

Projeto e Implementação de uma Ferramenta de Compilação para a Linguagem TPP

Eduardo Kurek¹

¹Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

This work presents the construction of a compiler for the **TPP** language, following the four essential steps: lexical analysis, syntactic analysis, semantic analysis, and code generation. In the lexical analysis, the source code is segmented into tokens using regular expressions, identifying basic elements such as operators, identifiers, and numbers. Next, the syntactic analysis, based on a context-free grammar, validates the program's structure and builds a parse tree. The semantic analysis checks for type consistency, scopes, declarations, and the use of variables/functions, using symbol tables to ensure compliance with the language rules. Finally, code generation translates the abstract syntax tree into LLVM intermediate code, allowing for the creation of portable executables. The project demonstrates the practical application of tools such as **Flex**, **Bison**, and **LLVM**, illustrating the feasibility of implementing a functional compiler even for a simplified language.

Resumo

Este trabalho apresenta a construção de um compilador para a linguagem **TPP**, seguindo as quatro etapas essenciais: análise léxica, sintática, semântica e geração de código. Na análise léxica, o código-fonte é segmentado em tokens através de expressões regulares, identificando elementos básicos como operadores, identificadores e números. Em seguida, a análise sintática, baseada em uma gramática livre de contexto, valida a estrutura do programa e constrói uma árvore sintática. A análise semântica verifica coerência de tipos, escopos, declarações e uso de variáveis/funções, utilizando tabelas de símbolos para garantir conformidade com as regras da linguagem. Por fim, a geração de código traduz a árvore sintática abstrata para código intermediário LLVM, permitindo a criação de executáveis portáteis. O projeto demonstra a aplicação prática de ferramentas como **Flex**, **Bison** e **LLVM**, ilustrando a viabilidade de implementar um compilador funcional mesmo para uma linguagem simplificada.

1 Introdução

Compiladores são programas de computadores capazes de traduzir programas em uma linguagem para outra. Eles recebem um programa escrito na linguagem fonte e produzem um equivalente na linguagem-alvo (LOUDEN 2004).

Tais programas são muito importantes, pois permitem a escrita de programas em uma linguagem de alto nível, desse modo, não precisamos ter ciência completa do *hardware* em que estamos desenvolvendo. O compilador abstrai as informações e nuances de cada máquina, facilitando a portabilidade e simplificando muito o desenvolvimento de novos programas. Também são fundamentais, pois são responsáveis por otimizar o código-fonte; através de técnicas, é possível tornar o código resultante mais eficiente do que o original.

As linguagens de alto nível tentam aproximar a forma de escrever programas com a forma que nós, humanos, nos comunicamos. Escondendo detalhes de baixo nível como pertinência dos registradores entre procedimentos, cálculo de endereços para desvios, gerenciamento de memória, etc.

2 Análise Léxica

A análise léxica ou varredura é a fase em que o compilador lê o arquivo de entrada e o separa em *tokens*, que são marcações que representam as palavras daquela linguagem. Exemplos de tokens são as palavras reservadas, como **if** e **else**, identificadores, que são rótulos atribuídos pelo usuário, símbolos algébricos, entre outros.

Para realizar a varredura, precisamos de um método eficiente para identificar os padrões dessa linguagem, que são: **expressões regulares** ou **autômatos finitos** (Menezes 2011). O método utilizado para esse desenvolvimento foi a utilização de expressões regulares (JARGAS 2012), que é uma maneira formal de descrever um padrão de caracteres.

2.1 Especificações da linguagem TPP

Tpp é uma linguagem simples, sem suporte a orientação a objetos, *strings* e ponteiros. Seus tipos primitivos são **inteiro** e **flutuante**, e também não é possível definir estruturas. A linguagem suporta a criação de funções com múltiplos parâmetros, e vetores de no máximo duas dimensões.

Como a linguagem não suporta ponteiros, é necessário trabalhar com vetores globais, em caso de utilização em várias funções. Os procedimentos **principal**, **escreva** e **leia** são reservados da linguagem. Comentários em tpp ficam dentro de chaves ({ Comentário }). O Código 1 é um exemplo escrito na linguagem tpp, já a tabela 1 mostra todos os padrões de expressões regulares utilizados e seus respectivos tokens.

```
1 inteiro: n
2
3 inteiro fatorial(inteiro: n)
4     inteiro: fat
5     se n > 0 então {não calcula se n > 0}
6         fat := 1
7         repita
8             fat := fat * n
9             n := n - 1
10        até n = 0
11        retorna(fat) {retorna o valor do fatorial de n}
12    senão
13        retorna(0)
14    fim
15 fim
16
17 inteiro principal()
18     leia(n)
19     escreva(fatorial(n))
20     retorna(0)
21 fim
```

Código 1: Exemplo de código em tpp

Expressão regular	Token
\+	MAIS
-	MENOS
*	VEZES
\/	DIVIDE
\:	DOIS_PONTOS
,	VIRGULA
\<	MENOR
>	MAIOR
=	IGUAL
\<>	DIFERENTE
\<=	MENOR_IGUAL
>=	MAIOR_IGUAL
&&	E
\	OU
!	NAO
\(ABRE_PARENTESE
\)	FECHA_PARENTESE
\[ABRE_COLCHETE
\]	FECHA_COLCHETE
se	SE
então	ENTAO
senão	SENAO
fim	FIM
repita	REPITA
até	ATE
:=	ATRIBUICAO
leia	LEIA
escreva	ESCREVA
retorna	RETORNA
inteiro	INTEIRO
flutuante	FLUTUANTE
[0-9]+	NUM_INTEIRO
[-] ? [0 - 9] + (\ . ([0 - 9] +) ?)	NUM_PONTO_FLUTUANTE
(([-\+]?)([1-9])\.([0-9] + [eE]([- \+]?)([0-9] +))	NUM_NOTACAO_CIENTIFICA
[a-zA-ZáÂãÄàÀéÉíÍóÓõÕ][a-zA-ZáÂãÄàÀéÉíÍóÓõÕ0-9_]*	ID

Tabela 1: Expressões regulares e tokens da linguagem tpp

2.2 Implementação

Como o processo de análise léxica é muito semelhante entre as várias linguagens de programação, ferramentas de desenvolvimento para essa etapa foram criadas, que é o caso do **Flex**, feito para a linguagem C e posteriormente para C++ (Paxson 1990). Essa ferramenta permite o rápido desenvolvimento de um analisador léxico através da definição de regras. O Código 2 mostra como podemos definir essas regras.

