



Defesa Final

Compiladores

Jackson Júnior

Jackson Júnior 20200313

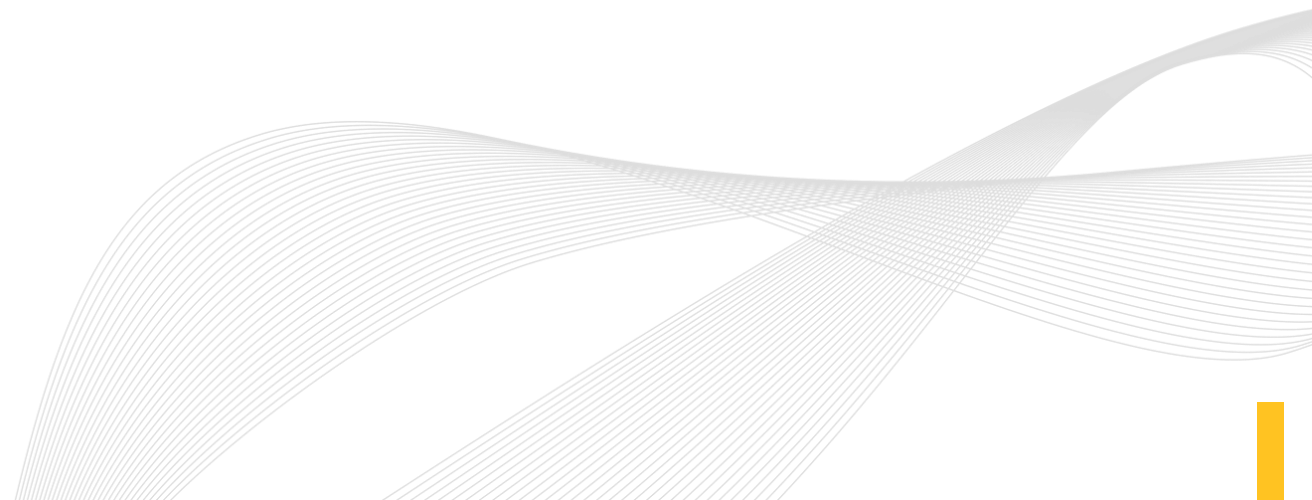
História

Introdução

Desenvolvimento

Resultados

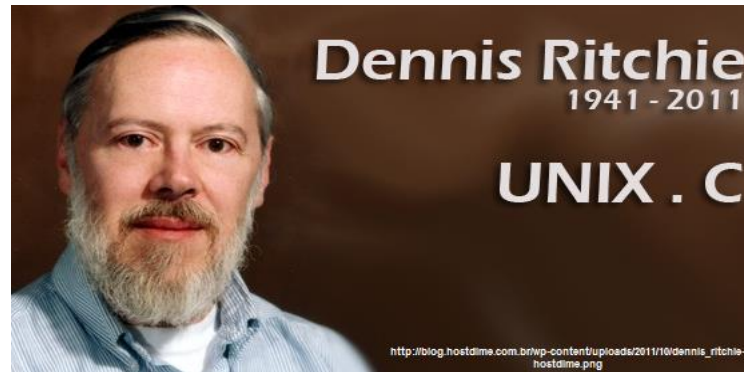
Conclusão



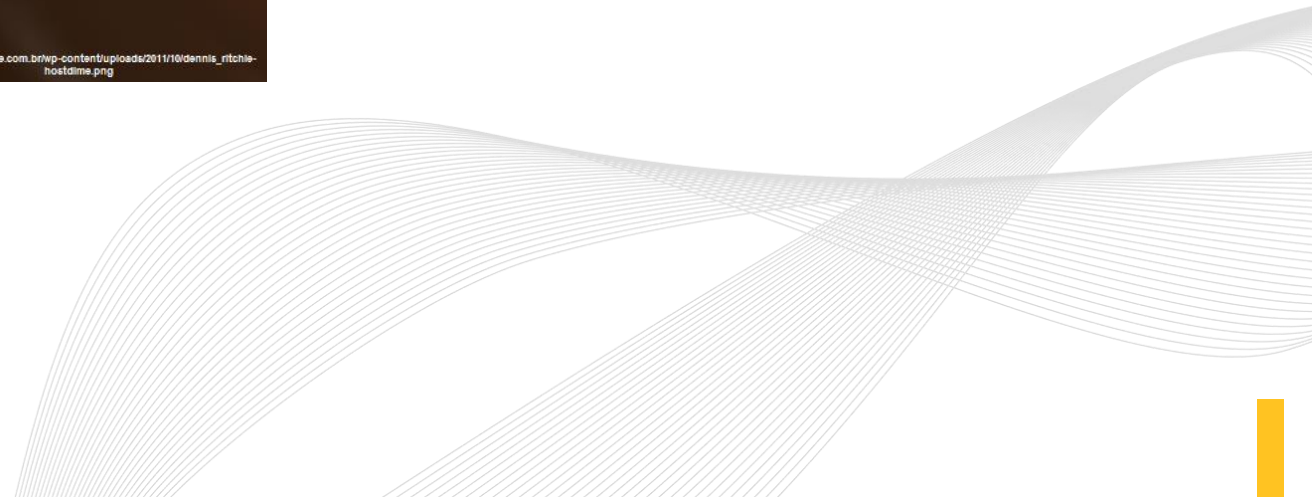
- Desenvolvida por Dennis Ritchie a linguagem C foi criada em 1972 com o objectivo de reescrever de forma portátil o sistema operativo Unix . Baseado na linguagem B foi desenvolvida nos laboratórios Bell da AT&T nos Estados Unidos.



