SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV
CAMPUS FLORESTAL

Trabalho prático 1 - Compiladores

Definição da linguagem “+O”

Alan Araújo dos Reis - 5096
Arthur Ataíde de Melo Saraiva - 5070
Eduardo Soares de Lima Filho - 4400
Gabriel Rodrigues Marques - 5097
Guilherme Augusto Schwann Wilke - 4685
Luiz Felipe Coutinho Bouchardet Santanna - 5256

Florestal - MG
2025

Sumário

1. Parte 1

Introdução

Esta documentação descreve o primeiro estágio de desenvolvimento da linguagem “+O” (pronunciada como “O'Plus”), que é uma linguagem procedural criada a partir de uma proposta temática baseada em um mundo fantástico, onde a alquimia prevalece sobre todas as teorias científicas e é aplicada amplamente na construção e manipulação de elementos. A linguagem tem como diferencial a execução invertida, onde a lógica dos comandos é organizada de forma retroativa: as operações são descritas antes do destino cujo valor será armazenado, subvertendo o fluxo convencional de controle. Ambas características são representadas no nome da linguagem, o qual é escrito de trás para frente e faz uma referência ao termo Opus (obra, em latim), utilizado no termo alquímico “Magnum Opus”, o qual representa a transmutação de substâncias materiais em ouro ou prata.

Decisões

Temática e ideia geral

Como ideia principal, decidimos que a escrita de um programa usando +O deveria ser semelhante à escrita de um documento de pesquisa alquímica, o que dá ao programador a experiência de um alquimista registrando suas grandes descobertas ao criar seus algoritmos. Dessa forma, utilizamos extensivamente de termos em Latim ou similares para definir palavras reservadas à linguagem. Além disso, também optamos por definir seu fluxo de escrita usando uma lógica invertida, onde a execução é declarada antes de condições, por exemplo, o que representa a prática alquímica em alcançar resultados por meio da experimentação antes mesmo da formulação das leis ou justificativas que os explicam. Decidimos usar como base para nossa linguagem a linguagem C, que é bem conhecida por todos integrantes e parece ser adequada para a implementação de +O.

Obstáculos

O conceito da linguagem foi criado a partir do interesse em comum dos integrantes do grupo quanto às histórias fantásticas, temas filosóficos e alquimia, entretanto, desde sua idealização, já identificamos algumas dificuldades potenciais que enfrentaremos durante o desenvolvimento.

A própria linguagem apresenta desafios únicos, começando pela escolha do latim como base, em contraste com a maioria das linguagens de programação, que utilizam o inglês. Além disso, pensar de forma diferente do convencional foi uma dificuldade significativa, especialmente porque, na linguagem +O, não há regras de precedência para operações aritméticas (a não ser que sejam explicitamente declaradas pelo programador, usando parênteses), elas são avaliadas da esquerda para a direita, seguindo a ordem de leitura, de forma semelhante ao que ocorre em competições matemáticas, onde o apresentador anuncia os números e as operações, e os participantes executam as contas nessa mesma sequência. Como muitas estruturas da +O seguem uma lógica distinta da que estamos acostumados em outras linguagens, isso tornou o processo de idealizar e escrever a linguagem consideravelmente mais desafiadora.

Definição da linguagem

Atribuição (-->)

Na atribuição de valores a variáveis, é utilizado o símbolo “-->”. Além disso, o valor a ser atribuído (ou a variável cujo valor será utilizado na atribuição) deve ser posicionado à esquerda do sinal de atribuição e a variável a qual o valor é atribuído à sua direita.

```
Unset
100 --> primeiroValor;
primeiroValor --> segundoValor;
```

Tipos primitivos

Os tipos primitivos definidos na linguagem +O e as palavras reservadas usadas para representá-los estão definidos na tabela abaixo:

Tipo primitivo	Palavra reservada
void	vacuum
int	atomus
float	fractio
double	fragmentum
bool	quantum
long int	magnus
short int	minimus
char	symbolum
string	scriptum