No código 2, cada regra retorna seu token correspondente, que é necessário na etapa de análise

```

1 %%
2
3 \{([^\}]]|\n)*\} /* IGNORA COMENTÁRIOS */ ;
4
5 \+ return MAIS;
6 - return MENOS;
7 \* return VEZES;
8 \/ return DIVIDE;
9
10 regras omitidas...
11
12 {NUM_INTEIRO} { assignValue(); return NUM_INTEIRO; }
13 {NUM_PONTO_FLUTUANTE} { assignValue(); return NUM_PONTO_FLUTUANTE; }
14 {NUM_NOTACAO_CIENTIFICA} { assignValue(); return NUM_NOTACAO_CIENTIFICA;
15 }
16 {ID} { assignValue(); return ID; }
17 [ \t] /* IGNORA ESPAÇOS EM BRANCO */
18 \r\n|\n /* IGNORA QUEBRAS DE LINHA */
19 .|ç showError();
20 %%

```

Código 2: Código das regras do analisador léxico

sintática, já os tokens de números guardam o lexema detectado naquela regra em uma variável global.

As linhas 3, 16 e 17 são necessárias para o Flex ignorar comentários e caracteres de quebra de linha. Na linha 17, foi necessário incluir a expressão `\r\n` pois em arquivos no formato CRLF, a quebra de linha é precedida pelo caracter `\r`. Na linha 18 pode-se notar a presença do `ç`, isso se faz necessário pois o Flex lê caracteres byte a byte (em codificação UTF-8), e como o caracter `ç` é uma junção de dois bytes (C3 A7), precisa ser identificado como uma expressão à parte. Caso contrário, C3 seria identificado como o caracter `Ã`, e a saída não seria como o esperado.

```

1 typedef struct error{
2     const char* cod;
3     const char* msg;
4     const char type;
5 } Error;
6
7 // Lexer errors
8 extern Error ERR_LEX_USE;
9 extern Error ERR_LEX_NOT_TPP;
10 extern Error ERR_LEX_FILE_NOT_EXISTS;
11 extern Error ERR_LEX_INV_CHAR;

```

Código 3: errors.h

Pode-se utilizar da *flag* `-k` para que a saída do analisador léxico exiba apenas os tokens, sem mensagens adicionais. Tratamentos de erros também foram implementados. Sendo eles:

- **ERR-LEX-USE:** Erro fatal, a execução do varredor deve ser feita com a seguinte assinatura:
`./lex code.tpp [-k];`
- **ERR-LEX-NOT-TPP:** Erro fatal, disparado quando o programa enviado por parâmetro não possui a extensão `.tpp`;

- **ERR-LEX-NOT-EXISTS:** Erro fatal, disparado quando o programa enviado por parâmetro não existe;
- **ERR-LEX-INV-CHAR:** Disparado quando o analisador encontra algum caracter inválido;

Ao compilar o varredor, podemos executar e ver a saída que ele gerou. As Figuras 1 e 2 exibem o arquivo gerado pelo lexer ao processar o Código 1, com e sem a flag -k, respectivamente. Na Figura 2, o formato LexToken exhibe, respectivamente: o token gerado pela regra, qual palavra gerou aquele token, qual linha e qual coluna a palavra foi encontrada.

INTEIRO	FECHA_PARENTESE	ID	FECHA_PARENTESE	FECHA_PARENTESE
DOIS_PONTOS	INTEIRO	VEZES	SENAO	ESCREVA
ID	DOIS_PONTOS	ID	RETORNA	ABRE_PARENTESE
FLUTUANTE	ID	ID	ABRE_PARENTESE	ID
DOIS_PONTOS	SE	ATRIBUICAO	NUM_INTEIRO	ABRE_PARENTESE
ID	ID	ID	FECHA_PARENTESE	ID
ABRE_COLCHETE	MAIOR	MENOS	FIM	FECHA_PARENTESE
NUM_INTEIRO	NUM_INTEIRO	NUM_INTEIRO	FIM	FECHA_PARENTESE
FECHA_COLCHETE	ENTAO	ATE	INTEIRO	RETORNA
INTEIRO	ID	ID	ID	ABRE_PARENTESE
ID	ATRIBUICAO	IGUAL	ABRE_PARENTESE	NUM_INTEIRO
ABRE_PARENTESE	NUM_INTEIRO	NUM_INTEIRO	FECHA_PARENTESE	FECHA_PARENTESE
INTEIRO	REPITA	RETORNA	LEIA	FIM
DOIS_PONTOS	ID	ABRE_PARENTESE	ABRE_PARENTESE	
ID	ATRIBUICAO	ID	ID	

