipo: 1
--<Variavel> value, tipo: 1, addr: 4
FIM FUNÇÃO - test

<Função> main, tipo: 0
INICIO PARAMETROS
FIM PARAMETROS
INICIO FUNÇÃO -  main
-<Declaração de Variavel> string1, tipo: 2, addr: 6
--<Atribuição> (tipo: 2) string1 =
---<String> "Nice to meet you"
-<Declaração de Variavel> string2, tipo: 2, addr: 7
-<Declaração de Variavel> result, tipo: 2, addr: 8
-<Declaração de Variavel> functionReturn, tipo: 1, addr: 9
-<Atribuição> (tipo: 2) string2 =
--<String> "do you like blueberries?"
-<Atribuição> (tipo: 2) result =
--<EXPRESSION> tipo: 2
--<Variavel> string1, tipo: 2, addr: 6
---<Simbolo> +
---<EXPRESSION> tipo: 2
---<String> ", "
----<Simbolo> +
----<Variavel> string2, tipo: 2, addr: 7
-<PRINT>
--<Variavel> result, tipo: 2, addr: 8
-<Atribuição> (tipo: 3) functionReturn =
--<CHAMA FUNÇÃO> test
--INICIO PARAMETROS
---<Inteiro> 4
```

```
---<Decimal> 5.5
--FIM PARAMETROS
-<PRINT>
--<Variavel> functionReturn, tipo: 1, addr: 9
-<IF>
-IF - INICIO CONDIÇÃO
--<CONDIÇÃO>
--<Variavel> functionReturn, tipo: 1, addr: 9
--<Simbolo> CGE
--<Inteiro> 20
-IF - FIM CONDIÇÃO
-IF - INICIO BLOCO
--<PRINT>
---<String> "maior ou igual"
-IF - FIM BLOCO
ELSE - INICIO BLOCO
--<PRINT>
---<String> "menor"
-ELSE - FIM BLOCO
-<WHILE>
-WHILE - INICIO CONDIÇÃO
--<CONDIÇÃO>
--<Variavel> functionReturn, tipo: 1, addr: 9
--<Simbolo> CLT
--<Inteiro> 1000
-WHILE - FIM CONDIÇÃO
-WHILE - INICIO BLOCO
--<Atribuição> (tipo: 1) functionReturn =
---<EXPRESSION> tipo: 1
---<Variavel> functionReturn, tipo: 1, addr: 9
----<Simbolo> +
----<Inteiro> 1
-WHILE - FIM BLOCO
-<PRINT>
--<Variavel> functionReturn, tipo: 1, addr: 9
-<Declaração de Variavel> scanNumero, tipo: 0, addr: 10
-<SCAN> tipo: 0
--<Variavel> scanNumero, tipo: 0, addr: 10
-<RETURN> tipo: 0
--<Inteiro> 0
FIM FUNÇÃO - main
```

# 7 Tabela de Símbolos

Durante a execução da analise sintática a tabela de símbolos vai sendo construída, possuindo a seguinte estrutura:

```
struct scope {
  char *scope_name;
  int type; // 0 = int, 1 = float, 2 = string
  struct scope *next;
};

struct symbol_node {
  char *key; // hash key -> scope_name + @ + id
  char *id; // vari vel
  char *scope_name; // nome do scopo
  int type; // 0 = int, 1 = float, 2 = string
  char *addr;
  UT_hash_handle hh;
};
```

A struct "scope"é usada para a implementação da pilha de escopos sinalizando em qual escopo está sendo considerado no momento. O compo "scope_name"identifica qual é aquele escopo, sendo geralmente o nome da função. O campo type se refere ao tipo da função, que será usado no analisador semântico no proximo passo.

A tabela de símbolos tem a função de identificar os IDs declarados, levando em conta o escopo que aquele ID foi inicializado e o tipo do mesmo. Para implementar essa tabela foi utilizado a estrutura "Hash Table"através da biblioteca UT Hash [2] junto da struct "symbol_node"para recuperar com mais facilidade e eficiência cada um desses símbolos. O campo "id"se refere ao nome da variável. O campo "scope_name"se refere ao escopo ao qual foi declarado aquela variável, permitindo que tenha variáveis com mesmo nome em diferentes escopos. O campo "key"é a chave para se recuperar essa variável, essa campo é construído por meio da concatenação (escopo da função) + "@"+ (nome da variável).

Após o fim da execução a tabela de símbolos é impressa no terminal. Esta é a tabela de símbolos referente ao exemplo de código mostrado na sessão "Motivação e Proposta":