laboratórios Bell da AT&T



Dennis Ritchie



Simplicidade: A linguagem C possui uma sintaxe simples e compacta, tornando-a fácil de aprender e ler. Ela é baseada em instruções e blocos delimitados por chaves.

Eficiência: A C é conhecida por sua eficiência em termos de uso de memória e tempo de execução. Ela permite um controle detalhado sobre a manipulação de memória e a execução de operações de baixo nível.

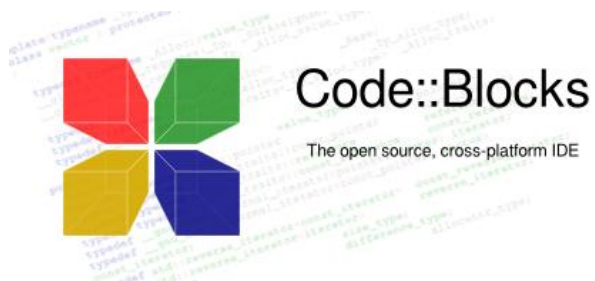
Acesso direto à memória: A C oferece recursos para manipulação direta da memória.

Portabilidade: A C foi projetada para ser altamente portátil, o que significa que os programas escritos em C podem ser executados em diferentes plataformas e sistemas operacionais com poucas modificações.







Bibliotecas padrão: A linguagem C possui uma biblioteca padrão rica, que inclui funções para entrada e saída de dados, manipulação de strings, gerenciamento de arquivos, matemática, alocação de memória e muito mais.

Ponteiros: A C apresenta o conceito de ponteiros, que são variáveis que armazenam endereços de memória. Os ponteiros fornecem um mecanismo poderoso para manipulação de dados e gerenciamento de memória, mas também podem ser uma fonte de erros se usados incorretamente.

Existem vários IDEs (Integrated Development Environments) para a linguagem de programação C porque cada IDE tem características e recursos diferentes que atendem às necessidades de diferentes programadores e projetos.



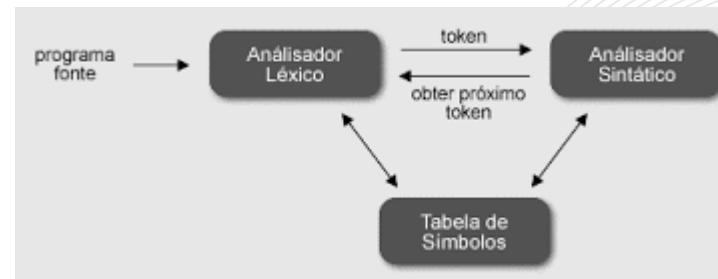
De acordo com o Índice TIOBE de março de 2023, o C é a 2ª linguagem de programação mais popular do mundo, atrás apenas do Python. O índice TIOBE é baseado em uma variedade de fatores, incluindo a quantidade de resultados de pesquisa no Google, a quantidade de programadores que utilizam a linguagem, a quantidade de cursos disponíveis e a quantidade de projetos em código aberto que utilizam a linguagem.

Dec 2022	Dec 2021	Change	Programming Language		Ratings	Change
1	1			Python	16.66%	+3.76%
2	2			C	16.56%	+4.77%
3	4	▲		C++	11.94%	+4.21%
4	3	▼		Java	11.82%	+1.70%
5	5			C#	4.92%	-1.48%
6	6			Visual Basic	3.94%	-1.46%
7	7			JavaScript	3.19%	+0.90%
8	9	▲		SQL	2.22%	+0.43%
9	8	▼		Assembly language	1.87%	-0.38%
10	12	▲		PHP	1.62%	+0.12%

Ranking das linguagens de programação mais populares do TIOBE Programming Community (Imagem: Reprodução/TIOBE)

Um compilador é uma ferramenta responsável por traduzir o código fonte de um programa em uma linguagem de programação para uma forma executável. O processo de compilação é dividido em várias fases, incluindo análise léxica, análise sintática e análise semântica.