Figura 1: Saída do léxico ao processar o código 1 com a flag -k

```

LexToken(INTEIRO, 'inteiro', 1, 1)
LexToken(DOIS_PONTOS, ':', 1, 8)
LexToken(ID, 'n', 1, 10)
LexToken(INTEIRO, 'inteiro', 3, 1)
LexToken(ID, 'fatorial', 3, 9)
LexToken(ABRE_PARENTESE, '(', 3, 17)
LexToken(INTEIRO, 'inteiro', 3, 18)
LexToken(DOIS_PONTOS, ':', 3, 25)
LexToken(ID, 'n', 3, 27)
LexToken(FECHA_PARENTESE, ')', 3, 28)
LexToken(INTEIRO, 'inteiro', 4, 5)
LexToken(DOIS_PONTOS, ':', 4, 12)
LexToken(ID, 'fat', 4, 14)
LexToken(SE, 'se', 5, 5)
LexToken(ID, 'n', 5, 8)
LexToken(MAIOR, '>', 5, 10)
LexToken(NUM_INTEIRO, '0', 5, 12)
LexToken(ENTAO, 'então', 5, 14)
LexToken(ID, 'fat', 6, 9)
LexToken(ATRIBUICAO, ':', 6, 13)
LexToken(NUM_INTEIRO, '1', 6, 16)
LexToken(REPITA, 'repita', 7, 9)
LexToken(ID, 'fat', 8, 13)
LexToken(ATRIBUICAO, ':', 8, 17)
LexToken(ID, 'fat', 8, 20)
LexToken(VEZES, '*', 8, 24)
LexToken(ID, 'n', 8, 26)
LexToken(ID, 'n', 9, 13)
LexToken(ATRIBUICAO, ':', 9, 15)
LexToken(ID, 'n', 9, 18)
LexToken(MENOS, '-', 9, 20)
LexToken(NUM_INTEIRO, '1', 9, 22)
LexToken(ATE, 'até', 10, 9)
LexToken(ID, 'n', 10, 14)
LexToken(IGUAL, '=', 10, 16)
LexToken(NUM_INTEIRO, '0', 10, 18)
LexToken(RETORNA, 'retorna', 11, 9)
LexToken(ABRE_PARENTESE, '(', 11, 16)
LexToken(ID, 'fat', 11, 17)
LexToken(FECHA_PARENTESE, ')', 11, 20)
LexToken(SENAO, 'senão', 12, 5)
LexToken(RETORNA, 'retorna', 13, 9)
LexToken(ABRE_PARENTESE, '(', 13, 16)
LexToken(NUM_INTEIRO, '0', 13, 17)
LexToken(FECHA_PARENTESE, ')', 13, 18)
LexToken(FIM, 'fim', 14, 5)
LexToken(FIM, 'fim', 15, 1)
LexToken(INTEIRO, 'inteiro', 17, 1)
LexToken(ID, 'principal', 17, 9)
LexToken(ABRE_PARENTESE, '(', 17, 18)
LexToken(FECHA_PARENTESE, ')', 17, 19)
LexToken(LEIA, 'leia', 18, 5)
LexToken(ABRE_PARENTESE, '(', 18, 9)
LexToken(ID, 'n', 18, 10)
LexToken(FECHA_PARENTESE, ')', 18, 11)
LexToken(ESCREVA, 'escreva', 19, 5)
LexToken(ABRE_PARENTESE, '(', 19, 12)
LexToken(ID, 'fatorial', 19, 13)
LexToken(ABRE_PARENTESE, '(', 19, 21)
LexToken(ID, 'n', 19, 22)
LexToken(FECHA_PARENTESE, ')', 19, 23)
LexToken(FECHA_PARENTESE, ')', 19, 24)

```

```
LexToken(RETORNA, 'retorna', 20, 5)
LexToken(ABRE_PARENTESE, '(', 20, 12)
LexToken(NUM_INTEIRO, '0', 20, 13)
```

```
LexToken(FECHA_PARENTESE, ')', 20, 14)
LexToken(FIM, 'fim', 21, 1)
```

Figura 2: Saída do léxico ao processar o código 1 sem a flag -k

3 Análise Sintática

A análise sintática ou *parser* é responsável por verificar a estrutura do código. Ela determina as regras gramaticais da linguagem através de uma **gramática livre de contexto (GLC)**. Observe a gramática da linguagem tpp no Código 4. Existem duas categorias gerais de algoritmos para análise sintática: *Análise Sintática Ascendente* e *Análise Sintática Descendente*.

Após essa etapa, é gerada uma árvore sintática com a estrutura do código e suas regras de derivações. A Figura 3 representa a árvore gerada pelo Código 5. O Código 6 exibe a estrutura de dados utilizada para cada nó da árvore.

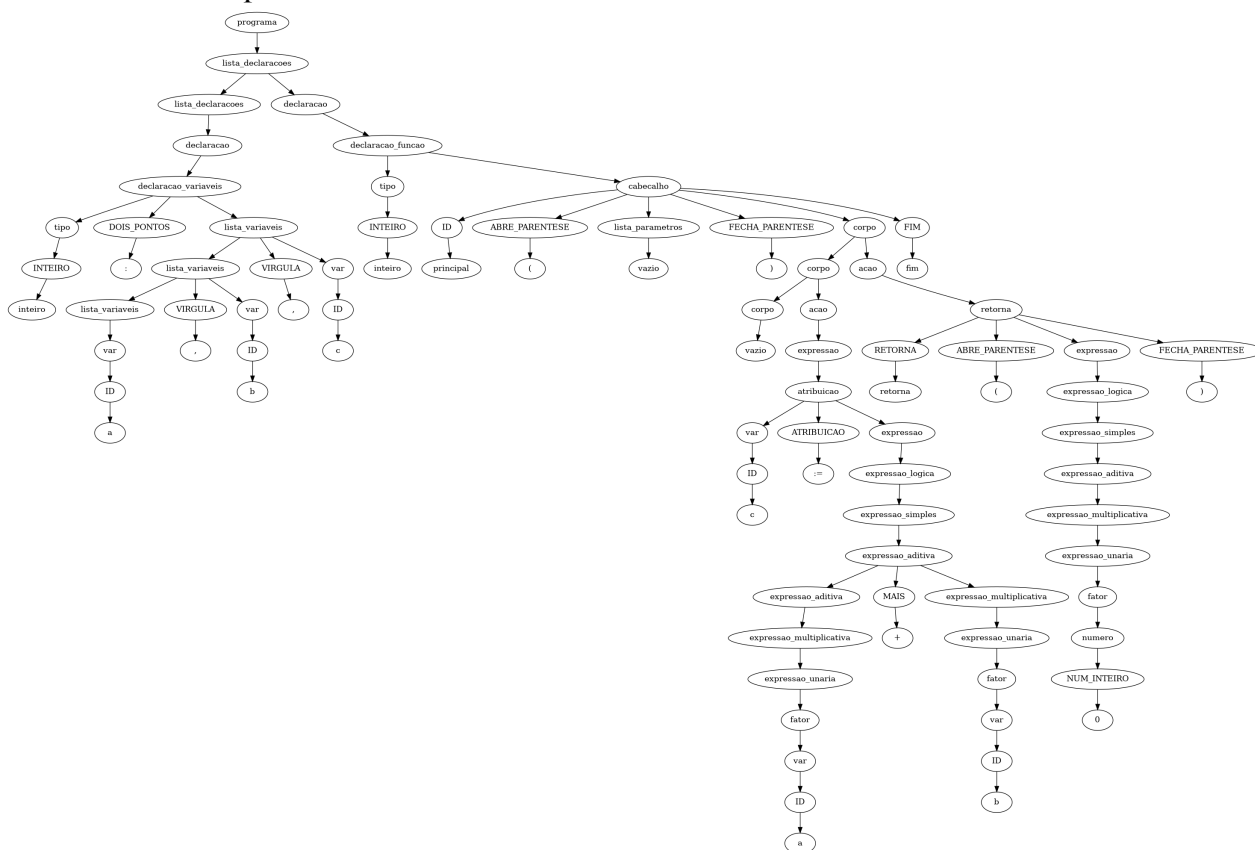


Figura 3: Árvore sintática gerada pelo Código 5

3.1 Implementação

Para implementar o *parser*, utilizei o programa **bison** Free Software Foundation (2014), pois ele funciona em conjunto com o Flex para a linguagem C. Com ele é possível definir as regras gramaticais da linguagem, e então, gerar a árvore sintática para essa linguagem. O bison utiliza do algoritmo LALR(1), que é um algoritmo amplamente utilizado para análise sintática, gerando a tabela de transições correspondente à gramática fornecida.

