

# Trabalho 4 da disciplina de Tradutores\*

Ana Paula Martins Tarchetti - 17/0056082

Departamento de Ciência da Computação, Universidade de Brasília (UnB) - Brasília,  
DF, 70910-900  
aptarchetti@gmail.com

## 1 Introdução e motivação

Um compilador consiste em traduzir um programa em uma linguagem de programação fonte para uma linguagem de programação alvo. Além disso, um compilador também relata possíveis erros que possam estar presentes no código-fonte. A compreensão do processo de compilação/tradução permite a utilização das técnicas aprendidas em aplicações futuras. Exemplos desse uso são: editores de texto, sistemas de recuperação de informações, reconhecimento de padrões, construção de novas linguagem de programação, desenho digital e verificadores [ALSU03].

Este trabalho tem como objetivo fixar de forma prática o entendimento desse processo de construção de compiladores computacionais. Dessa forma, este documento escrito traz os detalhes da implementação que foi feita com base nos processos envolvidos na compilação/tradução de programas. A linguagem de programação fonte que será utilizada é a C-IPL, que se baseia em um subconjunto da linguagem C com algumas alterações e uma nova primitiva que foi criada com o intuito de facilitar a manipulação de listas. Em adição a isso, a linguagem de programação intermediária da compilação é a *Three Address Code* [San15].

### 1.1 Analisador léxico

A primeira etapa que foi implementada é o analisador léxico, também chamado de analisador linear. A função dessa etapa é identificar em um código as sequências de caracteres que formam *tokens*, sendo esse a unidade mínima que compõe o vocabulário de um programa e é composto da seguinte forma: <nome do token, atributos do token>.

Para realizar a implementação dessa etapa foi utilizada a ferramenta *Flex*, [PEM07], cujo intuito é gerar programas que realizam correspondência de padrões em um dado texto. Então, por meio dessa ferramenta, foram declaradas expressões regulares (que foram detalhadas no **Apêndice A**) para indicar as seguintes definições:

- LETRA: qualquer caractere alfabético;
- DIGITO: qualquer algarismo;
- ID: identificadores de funções e variáveis;

---

\* Professora: Cláudia Nalon

- INTEGER: reconhece números inteiros;
- FLOAT: reconhece números de ponto flutuante;
- STRING\_LITERAL: cadeias de texto delimitadas por aspas duplas;
- ESPAÇO: qualquer espaço em branco;
- Outras expressões regulares para reconhecimento de erros específicos.

Após isso, foram definidas regras de identificação de *tokens*. As regras declaradas são divididas em:

- Palavras-chave: **int**, **float**, **list**, **read**, **write**, **writeln**, **main**, **return**, **if**, **else**, **for**, **NIL**;
- Novas operações primitivas: “?”, “%”, “»”, “«”, “:”, “!” (sobrecarregado, podendo também denotar negação);
- Operadores: ||, &&, >, <, >=, <=, ==, !=, +, -, \*, /, =;
- Delimitadores e outros: **vírgula**, **ponto e vírgula**, **parênteses** e **chaves**;
- Regras auxiliares: Ignorar espaços, indicar comentários e indicar erros léxicos e suas linhas e colunas correspondentes.

## 1.2 Analisador sintático

O analisador sintático, também chamado de analisador gramatical, corresponde à segunda etapa da compilação. Essa etapa se baseia em uma Gramática Livre de Contexto (GLC) para expressar regras recursivas ou não, que indicam as estruturas de agrupamento de *tokens* que podem ser aceitas pelo compilador. A GLC utilizada neste trabalho está descrita no **Apêndice B** deste documento. Ademais, é nessa etapa em que se cria/popula a árvore sintática abstrata, essa estrutura servirá de auxílio para as próximas etapas do compilador/tradutor.

Para realizar a implementação dessa etapa foi utilizada a ferramenta Bison, [DS15], cujo intuito é gerar programas que realizam essa análise supracitada a partir da declaração de uma GLC em um arquivo de extensão ".y". Em adição a isso, foram implementadas, em linguagem C, funções e estruturas auxiliares para que fosse possível a construção da tabela de símbolos e da árvore sintática. Ambas estruturas foram feitas por meio do uso de *struct's* para representar nós de uma árvore e ponteiros para *struct's* afim de poder relacionar esses nós.