- ❖ **Análise Léxica:** A primeira fase do compilador é a análise léxica. Nessa fase, o código fonte é dividido em tokens, que são unidades léxicas, como palavras-chave, identificadores, operadores, números e símbolos
- ❖ **Análise Sintática:** A análise sintática é responsável por verificar se a estrutura gramatical do código fonte está correta de acordo com a gramática da linguagem de programação.
- ❖ **Análise Semântica:** A análise semântica é a fase que verifica se o código fonte possui significado semântico correto. Ela analisa o contexto do programa e as relações entre as expressões e identificadores, garantindo que as operações sejam aplicadas corretamente e que as regras semânticas sejam obedecidas.

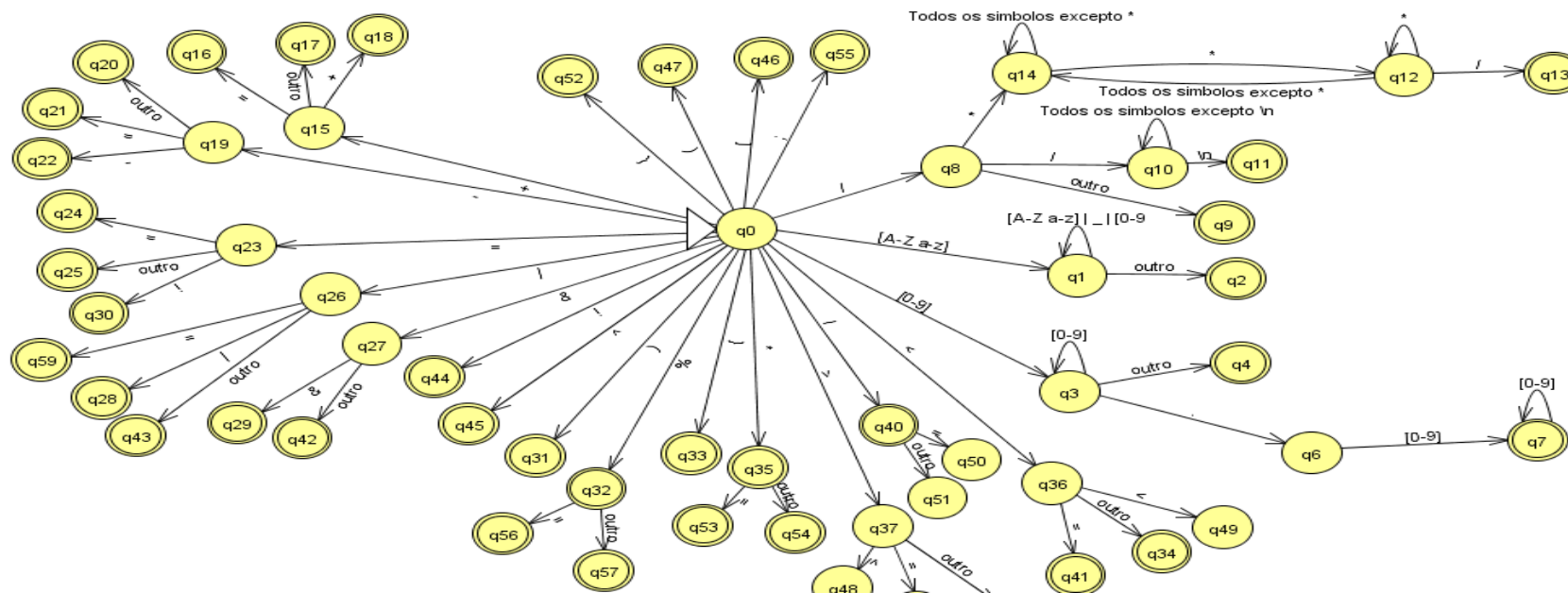


Um interpretador é uma ferramenta que lê e executa instruções de um programa linha por linha, em tempo real, sem a necessidade de compilar o código antecipadamente. Enquanto um compilador traduz todo o código fonte de uma vez para um formato executável, um interpretador analisa e executa cada instrução à medida que ela é encontrada.