No Código 7 podemos ver como as regras são declaradas, juntamente com suas ações. Criei uma estrutura de dados para armazenar a árvore gerada, e assim tornou-se possível conectar os nós. No bison, as palavras coringas iniciadas com \$ referem-se ao índice do elemento da regra, sendo a palavra \$\$ reservada para referenciar o lado esquerdo da regra.

```

1 programa ::= lista_declaracoes
2 lista_declaracoes ::= lista_declaracoes declaracao | declaracao |
3 declaracao ::= declaracao_variaveis | inicializacao_variaveis |
    declaracao_funcao
4 declaracao_variaveis ::= tipo DOIS_PONTOS lista_variaveis
5 inicializacao_variaveis ::= atribuicao
6 lista_variaveis ::= lista_variaveis VIRGULA var | var
7 var ::= ID | ID indice
8 indice ::= indice ABRE_COLCHETE expressao FECHA_COLCHETE | ABRE_COLCHETE
    expressao
9 FECHA_COLCHETE
10 tipo ::= INTEIRO | FLUTUANTE
11 declaracao_funcao ::= tipo cabecalho| cabecalho
12 cabecalho ::= ID ABRE_PARENTESE lista_parametros FECHA_PARENTESE corpo FIM
13 lista_parametros ::= lista_parametros VIRGULA parametro | parametro | vazio
14 parametro ::= tipo DOIS_PONTOS ID | parametro ABRE_COLCHETE FECHA_COLCHETE
15 corpo ::= corpo acao| vazio
16 acao ::= expressao | declaracao_variaveis | se | repita | leia | escreva |
    retorna |
17 erro
18 se ::= SE expressao ENTAO corpo FIM | SE expressao ENTAO corpo SENA0 corpo FIM
19 repita ::= REPITA corpo ATE expressao
20 atribuicao ::= var ATRIBUICAO expressao
21 leia ::= LEIA ABRE_PARENTESE var FECHA_PARENTESE
22 escreva ::= ESCREVA ABRE_PARENTESE expressao FECHA_PARENTESE
23 retorna ::= RETORNA ABRE_PARENTESE expressao FECHA_PARENTESE
24 expressao ::= expressao_logica | atribuicao
25 expressao_logica ::= expressao_simples| expressao_logica operador_logico
26 expressao_simples
27 expressao_simples ::= expressao_aditiva | expressao_simples operador_relacional
28 expressao_aditiva
29 expressao_aditiva ::= expressao_multiplicativa | expressao_aditiva operador_soma
30 expressao_multiplicativa
31 expressao_multiplicativa ::= expressao_unaria | expressao_multiplicativa
32 operador_multiplicacao expressao_unaria
33 expressao_unaria ::= fator | operador_soma fator | operador_negacao fator
34 operador_relacional ::= MENOR | MAIOR| IGUAL| DIFERENTE| MENOR_IGUAL|
    MAIOR_IGUAL
35 operador_soma ::= MAIS | MENOS
36 operador_logico ::= E | OU
37 operador_negacao ::= NAO
38 operador_multiplicacao ::= VEZES | DIVIDE
39 fator ::= ABRE_PARENTESE expressao FECHA_PARENTESE | var | chamada_funcao |
    numero
40 numero ::= NUM_INTEIRO | NUM_PONTO_FLUTUANTE | NUM_NOTACAO_CIENTIFICA
41 chamada_funcao ::= ID ABRE_PARENTESE lista_argumentos FECHA_PARENTESE
42 lista_argumentos ::= lista_argumentos VIRGULA expressao | expressao | vazio

```

Código 4: Gramática da linguagem tpp

```

1 inteiro: a, b, c
2
3 inteiro principal()
4     c := a + b
5     retorna(0)
6 fim

```

Código 5: Código para geração de uma árvore sintática

```
1 typedef struct node{
2     int id;
3     char label[256];
4     struct node** ch;
5     int child_count;
6     int child_max;
7     node_type type;
8     int line;
9 } Node;
```

Código 6: Estrutura de dados usada para os nós da árvore sintática

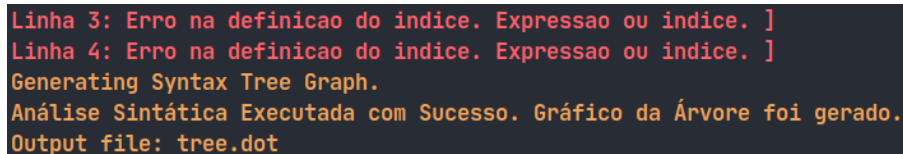
```
1 programa:
2     lista_declaracoes {
3         $$ = node_create("programa");
4         node_add_child($$, $1);
5         syntax_tree = $$;
6     }
7     | error { success = false; }
8 ;
9
10 lista_declaracoes:
11     lista_declaracoes declaracao {
12         $$ = node_create("lista_declaracoes");
13         node_add_children($$, 2, $1, $2);
14     }
15     | declaracao {
16         $$ = node_create("lista_declaracoes");
17         node_add_child($$, $1);
18     }
19 ;
```

Código 7: Código exemplo de utilização do bison


```
1 inteiro: a
2 inteiro: b
3 inteiro: c[]
4 flutuante: d[10][]
5
6 inteiro principal()
7     leia(a)
8     escreva(b)
9 fim
```

Código 8: Código tpp com erro nos índices

Após a implementação do arquivo de entrada para o bison, é possível iniciar o programa com um arquivo da linguagem fonte, e então será gerada uma imagem da árvore sintática. Tratamento de erros também são feitos nessa etapa, caso haja algum erro, ele será impresso na tela. Observe na Figura 4 a saída para o código 8.



```
Linha 3: Erro na definicao do indice. Expressao ou indice. ]
Linha 4: Erro na definicao do indice. Expressao ou indice. ]
Generating Syntax Tree Graph.
Análise Sintática Executada com Sucesso. Gráfico da Árvore foi gerado.
Output file: tree.dot
```

Figura 4: Erros identificados pelo parser

4 Análise Semântica

A análise semântica é a fase onde o código-fonte é analisado novamente e nele são verificadas anotações de atributos, inferência de tipos e se as construções sintáticas fazem algum sentido para a linguagem.