A seguir, é possível compreender como se dá a organização das estruturas declaradas:

1. A tabela de símbolos é uma lista encadeada, aonde os nós possuem os seguintes campos:
  - char \*nome;
  - char \*tipo;
  - int escopo;
  - char \*var\_ou\_func;
  - char \*tipos\_dos\_parametros[253];
  - int contador\_de\_parametros;
  - struct t\_simbolo \* proximo;

2. A árvore sintática abstrata é uma estrutura mista de árvore e lista encadeada, aonde os nós possuem os seguintes campos:
  - char \*nome;
  - int linha;
  - int coluna;
  - char \*tipo;
  - struct t\_node \*primeiro\_filho;
  - struct t\_node \*proximo\_irmao;;
3. Dessa forma a profundidade da árvore é definida pelos ponteiros de filho e a largura da árvore é definida pelos ponteiros de irmão.

### 1.3 Analisador semântico

Nesta etapa, foi implementada a análise semântica. Nessa parte, além do uso da árvore sintática abstrata e da tabela de símbolos (lista encadeada), também foi feito o uso da árvore de escopo. A árvore de escopo se dá da seguinte forma:

1. Árvore de escopo: composta por *struct* de nós de árvore de escopo com os seguintes campos:
  - Número do escopo;
  - Ponteiro para o escopo pai;
2. Além disso, foram utilizadas variáveis auxiliares para armazenar o escopo atual e o contador de escopos existentes.
3. Por fim, as funções 'incrementa\_escopo' e 'decrementa\_escopo' são chamadas ao início de funções/blocos e ao final de funções/blocos respectivamente para o gerenciamento da árvore de escopo e das variáveis auxiliares.

Com base nisso, esta parte da implementação realiza a verificação dos seguintes pontos:

1. A existência da declaração da função 'main'.
  - Para isso foi criada a função '**existe\_main**' que verifica isso de acordo com a tabela de símbolos.
2. A redeclaração de variáveis e funções no mesmo escopo devem gerar erro.
  - Para isso foi criada a função '**incrementa\_tabela**' que verifica isso de acordo com a tabela de símbolos e apenas adiciona o símbolo caso ele não tenha sido declarado no escopo em questão, caso o contrário um erro é gerado.
3. O uso de variáveis e funções que não tenha sido declaradas em seu contexto de uso (escopo de uso ou escopo ancestral do escopo de uso) deve gerar erro.
  - Para isso foi criada a função '**verifica\_contexto**' que verifica isso de acordo com a tabela de símbolos e também de acordo com a árvore de escopo, supracitada.
4. A quantidade de parâmetros passados na chama de uma função deve ser a mesma quantidade de parâmetros definidos na declaração da função.
  - Para isso foi criada a função '**verifica\_qnt\_parametros\_chamada\_func**' que verifica isso de acordo com a tabela de símbolos.

5. O tipo da expressão de retorno, caso exista chamada de retorno, deve ter o mesmo tipo de retorno do tipo da função, ou deve poder ser convertido implicitamente para o tipo declarado na função.
6. Para todos os casos em que se apliquem deve ser possível fazer a conversão implícita de tipos, caso o contrário, um erro deve ser gerado.

#### 1.4 Tradutor (Gerador de Código Intermediário)

Após verificar a inexistência de erros (léxicos, sintáticos e semânticos) no arquivo de entrada em C-IPL, passamos agora para a fase de tradução do código de linguagem C-IPL para a linguagem TAC, previamente descrita. Para isso, é criado um arquivo de mesmo nome do arquivo de teste, porém agora com a extensão *".tac"*.

A seção *".table"* do arquivo de saída foi feita a partir da tabela de símbolos advinda das fases de Análise Sintática e Semântica e levando em consideração o escopo de cada variável, sendo esse escopo indicado pela concatenação do número do escopo ao final da *string* que indica o nome da variável. Já a seção de *".code"* foi feita a partir da árvore sintática abstrata anotada advinda também das fases de Análise Sintática e Semântica.