- ❖ Execução em tempo real: O código é lido e executado linha por linha à medida que é encontrado, sem a necessidade de compilação prévia.
- ❖ Não produz um arquivo executável: Ao contrário dos compiladores, os interpretadores não geram um arquivo executável separado. A interpretação ocorre diretamente durante a execução.
- ❖ Portabilidade: Os interpretadores são frequentemente projetados para serem portáteis, permitindo a execução em diferentes sistemas operacionais, desde que haja um interpretador adequado disponível.
- ❖ Flexibilidade: Os interpretadores geralmente oferecem recursos dinâmicos, como avaliação de expressões e execução interativa, proporcionando flexibilidade e agilidade no desenvolvimento e experimentação.
- ❖ Linguagens interpretadas: Algumas linguagens de programação são projetadas para serem interpretadas, possuindo recursos e bibliotecas específicas para suportar a interpretação eficiente.

Foi necessário definir as regras léxicas da linguagem de programação. As regras descrevem os padrões de caracteres que correspondem a diferentes tokens, como palavras-chave, identificadores, números, símbolos e operadores.

Em seguida houve a necessidade da implementação da expressão regular e o autômato finito determinístico (DFA) como técnica de análise e implementação.



Para o desenvolvimento do mesmo houve a necessidade de recorrer a gramática da linguagem, onde foi necessário organizá-la para posteriormente implementá-la.

Ambiguidade:

Programa: programa -> declaração*

Declarações: declaração -> declaração_biblioteca | declaração_variável | declaração_função

Declaração de Variáveis: declaração_variável -> tipo identificador lista_variáveis? ';'

tipo -> int | float | char | double | void **lista_variáveis** -> identificador ('=' expressão)? (',' identificador ('=' expressão)?)* ;

Declaração de Funções: declaração_função -> tipo identificador '(' lista_parâmetros? ')' bloco

lista_parâmetros -> declaração_variável (',' declaração_variável)* ;

Declaração de Estruturas: declaração_estrutura -> struct identificador '{' declaração_variável* '}'
identificador_lista_variáveis? ';'

identificador_lista_variáveis -> identificador ('=' expressão)? (',' identificador ('=' expressão)?)* ;

Bloco: bloco -> '{' declaração* comando* '}'

Sem Ambiguidade:

Programa: programa \rightarrow declaração programa | ϵ

Declarações: declaração \rightarrow declaração_variável ';' | declaração_função

Declaração de Variáveis: declaração_variável \rightarrow tipo lista_variáveis

tipo \rightarrow int | float | char | double | void

lista_variáveis \rightarrow identificador ('=' expressão)? (',' identificador ('=' expressão)?)*

Declaração de Funções: declaração_função \rightarrow

tipo identificador '(' lista_parâmetros? ')' bloco

lista_parâmetros \rightarrow lista_parametros_aux | ϵ

lista_parametros_aux \rightarrow declaração_variável (',' lista_parametros_aux)?

Declaração de Estruturas: declaração_estrutura \rightarrow struct identificador '{' lista_variáveis '}'
identificador_lista_variáveis? ';'

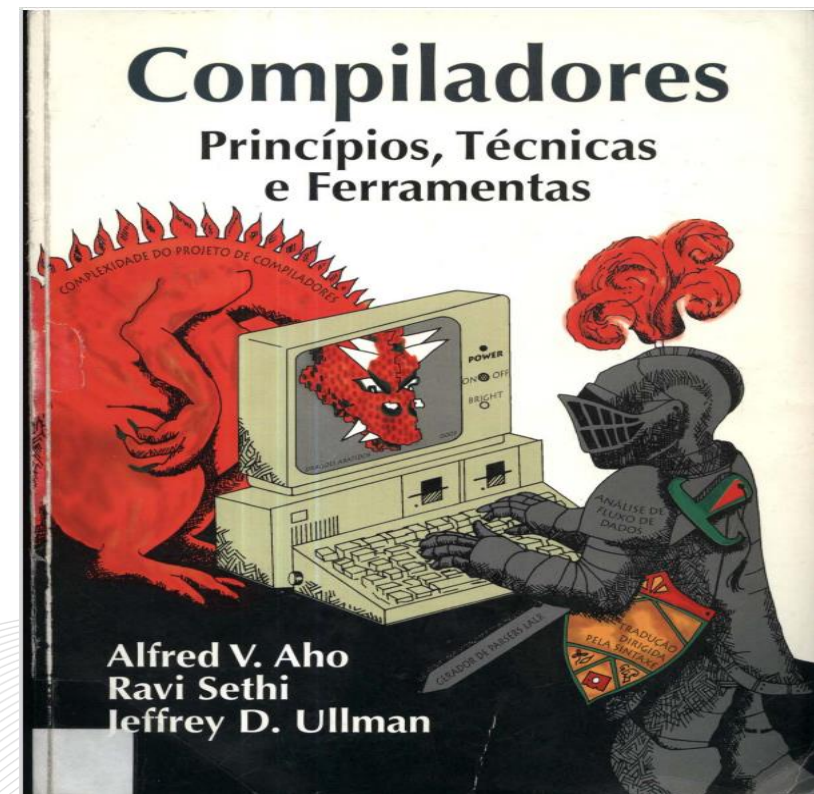
identificador_lista_variáveis \rightarrow identificador ('=' expressão)? (',' identificador ('=' expressão)?)*

Bloco: bloco \rightarrow '{' declaração* comando* '}'

Esta fase foi implementada juntamente com o analisado Sintático devido as suas verificações e facilidade de implementa-lo durante a implementação de cada função da gramática.