4.1 Regras Semânticas

As Regras Semânticas que são verificadas para a identificação de erros semânticos ou de contexto são:

- **Função principal (erro):** Todo programa escrito em tpp deve possuir uma função chamada principal. Tipo:
- **Declaração de funções (erro):** Funções precisam ser declaradas antes de serem chamadas;
- **Identificador de Função (erro):** Uma função ou procedimento é identificado pelo seu tipo de retorno, nome e parâmetros formais;
- **Função declarada e não utilizada (aviso):** Uma função pode ser declarada e não utilizada. Se isto acontecer uma aviso deverá ser emitido;
- **Retorno de Função (erro):** Uma função deve retornar um valor de tipo compatível com o tipo de retorno declarado;
- **Chamada à função principal (erro):** Uma função qualquer não pode fazer uma chamada à função principal;
- **Chamada recursiva da função principal (aviso):** Se a função principal fizer uma chamada para ela mesmo, a mensagem de aviso deve ser emitida;
- **Leitura de variável não declarada (erro):** Se houver a tentativa de leitura ou escrita de qualquer variável;
- **Variável declarada mas não inicializada (aviso):** Se uma variável 'a' for declarada e não for utilizada (não lida);

- **Declaração repetida de variáveis (aviso):** Se uma variável for declarada duas vezes no mesmo escopo;
- **Coerção de tipos (aviso):** Atribuição de tipos distintos devem gerar um aviso de coerção implícita de tipos;
- **Tipo do índice de *arrays* (erro):** O tipo do índice deve ser inteiro;
- **Índice fora do intervalo (erro):** O acesso à um índice fora do intervalo de um array deve gerar erro;

4.2 Tabela de funções

Na implementação da análise sintática, utilizei uma tabela para armazenar as funções disponíveis no meu programa. Naveguei por toda a árvore verificando as regras semânticas ao mesmo tempo que adicionava funções à tabela conforme as encontrava. Caso eu encontrasse uma chamada para determinada função a *flag* *used* era marcada como *true*. Observe a estrutura de dados utilizada para as entradas da tabela de funções no Código 9, já o Código 10 mostra o tratamento de quando um nó do tipo `NT_DECLARACAO_FUNCAO` era encontrado.

```
1 typedef struct func_table_entry {  
2     primitive_type return_type;  
3     char* name;  
4     int param_count;  
5     param** params;  
6     bool used;  
7     int line;  
8  
9     struct func_table_entry* next;  
10 } ft_entry;
```

Código 9: Estrutura de dados para as entradas da tabela de funções

4.3 Tabela de variáveis

Utilizei também uma tabela (Código 11) para armazenar as variáveis declaradas e utilizadas no meu programa. As variáveis são armazenadas com nome e escopo de função, desse jeito é possível declarar variáveis com o mesmo nome em escopo diferentes. A mesma estratégia utilizada na Seção 4.2 também foi utilizada aqui, conforme o algoritmo encontrava declarações de variáveis, elas eram inseridas na tabela de variáveis. Parâmetros de funções também eram inseridos. O Código 12 mostra a inclusão dessas variáveis na tabela quando um nó de declaração de variáveis era encontrado.

4.4 Implementação

Com as estruturas de dados definidas, a implementação da análise semântica se tornou mais simples. O algoritmo inicia na raiz da árvore e processa os nós filhos em suas respectivas funções de processamento. Esse processo é feito recursivamente até que nós folhas fossem atingidos. O Código 13 contém o início do processamento da análise semântica. Após a passagem completa na árvore, algumas regras também são verificadas para garantir a semântica do código-fonte.

A saída do processamento da análise semântica deve ser uma **árvore sintática abstrata (AST)**, que é um resumo da árvore sintática já vista na Figura 3. Essa árvore abstrata deve conter apenas os nós essenciais para a geração de código, a Figura 5 é a abstração dessa árvore citada.

```

1 void analisa_declaracao_funcao(Node* func){
2     ft_entry* entry = ft_inserir(func);
3     for(int i = 0; i < entry->param_count; i++){
4         vt_inserir_parametro(entry->params[i]->label,
5                               entry->params[i]->type, entry->name, entry->line);
6     }
7
8     func_has_return = false;
9     expected_return_type = entry->return_type;
10    if(func->child_count == 1)
11        analisa_corpo(func->ch[0]->ch[4], entry->name);
12    else
13        analisa_corpo(func->ch[1]->ch[4], entry->name);
14
15    if(expected_return_type != T_VAZIO && !func_has_return){
16        if(check_key) printf("%s\n", ERR_SEM_FUNC_RET_TYPE_ERROR.cod);
17        else printf("\033[1;31mLinha %d: %s %s\033[0m\n",
18                    func->ch[0]->line, ERR_SEM_FUNC_RET_TYPE_ERROR.msg,
19                    entry->name);
20    }
21 }

```

Código 10: Processamento do nó de declaração de função

```

1 struct var_table_entry {
2     char* name;
3     char* scope;
4     primitive_type type;
5     struct var_table_entry* next;
6     var_dimension dim;
7     int dim_1_size; // Usado para vetores
8     int dim_2_size; // Usado para matrizes
9     bool initialized;
10    bool used;
11    int line;
12 };

```

Código 11: Estrutura de dados para as entradas da tabela de variáveis

```

1 void analisa_declaracao_variaveis(Node* var_declaration, char* scope){
2     int qtde_variaveis;
3     vt_inserir(var_declaration, scope, &qtde_variaveis);
4 }