Ainda cabe citar os seguintes pontos:

- Os *labels* de funções foram gerados concatenando-se a *string* "F\_" e o nome da função, para que fosse possível diferenciar os *labels* de função dos outros *labels*.
- A mesma lógica do item anterior foi seguida para criação dos *labels* usados no fluxo de controle, porém nesse caso, utilizou-se a concatenação da *string* "C\_" para indicar esses casos específicos.
- Os temporários foram criados de acordo com a *string* "\$<contador>", sendo que esse contador é um contador inteiro de temporários que é incrementado a cada novo temporário.
- A nova primitiva será implementada com o auxílio das operações "mema" e "memf" em TAC que possibilitam alocação dinâmica de memória e a desalocação respectivamente.

## 2 Compilação, Execução e Testes

Os passos para compilação e execução do analisador podem ser encontrados no arquivo **README.txt** do diretório principal do projeto, ou, a partir do diretório principal, siga os seguintes comandos:

```

Compilar:
$ bison -v --defines=lib/sintatico.tab.h src/sintatico.y
  -o src/sintatico.tab.c
$ flex -o src/lex.yy.c src/lexico.l
$ gcc-11 -std=c18 -std=c99 -Wall -Wextra -Wpedantic -g -I lib/

```

```
-Wall -o tradutor src/sintatico.tab.c src/lex.yy.c
src/tradutor_utils.c -lfl
```

Compilar e executar:

```
$ make
$ ./tradutor tests/lexico/<nome_do_teste>
$ ./tradutor tests/sintatico/<nome_do_teste>
$ ./tradutor tests/semantico/<nome_do_teste>
$ ./tradutor tests/tradutor/<nome_do_teste>
```

Para testar no Interpretador de TAC, baixe o repositório <https://github.com/lhsantos/tac> e gere o executável de acordo com as instruções do pdf do diretório "doc". Por fim, copie o executável chamado "tac" para o diretório principal deste trabalho e execute a seguinte linha de comando:

```
$ ./tac tests/tradutor/<nome_do_teste.tac>
```

Tendo isso, foram criados os seguintes **arquivos de teste** para validar o funcionamento esperado nesta etapa da implementação:

- (i) **tradutor/correto\_teste\_1.c** e (ii) **tradutor/correto\_teste\_2.c** que não apresentam erros;
- (iii) **semantico/errado\_teste\_1.c**: Esse teste possui dois erros sintáticos e dois erros semânticos, o primeiro erro semântico é a falta da declaração da função 'main' no escopo global, o segundo erro semântico é o uso da variável 'b' (linha 4, coluna 5) sem ter declarado a variável no escopo do uso ou em algum escopo pai do escopo do uso. Já o primeiro erro sintático é um ponto-e-vírgula inesperado (linha 6, coluna 8) e o segundo erro sintático é a vírgula inesperada (linha 7, coluna 21).
- (iv) **tradutor/errado\_teste\_2.c**: Esse teste possui dois erros sintáticos e dois erros semânticos, o primeiro erro semântico é a redeclaração da variável 'a' no mesmo escopo (linha 3, coluna 11), inicialmente declarada na linha 2 e coluna 9, o segundo erro semântico é a chamada de uma função passando a quantidade errônea de parâmetros esperados (linha 13, coluna 23). Já o primeiro erro sintático é um FECHA\_PARENTESES esperado (linha 9, coluna 11) e o segundo erro sintático é uma CONANTE (INT) inesperada (linha 10, coluna 11).

## Referências

- [ALSU03] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools (for Anna University)*, 2/e. Pearson Education India, 2003.
- [DS15] Charles Donnelly and Richard Stallman. Gnu bison—the yacc-compatible parser generator. *Free Software Foundation, Cambridge*, 2015.