Ferramentas Utilizadas:

- ❖ 1- NetBeans
- ❖ 2- Java
- ❖ 3-GitHub
- ❖ 4- JFLAP



run:

LEXEMA

Linha

TOKEN

#

1

TOK_AST

include

1

include

<

1

Tok_menor

stdio.h

1

Biblioteca

>

2

Tok_maior

#

2

TOK_AST

include

2

include

<

2

Tok_menor

stdio.h

2

Biblioteca

>

4

Tok_maior

int

4

int

mai

4

TOK_ID

(

4

TOK_AP

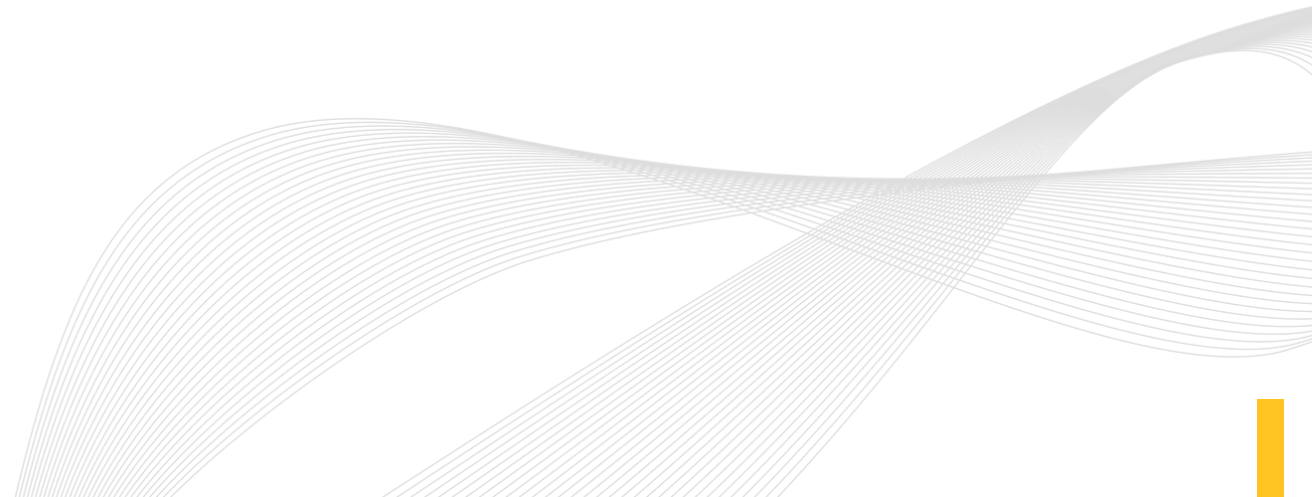
int

4

int

Não foi o resultado esperado na sua totalidade, mas foi satisfatório, isso devido a algumas partes (mínimas) que não tive me conta com por exemplo o printf e scanf.

Mas contudo o resultado para mim está satisfatório como dito anteriormente e é satisfatório.



run:

Erro de Sintaxe| Linha: 10| Esperava: ID

Erro de Sintaxe| Linha: 10| Esperava: ;

Erro de Sintaxe| Linha: 10| Esperava: ID

Erro de Sintaxe| Linha: 12| Esperava: atribuicao

Erro de Sintaxe| Linha: 12| Esperava: ID

Erro de Sintaxe| Linha: 13| Esperava: Inteiro

Erro de Sintaxe| Linha: 17| Esperava: Inteiro

Erro de Sintaxe| Linha: 17| Esperava: ;

Erro de Sintaxe| Linha: 17| Esperava: Inteiro

Erro de Sintaxe| Linha: 17| Esperava: ;