```

Código 12: Processamento do nó de declaração de variáveis

```
1 void analise_semantica(Node* node){
2     ft_init();
3     vt_init();
4
5     analisa_programa(node);
6
7     ft_verifica_principal_existe();
8     ft_verifica_declarada_nao_chamada();
9     vt_verifica_nao_inicializada();
10    vt_verifica_nao_utilizada();
11    vt_verifica_inicializada_nao_utilizada();
12
13    podar_programa(node);
14
15    vt_destroy();
16    ft_destroy();
17 }
18
19 void analisa_programa(Node* program){
20     analisa_lista_declaracoes(program->ch[0], "global");
21 }
22
23 void analisa_lista_declaracoes(Node* declaration_list, char* scope){
24     if(declaration_list->child_count == 2){
25         analisa_lista_declaracoes(declaration_list->ch[0], scope);
26         analisa_declaracao(declaration_list->ch[1], scope);
27     }else{
28         analisa_declaracao(declaration_list->ch[0], scope);
29     }
30 }
31
32 void analisa_declaracao(Node* declaration, char* scope){
33     switch (declaration->ch[0]->type){
34         case NT_DECLARACAO_VARIAVEIS: ...
35         case NT_INICIALIZACAO_VARIAVEIS: ...
36         case NT_DECLARACAO_FUNCAO: ...
37         default: break;
38     }
39 }
```

Código 13: Início do processamento da análise semântica

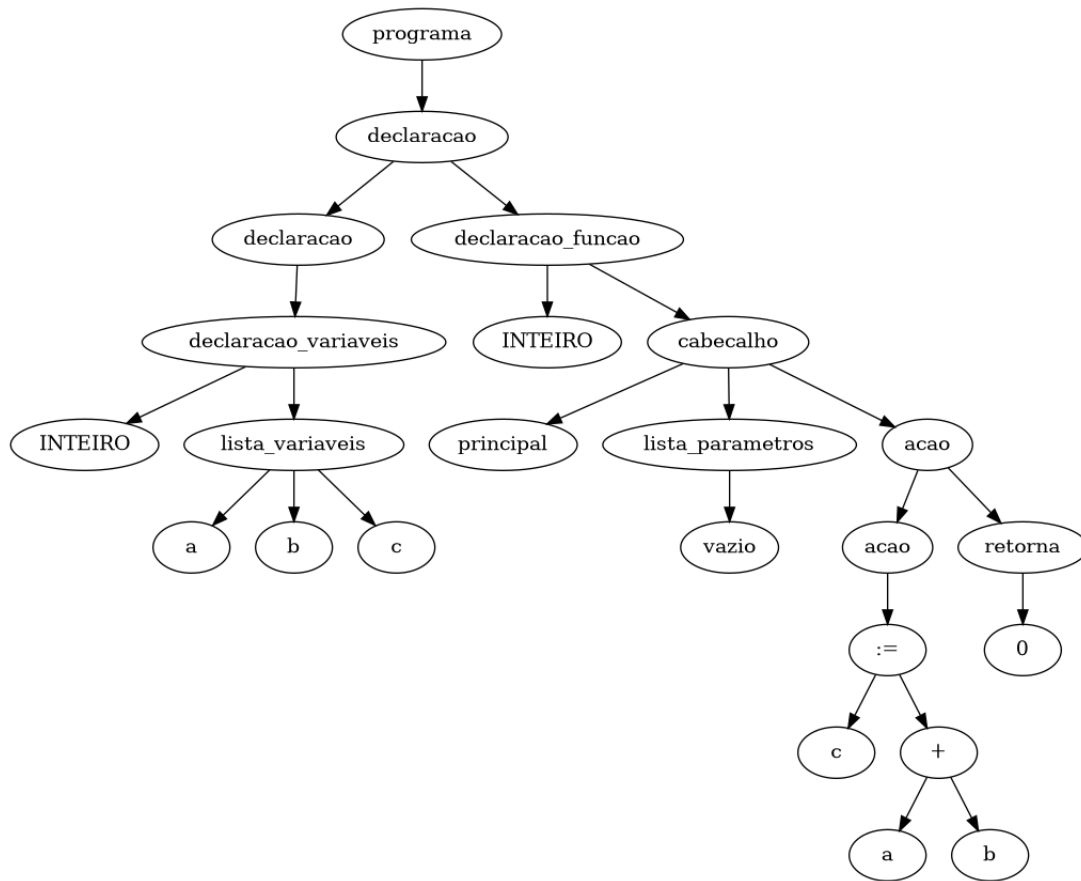


Figura 5: Árvore sintática abstrata gerada pelo Código 5

Para realizar essa poda, o algoritmo foi semelhante ao de processamento dos nós da análise sintática. A árvore era percorrida recursivamente, os nós antigos eram destruídos e os novos eram inseridos. O Código 14 demonstra o início da poda. Funções auxiliares da árvore foram utilizadas para fazer essa substituição e reorganizar dos nós, tais como `node_clone`, `node_clone_and_destroy`, `node_create_and_destroy`, `node_raise_child`, entre outras.

5 Geração de Código

A geração de código é a última etapa de compilação, nela o código-fonte é convertido para código da máquina alvo. Como existem muitas arquiteturas e sistemas operacionais, é comum primeiro converter o código-fonte para uma linguagem intermediária, e depois essa ferramenta intermediária é responsável por fazer a geração do código de máquina.

A ferramenta utilizada para geração de código intermediário nesse trabalho foi o LLVM Project (2024b). Com ele, podemos gerar código intermediário independente da arquitetura alvo, e, após isso, essa ferramenta fica responsável por gerar o código de máquina. Observe um exemplo de utilização dessa ferramenta na linguagem C no Código 15 tirado do repositório GONÇALVES (2016).

5.1 Implementação

Com o LLVM, é possível criar um módulo e inserir as instruções nesse módulo. Criei um algoritmo recursivo que percorria a árvore abstrata e gerava as instruções. Também utilizei uma tabela de símbolos nessa etapa, variáveis encontradas em um determinado escopo eram armazenadas nessa tabela, e ao fim do escopo, removidas.

```
1 void podar_programa(Node* node){
2     node->ch[0] = podar_lista_declaracoes(node->ch[0]);
3 }
4
5 Node* podar_lista_declaracoes(Node* node){
6     if(node->child_count == 1){
7         node->ch[0] = podar_declaracao(node->ch[0]);
8         return node_raise_child(node, 0);
9     }else{
10         node->ch[0] = podar_lista_declaracoes(node->ch[0]);
11         node->ch[1] = podar_declaracao(node->ch[1]);
12         Node* new = node_create(node->ch[1]->label, node->ch[1]->type);
13         node_add_children(new, 2, node_clone(node->ch[0]),
14             node_clone(node->ch[1]->ch[0]));
15         node_destroy(node);
16         return new;
17     }
18 }
19
20 Node* podar_declaracao(Node* node){
21     switch (node->ch[0]->type){
22         case NT_DECLARACAO_VARIAVEIS: ...
23         case NT_INICIALIZACAO_VARIAVEIS: ...
24         case NT_DECLARACAO_FUNCAO: ...
25         default: break;
26     }
27     return node;
28 }
```