Tabela 1 - Representação de tipos primitivos usando palavras reservadas da linguagem +O

void (vacuum)

Para a declaração do tipo vazio, é possível utilizar a palavra reservada “vacuum”.

int (atomus)

Para números inteiros, é utilizada a palavra reservada “atomus”. As variáveis declaradas utilizando o tipo “atomus” podem ou não serem declaradas já com sua atribuição de valor ou não.

```
Unset
valorDoOuro atomus;
```

```
200 --> frascosDeFluido atomus;
10 --> pedrasFilosofais atomus;
```

float e double (fractio, fragmentum)

O tipo primitivo float (número com ponto flutuante) é representado pela palavra reservada “fractio”, assim como o tipo primitivo double float (número com ponto flutuante de precisão dupla) pode ser declarado usando a palavra reservada “fragmentum”.

```
Unset
moedasDePrata fractio;
tacasDeVinho fractio;
2.5 --> coposDeVeneno fractio;
3.141592653589793 --> pi fragmentum;
```

bool (quantum)

Para valores booleanos é utilizado a palavra reservada “quantum”, assim como os seus respectivos possíveis valores true (verdadeiro, representado pela palavra reservada “factum”) e false (falso, representado pela palavra reservada “fictum”).

```
Unset
estaChovendo quantum;
factum --> estouPensando quantum;
fictum --> estouComendo quantum;
```

long int e short int (magnus, minimus)

Para valores inteiros de grande tamanho (long int) ou pequeno tamanho (short int), são utilizadas, respectivamente, as palavras reservadas “magnus” e “minimus”.

```
Unset
minhasRiquezas magnus;
anosDeVida minimus;
9.223.372.036.854.775.800 --> graosDeAreia magnus;
```

char e string (symbolum, scriptum)

Para definição de caracteres foi designada a palavra reservada “symbolum”, além de podermos definir nessa linguagem um vetor de caracteres (string), representado pela palavra reservada “scriptum”.

Symbolum (char)

Um caractere representa um único símbolo, que pode ser um caractere simples ou um caractere especial (escape).

```
Unset
'o' --> oxigenio symbolum;
```

Scriptum (string)

São sequências de caracteres delimitadas por aspas duplas. Podem conter caracteres comuns e sequências de escape.

```
Unset
"Augustus" --> nome scriptum;
```

Operadores aritméticos

Para implementar os operadores aritméticos, decidimos manter o mesmo padrão utilizado em C, para facilitar a compreensão de algoritmos escritos com +O, com exessão do ;

Operação aritmética	Símbolo
Soma	+
Subtração	-
Multiplicação	*
Divisão	/
Módulo	%
Divisão Inteira	//
Potência	**
Xor	^

Tabela 2 - Representação de operações aritméticas usando símbolos da linguagem +O

```
Unset
15 --> diasSemDormir atomus;
10 --> diasSemComer atomus;
diasSemComer + diasSemDormir --> diasSemComerOuDormir atomus;
2.0 + 3.5 --> horas fractio;
2 ** 10 --> grandeValor magnus;
```

Operadores relacionais

Assim como os operadores aritméticos, decidimos manter o mesmo padrão de símbolos usado em C para os operadores relacionais, o que facilita a leitura dos algoritmos escritos em O+;

Operação relacional	Símbolo
Igualdade	==
Desigualdade	!=
Maior que	>
Maior ou igual a	>=
Menor que	<
Menor ou igual a	<=

Tabela 3 - Representação de operadores relacionais usando símbolos da linguagem +O

Operadores lógicos

Nesse caso, decidimos definir tanto palavras reservadas para cada um dos operadores lógicos como aceitar os símbolos já definidos em C, isso facilita a leitura dos códigos para programadores já acostumados com a convenção de C mas também permite a liberdade de tornar o código mais verboso, atendo-se à proposta da linguagem.