- [PEM07] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex. *University of California*, page 28, 2007.
- [San15] Luciano Santos. Tac - interpretador de código de três endereços - manual de referência, ref. versão 0.11. <https://github.com/lhsantos/tac/blob/master/doc/tac.pdf>, Março 2015. (Último acesso em 10/03/2021).

## Apêndice A Tabela de definições regulares

Nome	Expressões Regulares (Regex)	Padrão
LETRA	[a-zA-Z]	letras (maiúsculas e minúsculas)
DIGITO	[0-9]	dígitos de 0 a 9
ID	[_a-zA-Z][_a-zA-Z0-9]*	identificadores
INTEGER	{DIGITO}+	número inteiro
FLOAT	{(DIGITO)*\.\?(DIGITO)+   {DIGITO} + \.}	número com ponto flutuante
STRING_LITERAL	\\"(\\".\[^"\\])*\\"	expressões entre aspas duplas
ESPACO	[ \t \v \f \r]	todos os tipos de espaço em branco

## Apêndice B Gramática

$\langle \text{programa} \rangle ::= \langle \text{lista\_de\_declaracoes} \rangle$   
 $\langle \text{lista\_de\_declaracoes} \rangle ::= \langle \text{lista\_de\_declaracoes} \rangle \langle \text{declaracao} \rangle$   
 $\quad \mid \varepsilon$   
 $\langle \text{declaracao} \rangle ::= \langle \text{declaracao\_de\_variavel} \rangle$   
 $\quad \mid \langle \text{declaracao\_de\_funcao} \rangle$   
 $\langle \text{declaracao\_de\_variavel} \rangle ::= \langle \text{tipo\_de\_variavel\_id} \rangle \text{PONTO\_VIRGULA}$   
 $\langle \text{tipo\_de\_variavel\_id} \rangle ::= \langle \text{tipo\_de\_variavel} \rangle \langle \text{id} \rangle$   
 $\langle \text{id} \rangle ::= \text{ID}$   
 $\langle \text{declaracao\_de\_funcao} \rangle ::= \langle \text{tipo\_de\_variavel\_id} \rangle \text{ABRE\_PARENTESSES}$   
 $\quad \langle \text{parametros} \rangle \text{FECHA\_PARENTESSES} \langle \text{definicao\_de\_funcao} \rangle$   
 $\langle \text{definicao\_de\_funcao} \rangle ::= \langle \text{bloco\_de\_comando} \rangle$   
 $\langle \text{parametros} \rangle ::= \langle \text{lista\_de\_parametros} \rangle$   
 $\quad \mid \varepsilon$   
 $\langle \text{lista\_de\_parametros} \rangle ::= \langle \text{lista\_de\_parametros} \rangle \text{VIRGULA} \langle \text{parametro} \rangle$   
 $\quad \mid \langle \text{parametro} \rangle$   
 $\langle \text{parametro} \rangle ::= \langle \text{tipo\_de\_variavel\_id} \rangle$   
 $\langle \text{comando} \rangle ::= \langle \text{bloco\_de\_comando} \rangle$   
 $\quad \mid \langle \text{comando\_unico} \rangle$   
 $\langle \text{comandos} \rangle ::= \langle \text{comandos} \rangle \langle \text{comando} \rangle$   
 $\quad \mid \varepsilon$   
 $\langle \text{bloco\_de\_comando} \rangle ::= \text{ABRE\_CHAVES} \langle \text{comandos} \rangle \text{FECHA\_CHAVES}$