Código 14: Início da poda da árvore sintática

```
1 int main(int argc, char *argv[]) {
2     LLVMContextRef context = LLVMGetGlobalContext();
3     LLVMModuleRef module = LLVMModuleCreateWithNameInContext("meu_modulo.bc",
4         context);
5     LLVMBuilderRef builder = LLVMCreateBuilderInContext(context);
6
7     // Cria um valor zero para colocar no retorno.
8     LLVMValueRef Zero64 = LLVMConstInt(LLVMInt64Type(), 0, false);
9
10    // Declara o tipo do retorno da função main.
11    LLVMTypeRef mainFnReturnType = LLVMInt64TypeInContext(context);
12    // Cria a função main.
13    LLVMValueRef mainFn = LLVMAddFunction(module, "main",
14        LLVMFunctionType(mainFnReturnType, NULL, 0, 0));
15
16    // Declara o bloco de entrada.
17    LLVMBasicBlockRef entryBlock = LLVMAppendBasicBlockInContext(context, mainFn,
18        "entry");
19    // Declara o bloco de saída.
20    LLVMBasicBlockRef endBasicBlock = LLVMAppendBasicBlock(mainFn, "exit");
21
22    // Adiciona o bloco de entrada.
23    LLVMPositionBuilderAtEnd(builder, entryBlock);
24
25    // Cria o valor de retorno e inicializa com zero.
26    LLVMValueRef returnVal = LLVMBuildAlloca(builder, LLVMInt64Type(), "retorno");
27    LLVMBuildStore(builder, Zero64, returnVal);
28
29    // Cria um salto para o bloco de saída.
30    LLVMBuildBr(builder, endBasicBlock);
31
32    // Adiciona o bloco de saída.
33    LLVMPositionBuilderAtEnd(builder, endBasicBlock);
34
35    // Cria o return.
36    LLVMBuildRet(builder, LLVMBuildLoad(builder, returnVal, ""));
37
38    // Imprime o código do módulo.
39    LLVMDumpModule(module);
40
41    // Escreve para um arquivo no formato bitcode.
42    if (LLVMWriteBitcodeToFile(module, "meu_modulo.bc") != 0) {
43        fprintf(stderr, "error writing bitcode to file, skipping\n");
44    }
45 }
```

Código 15: Exemplo de geração de código llvm em C

```

1 typedef LLVMValueRef Value;
2
3 Value processa_node(Node* node, Type expectedType){
4     switch (node->type){
5         case NT_RETORNA: return gerar_retorna(node, expectedType);
6         case NT_EXPRESSAO_ADITIVA: return expressao_aditiva(node,
7             expectedType);
8         case NT_EXPRESSAO_MULTIPLICATIVA: return
9             expressao_multiplicativa(node, expectedType);
10
11         // Código omitido...
12
13         case NT_ESCREVA: return escreva(node, expectedType);
14         case NT_LEIA: return leia(node, expectedType);
15         case NT_VAZIO: return NULL;
16
17         default: printf("\033[0;35mProcessamento '%s' não
18             implementado\033[0m\n", node_type_to_string(node));
19     }
20
21     return NULL;
22 }

```

Código 16: Função principal da geração de código LLVM

Uma função para processar um nó que retorna um `LLVMValueRef` foi a base do programa, ela também recebe um parâmetro chamado `expectedType` para que a coerção de tipos funcionasse corretamente. Veja o Código 16.

A função `processa_node`, vista no Código 16, é chamada várias vezes na geração de código. Veja a continuação das outras funções de geração no Código 17.

5.2 Biblioteca padrão

Para a leitura e impressão na tela, foi criado um código `llvm` que contém as funções necessárias para realizar essas operações. Elas foram escritas na linguagem C, e compilada para um arquivo `llvm` usando o compilador `clang` (LLVM Project (2024a)). O Código 18 contém as funções da biblioteca padrão.

Posteriormente, essas funções de entrada e saída, foram construídas por código a fim de evitar a dependência de um arquivo externo. Todo código compilado recebia essas funções em seu binário.

6 Resultados

Após a criação do compilador, teremos um executável que recebe um programa escrito na linguagem `tpp` e gera outro executável para a arquitetura `intel x64` com o sistema operacional `linux`. Observe o resultado da execução do Código 1 na Figura 6 juntamente com o código intermediário 19 gerado pelo compilador.


```

1 #define PROCESSA_NODE_EXPECTED(node, expectedType) processa_node(node,
2   expectedType)
3
4 // Retorna uma instrução de retorno
5 Value gerar_retorna(Node* node, Type expectedType){
6     return LLVMBuildRet(builder, PROCESSA_NODE_EXPECTED(node->ch[0],
7   expectedType));
8 }
9
10 /**
11  * Calcula uma expressão aditiva e retorna o valor
12  * convertido para o tipo esperado.
13  * Se um dos valores da operação for float,
14  * realiza a soma com as operações de float do llvm,
15  * e então retorna a conversão para o tipo esperado.
16  */
17 Value expressao_aditiva(Node* node, Type expectedType){
18     Node* left = node->ch[0], *right = node->ch[1];
19     Value leftValue = PROCESSA_NODE_EXPECTED(left, LLVMVoidType()),
20     rightValue = PROCESSA_NODE_EXPECTED(right, LLVMVoidType());
21     Kind leftKind = get_kind(&leftValue), rightKind =
22     get_kind(&rightValue);
23     bool isSum = strcmp(node->label, "+") == 0;
24
25     // Se os dois numeros forem inteiros, retorna um inteiro
26     if(leftKind == LLVMIntegerTypeKind && rightKind ==
27     LLVMIntegerTypeKind){
28         if(isSum) return converte(LLVMBuildAdd(builder, leftValue,
29     rightValue, "addtmp"), expectedType);
30         return converte(LLVMBuildSub(builder, leftValue, rightValue,
31     "subtmp"), expectedType);
32     }
33
34     // Converte ambos para float
35     leftValue = converte(leftValue, LLVMFloatType());
36     rightValue = converte(rightValue, LLVMFloatType());
37
38     // Faz a soma em floats
39     if(isSum) return converte(LLVMBuildFAdd(builder, leftValue,
40     rightValue, "addtmp"), expectedType);
41     return converte(LLVMBuildFSub(builder, leftValue, rightValue,
42     "subtmp"), expectedType);
43 }