Operação lógica	Símbolo	
E lógico (and)	&&	et
OU lógico (or)	||	vel
NÃO lógico (not)	!	ne

Tabela 4 - Representação de operadores lógicos usando símbolos e palavras reservadas da linguagem +O

And (&&, et)

O operador lógico E (conjunção) é representado pela palavra reservada “et”, mas também pode ser expresso utilizando o símbolo &&

Or (||, vel)

O operador lógico OU (disjunção) é representado pela palavra reservada “vel”, mas também pode ser expresso utilizando o símbolo ||.

Not (!, ne)

O operador lógico NÃO (negação) é representado pela palavra reservada “ne”, mas também pode ser expresso utilizando o símbolo !

```
Unset
factum || fictum --> verdade quantum;
factum vel fictum --> verdade;

factum && fictum --> mentira quantum;
factum et fictum --> mentira;

!factum --> mentira;
ne factum --> mentira;
```

Definição de blocos de código

Assim como em C, decidimos manter a definição de blocos de código usando o símbolo “{” para o início de blocos de código e “}” para o fim de blocos de código.

Definição de precedência de operadores

Em +O não existe precedência de operadores por importância, a menos que o programador defina-a usando parênteses. Dessa forma, todas as operações são executadas da esquerda para a direita, ou seja, a operação “5 + 5 * 2” primeiro executará a soma “5 + 5” e depois a multiplicação entre o resultado anterior e o próximo valor “10 * 2”, o que resultará em 20, ao contrário de C, onde a operação de multiplicação seria executada antes da soma, que resultaria no valor 15.

Delimitador de string e char

Os literais de caracteres e strings são delimitados de forma semelhante à linguagem Python, os literais de string são delimitados por aspas duplas ("), enquanto os literais de char (um único caractere, correspondente ao tipo symbolum) são delimitados por aspas simples (').

```
Unset
'A' --> inicialDoNome symbolum;
"Bem-vindo ao laboratório!" --> mensagemBoasVindas scriptum;
```

Palavras reservadas

Palavra reservada em C	Respectiva palavra reservada em +O
while	persisto
for	iterare
if	si
else	non
else if	non si
switch	vertere
case	casus
default	axiom
break	ruptio
continue	continuum

Tabela 5 - Representação de palavras reservadas de C na linguagem +O

Definição de expressões lógicas ou condicionais usando parênteses

Para definir as expressões lógicas ou condicionais dos comandos “persisto”, “iterare” ou “si”, por exemplo, utilizamos os parênteses encapsulando as expressões condicionais ou lógicas.

while (persisto)

A estrutura de repetição while é implementada com a palavra-chave “persisto“. A sintaxe segue o formato (condição) persisto {bloco}, onde a condição é avaliada antes de cada iteração do bloco de código.

```
Unset
(valor1 <= 10) persisto {
valor1 + 1 --> valor1;
valor2 + 2 --> valor2;
}
```

for (iterare)

A estrutura de repetição for é implementada com a palavra-chave “iterare“. Sua sintaxe segue o padrão (inicialização; condição; incremento) iterare {bloco}, permitindo o controle preciso do número de iterações através de três expressões separadas por ponto e vírgula.

```
Unset
(i --> 0; i < 3; i + 1 --> i) iterare {
valor1 + 10 --> valor2;
valor1 + 1 --> valor1;
}
```

if (si), else (non) e else if (non si)

A estrutura condicional if-else é implementada com as palavras-chave “si“ e “non“. A sintaxe para o if é (condição) si {bloco}. A cláusula else é representada por non {bloco}. O else if é formado pela combinação das duas, como em non (outra_condição) si {bloco}.

```
Unset
(estaVivo == factum) si {
anosDeVida + 1 --> anosDeVida;
} non {
encarnacao + 1 --> encarnacao;
0 --> anosDeVida;
}

(anosDeVida <= 10) si {
"criança" --> faixaEtaria;
} non (anosDeVida <= 18) si {
"adolescente" --> faixaEtaria;
} non {
"adulto" --> faixaEtaria;
}
```

switch (vertere), case (casus) e default (axiom)