```
########## Tabela de Símbolos ##########
Simbolo: test, Tipo: 1, Escopo: Global, key: @test, addr: $1
Simbolo: x, Tipo: 0, Escopo: test, key: test@x, addr: #0
Simbolo: y, Tipo: 1, Escopo: test, key: test@y, addr: #1
Simbolo: value, Tipo: 1, Escopo: test, key: test@value, addr: $2
Simbolo: main, Tipo: 0, Escopo: Global, key: @main, addr: $14
Simbolo: string1, Tipo: 2, Escopo: main, key: main@string1, addr: $15
Simbolo: string2, Tipo: 2, Escopo: main, key: main@string2, addr: $17
Simbolo: result, Tipo: 2, Escopo: main, key: main@result, addr: $18
Simbolo: functionReturn, Tipo: 1, Escopo: main, key: main@functionReturn, addr: $19
Simbolo: scanNumero, Tipo: 0, Escopo: main, key: main@scanNumero, addr: $73
```

# 8 Semântica

O tipo de variável "int"se remete a valores do tipo inteiro, o tipo de variável "float"se remete a valores do tipo real, o tipo de variável "string"se remete a um vetor de caracteres. As expressões aritméticas(previsto na regra 11 e 12) permitem realizar operações com nomes, números e strings. A semântica da linguagem deve garantir que strings e nomes do tipo string façam operações somente entre si da mesma forma que números apenas opere com outros números e nomes do tipo int e float.

No caso de uma operação int com float o inteiro deve ser convertido para float. No caso de operação entre duas strings, o único operador permitido será o operador "+", os outros operadores não serão permitidos nesse contexto. No caso. de. int e float serão aceito os operadores "+", -", "*"e "/".

## 8.1 Tratamento de Erros Semânticos

O analisador semântico da linguagem reconhece os seguintes erros:

1. Utilização de uma variável não declarada.
2. Utilização de qualquer símbolo diferente de "+"(concatenação) em uma operação entre strings.
3. Qualquer operação entre uma string e uma variável/valor do tipo inteiro ou float.
4. Atribuição de um valor decimal a uma variável de inteiro.
5. Atribuição de uma string a uma variável float ou inteiro.
6. Atribuição de um valor ou variável inteira ou float para uma variável do tipo string.
7. Atribuição a uma variável um retorno de função de um tipo diferente da mesma.
8. Retorno da função ser de um tipo diferente da mesma.
9. Passagem de parâmetros de tipo diferentes.
10. O tipo do scan ser de um tipo diferente do ID.
11. O código não possuir uma função main.
12. A função não possui return.
13. Variáveis ou funções com o mesmo nome.

Para identificar os erros, durante o parser do Bison é analisado o valor "type"correspondente ao nó da instrução atual da arvore sintática, junto da tabela de símbolo previamente construída. Ao encontrar algum dos erros citados acima, é chamado a função "print_semantic_error"que imprimirá o erro e encerrará o programa.

O escopo da variável é controlado pela tabela de símbolos, que armazena a variável, o tipo junto ao escopo equivalente.

A conversão de tipo é usada somente em expressões aritméticas. Ao ocorrer alguma operação entre um valor/variável do tipo inteiro com um do tipo float, o resultado da expressão é convertido para float.

## 8.2 Exemplos de erros

Exemplo 1:

```
int main() {
  string string1 = "Nice to meet you";
  string string2;
  string result;

  string2 = "do you like blueberries?";
  variavelNaoDeclarada = string1 + ", " + string2;

  return(0);
}
```

```
######## ERRO ########
Variavel variavelNaoDeclarada não foi declarada, linha: 7
```

Exemplo 2:

```
int main() {
  string string1 = "Nice to meet you";
  string string2;
  string result;

  string2 = "do you like blueberries?";
  result = string1 - ", " + string2;

  return(0);
}
```

```
######## ERRO ########
Operações entre strings só aceitam o operador de concatenação (+), linha: 7
```

Exemplo 3:

```
int main() {
  string string1 = "Nice to meet you";
  int inteiro1;
  string result;

  inteiro1 = 5 * 10 + 4;
  result = string1 + ", " + inteiro1;

  return(0);
}
```

```
######## ERRO ########
Strings só podem fazer operações com outras strings, linha: 7
```

Exemplo 4:

```
int main() {
  string string1 = "Nice to meet you";
  string string2;
  int result;

  string2 = "do you like blueberries?";
  result = string1 + ", " + string2;

  return(0);
}
```

```
######## ERRO ########
A variavel result deve receber um valor de tipo compatível, linha: 7
```

### 8.3 Verificação de passagem de parâmetro

Para verificar os parâmetros foi necessário adicionar a seguinte estrutura de dados:

```
struct params {
  int type; // 0 = int, 1 = float, 2 = string
  struct params *next;
};

struct func_params {
  struct params *first;
  char *func_name; // nome da  funcao
  int n_params; // numero de parametros
  UT_hash_handle hh;
};
```

A struct func_params é um hash cuja a chave é o nome da função. Essa struct possui uma lista de parâmetros responsável por armazenar o tipo dos mesmos. Ao ser feita a chamada de uma função, é recuperado a struct referente pelo nome e então é percorrido todos os parâmetros para ver se o número e o tipo dos parâmetros são compatíveis.

## 9 Código Intermediário

Durante o parser do código, a partir da árvore abstrata, em apenas uma passagem, vai sendo gerado um código intermediário de 3 elementos que será executado pelo TAC - the Three Address Code interpreter [3]. Caso não aconteça nenhum erro léxico, sintático ou semântico, o código será impresso no terminal.

## 9.1 Detalhes da implementação

Nessa implementação a sessão ".table"fica vazia e cada variável do programa é representada por um registrador diferente. Os registradores são usados de duas formas: Para uso interno do sistema, sendo esses usos retornar funções, variáveis intermediárias e armazenamento de strings literais, e também são utilizados para representar as variáveis do código, sendo os mesmos representados pelo campo addr contido na tabela de símbolos e na árvore abstrata.

Para imprimir o código intermediário, as linhas desse código vão sendo salva em uma struct chamada "code_line"que serve como uma lista.