```

Código 17: Funções de geração de código LLVM

```

1  "@.str" = private global [4 x i8] c"%d\0A\00"
2  "@.str.1" = private global [4 x i8] c"%f\0A\00"
3  "@.str.2" = private global [3 x i8] c"%d\00"
4  "@.str.3" = private global [3 x i8] c"%f\00"
5
6  declare i32 @printf(ptr, i32)
7
8  declare i32 @__isoc99_scanf(ptr, ptr)
9
10 define void @escrevaInteiro(i32 %0) {
11 entry:
12   %alloca = alloca i32, align 4
13   store i32 %0, ptr %alloca, align 4
14   %load = load i32, ptr %alloca, align 4
15   %printf = call i32 @printf(ptr "@.str", i32 %load)
16   ret void
17 }
18
19 define void @escrevaFlutuante(float %0) {
20 entry:
21   %alloca = alloca float, align 4
22   store float %0, ptr %alloca, align 4
23   %load = load float, ptr %alloca, align 4
24   %fpext = fpext float %load to double
25   %printf = call i32 @printf(ptr "@.str.1", double %fpext)
26   ret void
27 }
28
29 define void @leiaInteiro(ptr %0) {
30 entry:
31   %alloca = alloca ptr, align 8
32   store ptr %0, ptr %alloca, align 8
33   %load = load ptr, ptr %alloca, align 8
34   %__isoc99_scanf = call i32 @__isoc99_scanf(ptr "@.str.2", ptr %load)
35   ret void
36 }
37
38 define void @leiaFlutuante(ptr %0) {
39 entry:
40   %alloca = alloca ptr, align 8
41   store ptr %0, ptr %alloca, align 8
42   %load = load ptr, ptr %alloca, align 8
43   %__isoc99_scanf = call i32 @__isoc99_scanf(ptr "@.str.3", ptr %load)
44   ret void
45 }

```

Código 18: Funções da biblioteca padrão da linguagem tpp

```

1  define i32 @fatorial(i32 %0) {
2  entry:
3      %fat = alloca i32, align 4
4      %num = alloca i32, align 4
5      store i32 %0, ptr %num, align 4
6      %num1 = load i32, ptr %num, align 4
7      %int2float = sitofp i32 %num1 to float
8      %cmptmp = fcmp ogt float %int2float, 0.000000e+00
9      br i1 %cmptmp, label %then, label %else
10
11  then:
12      store i32 1, ptr %fat, align 4
13      br label %repita
14
15  else:
16      ret i32 0
17      br label %end
18
19  end:
20      ret i32 0
21
22  repita:
23      %fat2 = load i32, ptr %fat, align 4
24      %num3 = load i32, ptr %num, align 4
25      %multmp = mul i32 %fat2, %num3
26      store i32 %multmp, ptr %fat, align 4
27      %num4 = load i32, ptr %num, align 4
28      %subtmp = sub i32 %num4, 1
29      store i32 %subtmp, ptr %num, align 4
30      %num5 = load i32, ptr %num, align 4
31      %int2float6 = sitofp i32 %num5 to float
32      %cmptmp7 = fcmp ogt float %int2float6, 0.000000e+00
33      br i1 %cmptmp7, label %repita, label %fim
34
35  fim:
36      %fat8 = load i32, ptr %fat, align 4
37      ret i32 %fat8
38      br label %end
39  }
40
41  define i32 @main() {
42  entry:
43      call void @leiaInteiro(ptr @n)
44      %n = load i32, ptr @n, align 4
45      %calltmp = call i32 @fatorial(i32 %n)
46      call void @escrevaInteiro(i32 %calltmp)
47      ret i32 0
48      ret i32 0
49  }

```

Código 19: Código intermediário do programa que calcula o fatorial de um número

```
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./gen fatorial.tpp
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
0
0
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
1
1
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
2
2
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
3
6
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
4
24
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
5
120
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
6
720
• eduardo@B360M:~/Faculdade/opt011-compiladores/4_code_gen$ ./fatorial
7
5040
```

Figura 6: Resultados do código fatorial

7 Considerações Finais

A criação de um compilador para a linguagem tpp mostrou como é trabalhoso transformar uma linguagem de alto nível em código que o computador entenda, passando desde a leitura dos caracteres até a geração de um código intermediário com llvm. Ferramentas como Flex e Bison ajudaram a simplificar etapas importantes, como separar os elementos do código e montar a estrutura do programa, permitindo que eu me concentrasse nas regras e melhorias.

O projeto ressaltou como os compiladores são essenciais para esconder detalhes do hardware e melhorar o desempenho do código, além de mostrar desafios práticos, como lidar com escopos, conversões de tipos e controle de erros. A integração com llvm provou que é possível gerar um código que funcione em diferentes sistemas e que seja eficiente, transformando conceitos teóricos em uma aplicação real.

Por fim, essa experiência mostrou que ferramentas modernas e métodos organizados podem transformar um problema complicado em uma solução prática, ajudando a entender melhor as linguagens de programação e como elas funcionam na prática.

Referências

- Free Software Foundation. 2014. *Gnu bison: The parser generator*. GNU Project. <https://www.gnu.org/software/bison/>.
- GONÇALVES, R. A. 2016. Projeto llvm-gencode-samples. <https://github.com/rogerioag/llvm-gencode-samples>.
- JARGAS, Aurélio Marinho. 2012. *Expressões regulares: uma abordagem divertida*. São Paulo, SP: Novatec 4th edn.
- LLVM Project. 2024a. Clang: A C Language Family Frontend for LLVM. Acessado em: 19 fev. 2025. <https://clang.llvm.org/>.
- LLVM Project. 2024b. LLVM: The Compiler Infrastructure. Acessado em: 19 fev. 2025. <https://llvm.org/>.
- LOUDEN, Kenneth C. 2004. *Compiladores: Princípios e práticas*. São Paulo, SP: Thomson 1st edn.

- Menezes, Paulo B. 2011. *Linguagens formais e autômatos - ufrgs*, vol. 3. Porto Alegre: Bookman
6th edn. <https://integrada.minhabiblioteca.com.br/reader/books/9788577807994/>.
E-book, p.80.
- Paxson, Vern. 1990. Lexical analysis with flex. Acessado em: 20 de Outubro de 2024. <https://westes.github.io/flex/manual/>.