A estrutura de seleção switch é implementada com a palavra-chave “vertere“. A sintaxe é (variável) vertere {bloco}. Dentro do bloco, cada caso é definido por casus valor -> {bloco} e o caso padrão (default) é definido por axiom -> {bloco}.

```
Unset
(posicao) vertere {
casus 1 -> "primeiro lugar" --> resultado; ruptio;
```

```
casus 2 -> "segundo lugar" --> resultado; ruptio;
casus 3 -> "terceiro lugar" --> resultado; ruptio;
axiom -> "fora do pódio" --> resultado;
}
```

break (ruptio) e continue (continuum)

Os comandos de controle de laço break e continue são implementados, respectivamente, pelas palavras-chave “ruptio“ e “continuum“. O comando ruptio; encerra a execução do laço imediatamente, enquanto continuum; pula para a próxima iteração do laço.

```
Unset
(factium) persisto {
	(estaVivo == factum) si {
		anosDeVida + 1 --> anosDeVida;
		continuum;
	} non {
		ruptio;
	}
}
```

Separação de parâmetros

Para separar mais de um parâmetro passado na chamada ou declaração de uma função, utilizamos o símbolo pipe (“|”), o que é inserido entre dois parâmetros.

Estruturas e funções

Palavras reservadas ou identificadores em C	Respectivas palavras reservadas ou identificadores em +O
typedef	designare

struct	homunculus
enum	enumerare
Declaração de função	formula
Chamada de função	(parametro1 | parametro2)funcao1
return	redire
const	mol

Tabela 6 - Representação de estruturas e funções usando símbolos e palavras reservadas da linguagem +O

typedef (designare) e struct (homunculus)

A nomeação de estruturas através do typedef é feita através da palavra-chave “designare” e a definição de estruturas “struct” é feita com “homunculus”. Para apenas nomeação a sintaxe segue designare tipo atribuição e para estrutura nomeEstrutura {bloco} designare homunculus.

```
Unset
designare atomus int;
designare quantum bool;

NomeDaStruct {
       valor1 atomus;
       valor2 fractio;
} designare homunculus;
```

enum (enumerare)

A enumeração com enum é feita através da palavra-chave “enumerare”. A sintaxe sendo nomeEstrutura {bloco} Enumerare.

```
Unset
TipoInimigo {
   COMUM | RARO | CHEFE
```

```
} Enumerare;
```

Declaração (formula), chamada de função e return (redire)

Para declaração de funções temos a palavra reservada “formula”, a qual é seguida pela declaração de seus argumentos (encapsulados por parênteses e separados por “|”) e seu identificador. Após isso, é declarado o tipo do valor retornado pela função com o símbolo “-->” seguido pelo tipo primitivo respectivo. Após isso, é declarado o bloco de execução da função. Caso não haja retorno na função, o tipo primitivo declarado como retorno é “vacuum”.

```
Unset
formula (heroi Jogador | inimigo Jogador) heroiGanhaBatalha --> quantum {
       heroi.vida >= inimigo.vida redire;
}

(heroi | inimigo)heroiGanhaBatalha --> estaVivo;
```

Declaração de constantes (mol)

A declaração de constantes segue a lógica inversa de declaração de variáveis utilizando “mol”. Com a sintaxe de valor --> nome tipo mol.

```
Unset
2 --> pedra atomus mol;
"Augustus" --> nome scriptum mol;
```

Funções pré-definidas

Tivemos a liberdade de definir que existirão algumas funções pré-definidas em +O, baseadas em algumas funções existentes em C ou bibliotecas

complementares, sem que seja necessária a importação de uma biblioteca externa. Essas funções estão definidas abaixo:

Funções existente em C	Respectivas funções em +O
sizeof	magnitudo
scanf	lectura
printf	revelare

Tabela 7 - Representação de funções pré-definidas da linguagem +O

sizeof (magnitudo)

O operador sizeof é implementado pela palavra-chave “magnitudo”. Ele retorna o tamanho em bytes de um tipo de dado ou de uma variável/expressão. A sintaxe é (tipo_ou_variavel)magnitudo, e ele pode ser utilizado em qualquer lugar onde um valor é esperado.

```
Unset
(atomus * numero)magnitudo;
```

scanf (lectura)

A função de entrada de dados, similar ao scanf, é implementada pelo comando “lectura”. Sua função é ler um valor da entrada padrão e armazená-lo em uma variável. A sintaxe é (“%x”, variavel_destino£) lectura;.

```
Unset
numero atomus;
("%d", numero£)lectura;
```

printf (revelare)

A função de saída de dados, similar ao printf, é implementada pelo comando “revelare”. Sua função é avaliar uma expressão e exibir o resultado na saída padrão. A sintaxe é (expressao_ou_valor) revelare;.

```
Unset
("numero: %d" | numero)revelare;
```

Ponteiros e endereços

Definição de ponteiros

A definição de ponteiros é feita com o símbolo ° (representando um grau de ligação ou vínculo místico com uma variável).

Atribuição de Endereços

Para capturar a essência de uma variável (seu endereço na memória), utilizamos o operador £, representando a obtenção do espírito de uma entidade.

Acesso ao Valor Apontado

Para invocar o valor contido no local apontado por um ponteiro, também usamos o símbolo ° antes da variável ponteiro, assim como na desreferenciação da linguagem C.

Acesso a valores internos às estruturas de dados

Para acessar campos dentro de uma estrutura (homunculus), utilizamos o operador “ . ” (ponto final) como nas linguagens convencionais, representando a decomposição de um corpo em suas partes.

Acesso a Campos via Ponteiros para Estrutura

Existem duas formas de acessar membros de estruturas a partir de ponteiros, usando desreferenciação explicita (°a).campo e usando o operador de acesso direto =>, que representa a invocação direta de um atributo por meio do vínculo místico com a estrutura.

Operação	Em C	Em +O
Declaração de ponteiro	int* a;	a °atomus;
Atribuição de endereço	a = &b;	b£ --> a;
Desreferenciação	x = *a;	°a --> x;
Acesso a membro direto	a.nome;	a.nome
Acesso com desreferenciação	(a*).nome;	(°a).nome;
Acesso simplificado via ponteiro	a -> nome;	a => nome;

Tabela 8 - Representação de ponteiros na linguagem +O

Vetores

Vetores são declarados usando “<<>>” , denotando uma sequência de elementos estruturados. Ao serem declarados, seu tamanho é definido entre os símbolos “<<” e “>>”, assim como o tipo de seus elementos, que vem imediatamente após seu identificador e imediatamente antes do símbolo “<<”

```
Unset
vetor1 atomus<<N>>;
```

O acesso é feito de maneira usual:

```
Unset
array<<3>>;
```

Comentários (§)

Os comentários são delimitados pelo símbolo “§”, escolhido por sua associação simbólica com manuscritos antigos e alquimia, visto que ele é a representação gráfica do parágrafo.

Comentário de uma linha

Para inserir um comentário de linha única, utiliza-se o símbolo “§§”. Tudo o que vier após esse marcador, até o final da linha, será ignorado pelo compilador.

```
Unset
§§ Este é um comentário de linha
```

Comentário que contém múltiplas linhas

Para anotar comentários que se estendem por mais de uma linha, utiliza-se um bloco iniciado e encerrado por §. Todo o conteúdo entre os dois símbolos será considerado comentário.

```
Unset
§
Este é um comentário
de múltiplas linhas
§
```

Analisador léxico

O analisador léxico da linguagem +O foi desenvolvido utilizando o utilitário Flex, com o objetivo de identificar e categorizar os diferentes elementos léxicos da

linguagem inspirada em alquimia. Ele é responsável por transformar o código-fonte escrito na linguagem em um fluxo de tokens compreensível para as próximas etapas do processo de compilação.