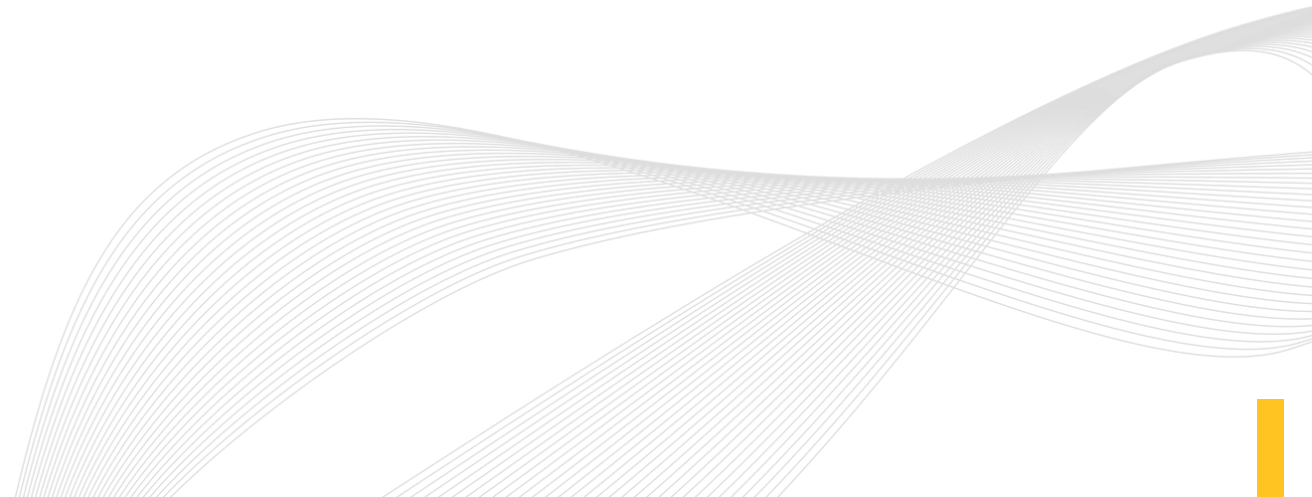
Erro de Sintaxe| Linha: 17| Esperava: Inteiro

Output



Não foi satisfatório!

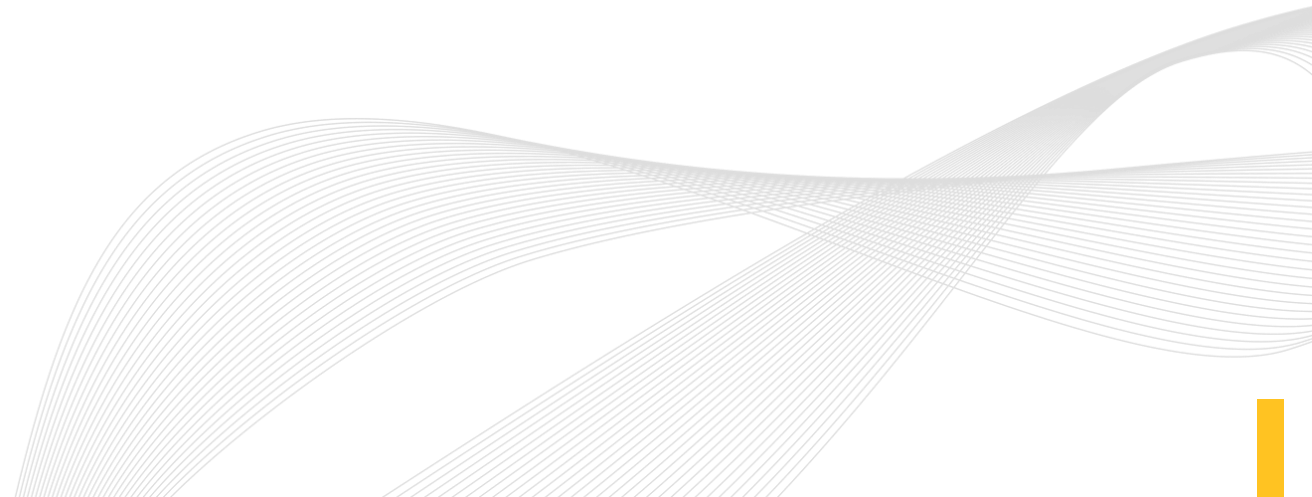
Os resultados foram muito abaixo do esperado, mesmo tendo organizado muito bem a gramática segui-la na sua totalidade foi uma tarefa muito desafiadora, pois o numero elevado de funções e as chamadas recursivas foram um tiro no pé! Sem contar que muitas expressões regulares eram realmente difíceis de serem implementadas.



```
run:
Erro de Semantico|      Linha: 6|      Esperava um Float
-----
Erro de Semantico|      Linha: 7|      Esperava um inteiro
-----
Erro de Semantico|      Linha: 8|      Esperava um inteiro
-----
Erro de Semantico|      Linha: 10|     ID invalido
-----
Erro de Semantico|      Linha: 12|     Variavel nao declarada
-----
```