$$\begin{aligned}
\langle \text{comando\_unico} \rangle &::= \langle \text{comando\_condicional} \rangle \\
&| \langle \text{comando\_iterativo} \rangle \\
&| \langle \text{declaracao\_de\_variavel} \rangle \\
&| \langle \text{chamada\_de\_retorno} \rangle \\
&| \langle \text{comando\_de\_atribuicao} \rangle \\
&| \langle \text{exp\_or\_empty} \rangle \text{ PONTO\_VIRGULA} \\
\langle \text{comando\_condicional} \rangle &::= \text{IF ABRE\_PARENTESSES} \langle \text{exp} \rangle \text{ FECHA\_} \\
&\quad \text{PARENTESSES} \langle \text{comando} \rangle \\
&| \text{IF ABRE\_PARENTESSES} \langle \text{exp} \rangle \text{ FECHA\_PARENTESSES} \langle \text{comando} \rangle \\
&\quad \text{ELSE} \langle \text{comando} \rangle \\
\langle \text{comando\_iterativo} \rangle &::= \text{FOR ABRE\_PARENTESSES} \langle \text{expressao\_for} \rangle \text{ PONTO\_} \\
&\quad \text{VIRGULA} \langle \text{expressao\_for} \rangle \text{ PONTO\_VIRGULA} \langle \text{expressao\_for} \rangle \text{ FECHA\_} \\
&\quad \text{PARENTESSES} \langle \text{comando} \rangle \\
\langle \text{expressao\_for} \rangle &::= \langle \text{id} \rangle \text{ ATRIB} \langle \text{exp\_or\_empty} \rangle \\
&| \langle \text{exp\_or\_empty} \rangle \\
\langle \text{exp\_or\_empty} \rangle &::= \langle \text{exp} \rangle \\
&| \varepsilon \\
\langle \text{exp} \rangle &::= \langle \text{exp} \rangle \text{ GT} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ LT} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ EQ} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ NE} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ LE} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ GE} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ AND} \langle \text{exp} \rangle \\
&| \langle \text{exp} \rangle \text{ OR} \langle \text{exp} \rangle \\
&| \langle \text{exp\_list} \rangle \\
\langle \text{exp\_list} \rangle &::= \langle \text{exp\_list} \rangle \text{ CONSTRUTOR exp\_l}(\text{exp\_list})\text{ist} \\
&| \langle \text{exp\_list} \rangle \text{ FILTER} \langle \text{exp\_list} \rangle \\
&| \langle \text{exp\_list} \rangle \text{ MAP} \langle \text{exp\_list} \rangle \\
&| \langle \text{exp\_aritmetica} \rangle \\
&| \varepsilon \\
\langle \text{exp\_aritmetica} \rangle &::= \langle \text{termo} \rangle \\
&| \langle \text{exp\_aritmetica} \rangle \text{ SOMA} \langle \text{termo} \rangle \\
&| \langle \text{exp\_aritmetica} \rangle \text{ SUB} \langle \text{termo} \rangle \\
\langle \text{termo} \rangle &::= \langle \text{fator} \rangle \\
&| \langle \text{termo} \rangle \text{ MULT} \langle \text{fator} \rangle \\
&| \langle \text{termo} \rangle \text{ DIV} \langle \text{fator} \rangle \\
\langle \text{fator} \rangle &::= \langle \text{constante} \rangle \\
&| \langle \text{func\_call\_exp} \rangle \\
&| \text{SUB} \langle \text{fator} \rangle \\
&| \text{SOMA} \langle \text{fator} \rangle
\end{aligned}$$



```

|   TAIL_OR_NOT <fator>
|   TAIL_POP <fator>
|   HEADER <fator>
|   ID
|   ABRE_PARENTESES <exp> FECHA_PARENTESES
<comando_de_atribuicao> ::= <id> ATRIB <exp> PONTO_VIRGULA
<func_call_exp> ::= <id> ABRE_PARENTESES <func_call_parameters> FECHA_
PARENTESES
|   <id> ABRE_PARENTESES FECHA_PARENTESES
|   READ ABRE_PARENTESES <id> FECHA_PARENTESES
|   WRITE ABRE_PARENTESES <exp> FECHA_PARENTESES
|   WRITELN ABRE_PARENTESES <exp> FECHA_PARENTESES
<func_call_parameters> ::= <func_call_parameters> VIRGULA <exp>
|   <exp>
<chamada_de_retorno> ::= RETURN <exp> PONTO_VIRGULA
<tipo_de_variavel> ::= INT
|   FLOAT
|   <list>
<list> ::= INT LIST
|   FLOAT LIST
<constante> ::= INTEGER_CONST
|   FLOAT_CONST
|   CONSTANTE_NIL
|   STRING_LITERAL

```