Estrutura Geral

O arquivo lex.l contém três seções principais: declarações, regras e código auxiliar. Abaixo, apresentamos como essas sessões foram organizadas e utilizadas na construção do analisador.

Definições Regulares

As definições regulares da linguagem foram criadas utilizando alias para facilitar a leitura e manutenção do código. Entre os principais grupos definidos, temos:

Espaços e delimitadores:

```
7    /* ===== ESPAÇOS E DELIMITADORES ===== */
8    delim              [ \t\n]
9    ws                 {delim}+
```

Figura 1 - Espaços e delimitadores

Usados para ignorar espaços em branco e quebras de linha durante a análise léxica.

Literais:

```
11      /* ===== LITERAIS ===== */
12      letter              [a-zA-Z_]
13      digit               [0-9]
14      positive_digit      \+?[0-9]+
15      negative_digit      -[0-9]+
16      positive_real       \+?[0-9]+\.[0-9]+
17      negative_real       -[0-9]+\.[0-9]+
18      character_literal   \'([^\\\n]|(\\.))\'
19      string_literal      \"([^\\\n]|(\\.))*\"
```

Figura 2 - Literais

Essas expressões lidam com inteiros, reais, caracteres e strings, todos com sintaxe inspirada em transcrições de manuscritos antigos — com o uso de sinais explícitos para valores positivos e negativos, e delimitadores próprios.

Comentários:

```
21      /* ===== COMENTÁRIOS ===== */
22      comment_line        §§.*
23      comment_block       §([^§]*\n?)*§
```

Figura 3 - Comentários

Os comentários usam o símbolo § como referência à proteção de trechos de texto, remetendo a textos arcanos ou selados.

Tokens Reconhecidos

Os tokens reconhecidos foram organizados em categorias com clara referência à temática da linguagem:

Palavras-chave/reservadas:

```
72      /* ===== PALAVRAS-CHAVE ===== */
73      /* Tipos Primitivos */
74      true                factum
75      false               fictum
76      null                nulo
77      void                vacuum
78      long_int            magnus[ ]atomus
79      short_int           minimus[ ]atomus
80      int                 atomus
81      float               fractio
82      double              fragmentum
83      char                symbolum
84      string              scriptum
85      bool                quantum
86
87      /* Tipos de Dados */
88      struct              homunculus
89      enum                enumerare
90
91      /* Estruturas de Controle */
92      else_if             non[ ]si
93      else                non
94      if                  si
95      for                 iterare
96      while               persisto
97      switch              vertere
98      case                casus
99      default             axiom
100     break               ruptio
101     continue            continuum
102     return              redire
```

Figura 4 - Palavras chaves/reservadas (parte 1)

```
104     /* Funções */
105     function            formula
106     output              revelare
107     input               lectura
108
109     /* Outros */
110     import              evocare
111     typedef             designare
112     const               mol
113     size_of             magnitudo
```

Figura 5 - Palavras chaves/reservadas (parte 2)

Palavras-chave como atomus, fractio, homunculus, si, persisto, entre outras, representam os tipos primitivos e as estruturas de controle. Cada termo é uma referência direta a conceitos de alquimia ou latim clássico. As palavras-chave foram definidas como reservadas com o objetivo de simplificar a implementação da linguagem, evitando ambiguidades durante a análise sintática e garantindo uma estrutura léxica mais previsível e robusta.

Operadores:

A linguagem define uma vasta gama de operadores personalizados com símbolos inspirados em representações mágicas e simbólicas, como:

- ° (símbolo de ligação mística) para definição de ponteiros;
- £ para desreferência (acesso ao conteúdo de um ponteiro);
- ==> para obtenção de endereço (invocação simbólica).