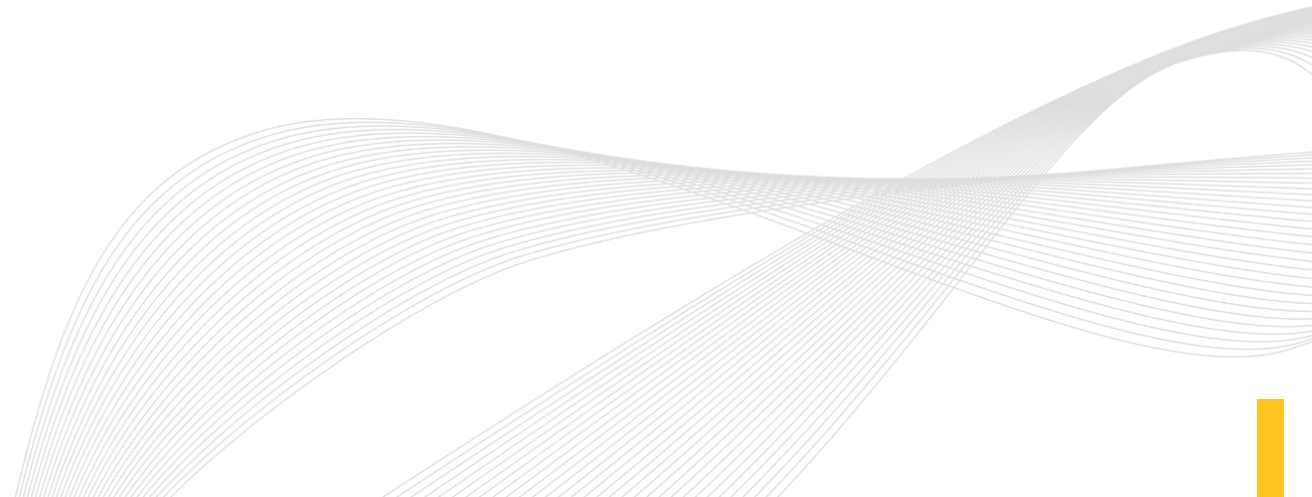
```
Numero de erros: 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
|
```



Satisfatório!

De longe a parte mais fácil de ser implementada, isso pois com algumas funções do sintático houve facilidade me aplicar as regras Semânticas. Por mais que tenha causado um pouco de confusão no código foi bem interessante implementá-la.



Bom pessoalmente foi uma experiência muito boa ver o que se passa por debaixo dos panos, aprendi muito, consegui me superar em muitos casos e as dificuldades me tornaram também um melhor programador de certo que agora olho com outra perspetiva o lado da programação , consigo utilizar melhor os recursos e aprender mais sobre indentificadores, funções e outros que compõem um programa em si e como a ide os interpreta.

No analisador léxico foi muito desafiante elaborar um autómato finito, visto que estava sempre num método de tentativa e erro. Mas foi muito desafiante trabalhar com estados e transforma-los em switch case.

Já no analisador Sintático como dito anteriormente foi um tiro no pé isso devido a sua complexidade e falta de apoio e rápida organização da gramática por parte dos colegas. Implementa-lo foi ainda mais desafiador e irritante também.

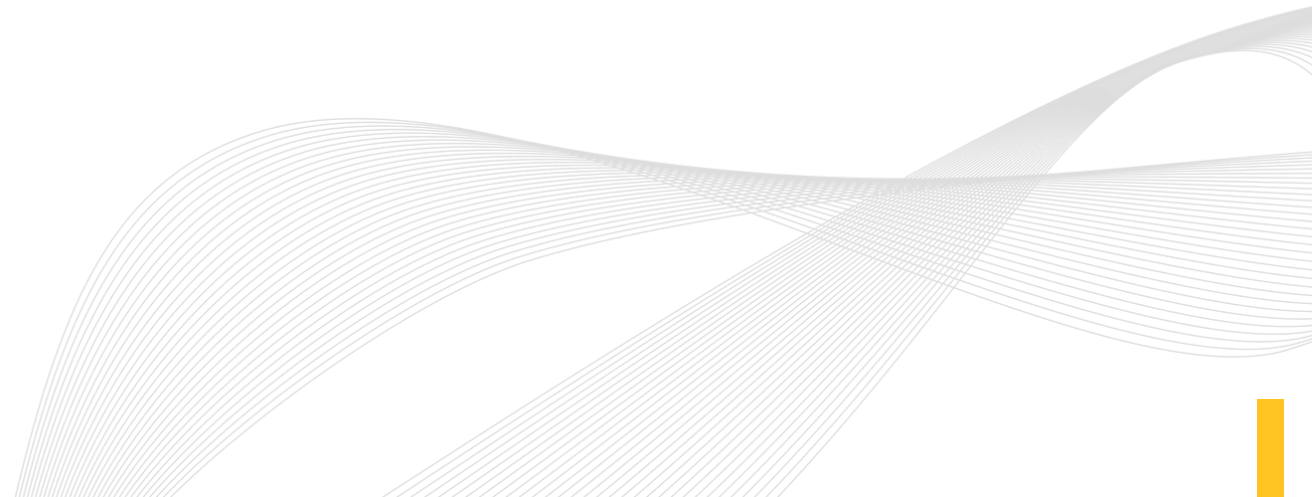
O analisador Semântico foi um pouco mais leve e divertido de implementar do que os outros.