```
struct code_line {
  UT_string *line;
  struct code_line *next;
};
```

Ao final da execução do código essa lista é percorrida imprimindo no terminal o código intermediário.

## 9.2 Exemplo de Implementacão

A seguir temos um código que executa a função fatorial de um número dado pelo usuário:

```
int fatorial(int var) {
  int result = 0;
  int pastVar = 0;
  pastVar = var - 1;
  if (pastVar <= 0) {
    return(1);
  }
  result = fatorial(pastVar);
  result = var * result;
  return (result);
}


int main() {
  int value;

  print("Digite um n mero\n");
  scan(value, int);
  value = fatorial(value);
  print(value);
  print("\n");

  return(0);
}
```

Esse código gera o seguinte código intermediário:

```
.table
.code
```

```
fatorial:
mov $2, 0
mov $3, 0
mov $4, #0
mov $5, 1
sub $6, $4, $5
mov $3, $6
mov $7, 0
mov $8, 0
sleq $8, $3, $7
brz L1, $8
return 1
L1:
param $3
call fatorial, 1
pop $2
mov $9, #0
mul $10, $9, $2
mov $2, $10
return $2
main:
mov $12, 0
mema $13, 19
mov $13[0], 68
mov $13[1], 105
mov $13[2], 103
mov $13[3], 105
mov $13[4], 116
mov $13[5], 101
mov $13[6], 32
mov $13[7], 117
mov $13[8], 109
mov $13[9], 32
mov $13[10], 110
mov $13[11], -61
mov $13[12], -70
mov $13[13], 109
mov $13[14], 101
mov $13[15], 114
mov $13[16], 111
mov $13[17], 10
mov $13[18], 0
mov $14, 0
L2:
mov $15, $13[$14]
seq $16, $15, 0
brnz L3, $16
inttoch $15, $15
print $15
add $14, $14, 1
jump L2
L3:
scani $12
param $12
```

```
57 call fatorial, 1
58 pop $12
59 print $12
60 mema $17, 2
61 mov $17[0], 10
62 mov $17[1], 0
63 mov $18, 0
64 L4:
65 mov $19, $17[$18]
66 seq $20, $19, 0
67 brnz L5, $20
68 inttoch $19, $19
69 print $19
70 add $18, $18, 1
71 jump L4
72 L5:
73 nop
```

Ao executar o código intermediário atravez do TAC - the Three Address Code interpreter [3] obetemos o seguinte resultado:

```
Digite um número
5
120
```

## 10 Arquivos de teste

O analisador possui 9 arquivos de teste: "input_correto1.txt", "input_correto2.txt", input_correto_fatorial.txt, "input_erro_lexico1.txt", "input_erro_lexico2.txt", "input_erro_sintatico1.txt", "input_erro_sintatico2.txt", "input_erro_semantico1.txt", "input_erro_semantico2.txt".

Para compilar e executar os arquivos, basta entrar no diretório do analisador léxico e digitar os seguintes comandos:

1. flex lang.l
2. bison -vdt –graph lang.y
3. gcc -g lex.yy.c lang.tab.c -Wall -o lang
4. ./lang input_correto1.txt
5. ./lang input_correto2.txt
6. ./lang input_correto_fatorial.txt
7. ./lang input_erro_lexico1.txt
8. ./lang input_erro_lexico2.txt
9. ./lang input_erro_sintatico1.txt

10. ./lang input_erro_sintatico2.txt

11. ./lang input_erro_semantico1.txt

12. ./lang input_erro_semantico2.txt

No Arquivo "input_erro_lexico1.txt", na linha 3 foi introduzido uma variável que se inicia com número. No Arquivo "input_erro_lexico2.txt", na linha 10 foi introduzido um caractere inválido (@). No Arquivo "input_erro_sintatico1.txt", na linha 11 faltou um ";". No Arquivo "input_erro_sintatico2.txt", na linha 22 o parentese do if não foi fechado. No Arquivo "input_erro_semantico1.txt", na linha 16 foi colocado uma variável não declarada. No Arquivo "input_erro_semantico2.txt", na linha 16 foi utilizado um "*"em uma operação entre strings. Operação entre strings aceita apenas o operador "+".

## Referências


[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, principles, techniques*, volume 7. 2nd edition edition, 2006.

[2] Troy D. Hanson. uthash. [Online; acessado 20-Novembro-2020; url: https://github.com/troydhanson/uthash.

[3] Luciano Santos. TAC - the Three Address Code interpreter, October 2014.