Outros operadores como et, vel, aut, ne substituem operadores lógicos comuns (&&, ||, ^, !) em latim.

```
38      /* ===== OPERADORES ===== */
39      /* Ponteiros e Referências */
40      reference           \=\=\>
41      dereference         \£
42      pointer             \º
43      access              \.
44
45
46      /* Aritméticos */
47      exponentiation      \*\*
48      increment           \+\+
49      decrement           \-\-
50      integer_divide      \/\/
51      add                 \+
52      subtract            \-
53      multiply            \*
54      divide              \/
55      modulus             \%
56      assign              \-\-\>
57
58      /* Relacionais */
59      greater_equal       \>=
60      less_equal          \<=
61      equal               \==
62      not_equal           \!=
63      greater_than        \>
64      less_than           \<
65
66      /* Lógicos */
67      logical_and         (\&\&|et)
68      logical_or          (\|\||vel)
69      logical_not         (!|ne)
70      logical_xor         (\^|aut)
```

Figura 6 - Operadores

Identificadores

Os identificadores seguem a convenção padrão de linguagens C-like, permitindo letras, dígitos e underscores, mas sempre iniciando por uma letra ou underscore:

```
115     /* ===== IDENTIFICADORES ===== */
116     identifier          {letter}({letter}|{digit})*
```

Figura 7 - Identificadores

A construção do analisador léxico da linguagem **+O** buscou integrar aspectos técnicos e estéticos inspirados na alquimia e em textos místicos. O uso de símbolos únicos, palavras em latim e comentários com aparência arcana reforça a identidade visual e temática da linguagem. Além disso, a estrutura modular e comentada do analisador facilita futuras expansões e manutenções.

Compilação e execução do analisador léxico

Para facilitar o uso do analisador léxico desenvolvido, foi criado um sistema de compilação e execução automatizado com auxílio de um Makefile. Essa automação permite compilar rapidamente o analisador e executá-lo tanto com entrada direta pelo terminal quanto com leitura de arquivos. Além disso, há opções específicas para rodar o analisador em modo padrão ou com saída detalhada para depuração. Todos esses comandos são acessíveis de forma simples, garantindo praticidade durante o processo de testes e validação do funcionamento do analisador.