Falando agora das dificuldades infelizmente foram além na área da programação e lógica. Problemas de saúde e perda de matérias contribuíram muito para o meu fraco desempenho em quase todas as cadeiras no geral.

Já a algum tempo que tenho enfrentado uma doença que até agora continuo a fazer a medicação, sofria de dores intensas que não me permitiam estar disposto a programar ou estudar qualquer coisa, algo que também me abalou muito é que as dores persistiam por muito tempo.

Outras dificuldades enfrentadas foram a perda do meu computador, a perda de um projecto já funcional e em alguns a falta de pessoas para cooperar para fazer trabalhos.



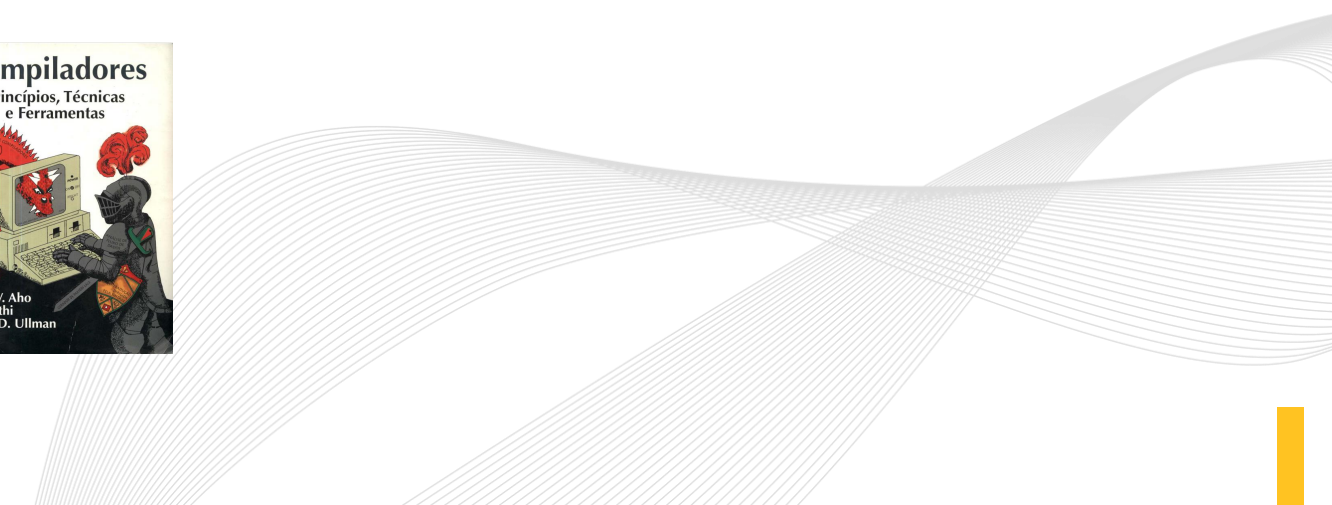
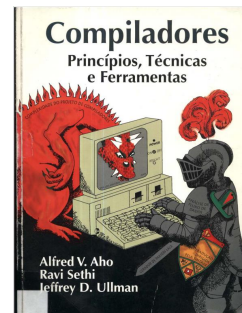
<https://ebaonline.com.br/blog/o-que-e-linguagem-c>

<https://idocode.com.br/blog/programacao/exemplos-e-aplicacoes-da-linguagem-c/>

<https://www.devmedia.com.br/top-10-linguagens-de-programacao-mais-usadas-no-mercado/39635>

<https://www.encyclopedia-crianca.com/desenvolvimento-da-linguagem-e-alfabetizacao/segundo-especialistas/desenvolvimento-da-linguagem-nos>

<https://chat.openai.com/chat>





MUITO OBRIGADO PELA ATENÇÃO!

ISPTec

“A quem quer, nada é difícil”