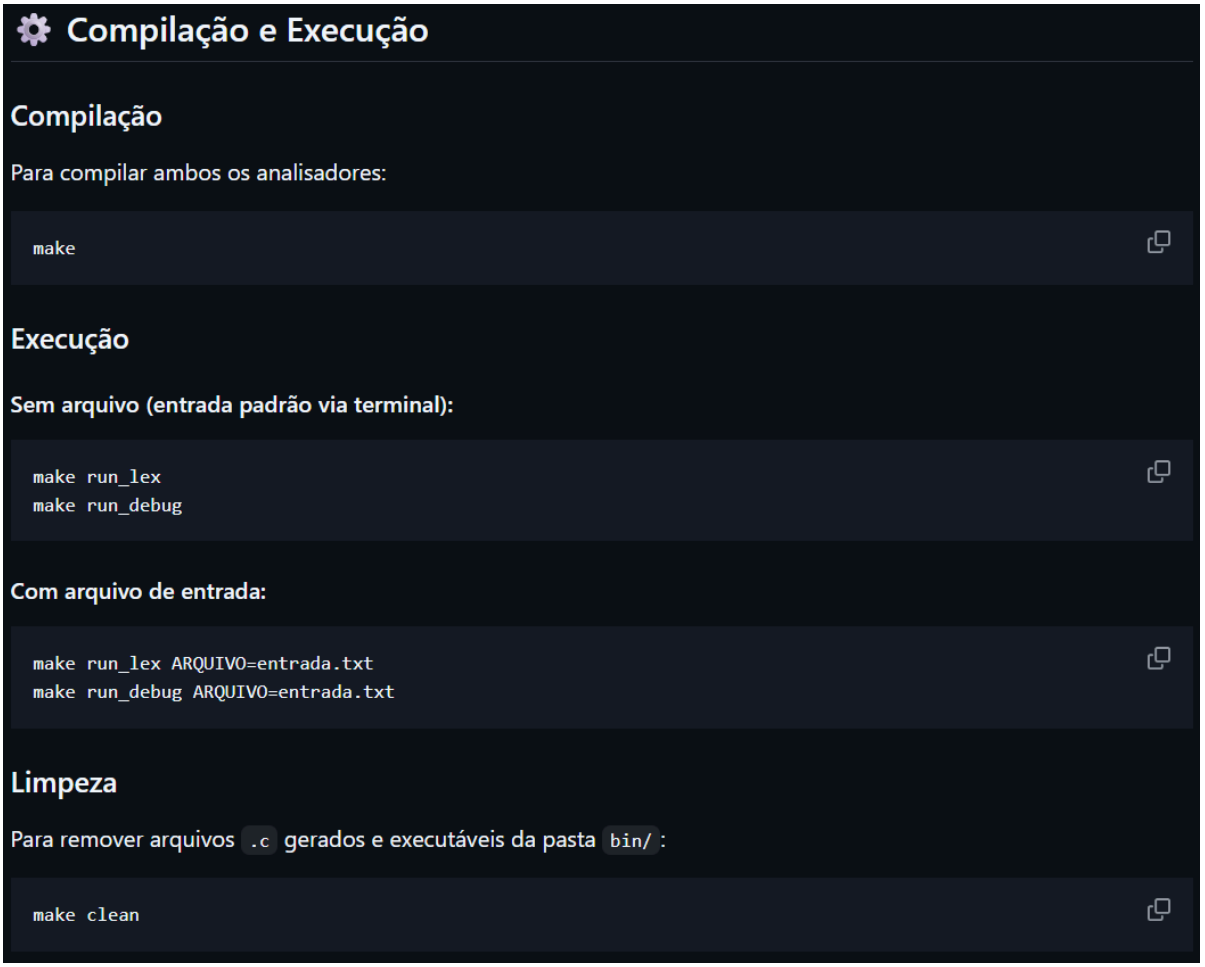

Figura 8 - Explicação da compilação e execução do analisador léxico

Conclusão

Após a idealização da linguagem e desenvolvimento de seu analisador léxico, é possível concluir que o resultado obtido foi satisfatório de acordo com a especificação do trabalho e conforme o desenvolvimento, foi possível praticar conceitos teóricos aprendidos em sala de aula.

A linguagem +O aplica uma lógica de execução invertida, termos em latim e fluxo de operações diferenciado, propõe uma experiência alternativa para os programadores que se sujeitam a entrar no mundo alquímico.

Ao utilizar do latim para a troca de nomenclaturas, como de palavras reservadas e operadores, é desenvolvida a semelhança entre essa linguagem e uma linguagem natural, assemelhando-se mais com a escritura de registros de um alquimista em suas criações.

Referências

1. LEVINE, J. R.; MASON, T.; BROWN, D. *A Compact Guide to Lex & Yacc*. Disponível em: https://courses.cs.umbc.edu/331/papers/compactGuideLexYacc.pdf. Acesso em: 24 maio 2025.

2. GABRIDULOL. *TP Compiladores*. GitHub Repository. Disponível em: https://github.com/gabridulol/tp-compiladores/tree/main. Acesso em: 24 maio 2025.

3. QUUT.COM. *The C Programming Language*. Disponível em: https://www.quut.com/c/. Acesso em: 24 maio 2025.

4. QUUT.COM. *ANSI C Grammar (Lex Specification) – 2011*. Disponível em: https://www.quut.com/c/ANSI-C-grammar-l-2011.html. Acesso em: 24 maio 2025.