



UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS FLORESTAL

Compiladores CCF 441

Vinícius Mendes - 3881
Artur Papa - 3886
Dener Vieira Ribeiro - 3872
Luciano Belo de Alcântara Júnior - 3897
Jhonata Miranda da Costa - 3859

Florestal
Maio de 2023

Contents

1	Introdução	2
2	Desenvolvimento	3
2.1	Tipos de dados primitivos	3
2.2	Palavras chaves e Palavras reservadas	4
2.3	Comandos	5
2.3.1	Básicos	5
2.3.2	Adicionais	5
2.4	Gramática da linguagem Endurance (para análise léxica)	6
2.4.1	Tokens	6
2.4.2	Identificadores	6
2.4.3	Palavras Chave	7
2.4.4	Constantes	8
2.4.5	Cadeia de Caracteres	9
2.4.6	Pontuadores	9
2.5	Analizador léxico	10
2.6	Gramática da linguagem Endurance (para análise sintática)	12
2.7	Analizador sintático	16
2.8	Tabela de símbolos	16
2.8.1	Implementação da tabela de símbolos em linguagem C	17
2.9	Compilando o projeto	18
2.10	Docker	19
2.10.1	Build	19
2.10.2	Run	19
2.10.3	Estrutura do Dockerfile	19
2.10.4	Exemplos	20
3	Extensão Visual Studio Code	23
3.1	Snippets	23
3.1.1	"block"	23
3.1.2	"comment"	23
3.1.3	"for"	24
3.1.4	"if-else"	24
3.1.5	"switch-case"	24
3.1.6	"while"	24
3.1.7	"main"	24
3.1.8	"enum"	24
3.1.9	"print"	24
3.1.10	"struct"	24
3.2	Sintaxe	25

1 Introdução

Este trabalho consiste na primeira de três etapas da criação de uma linguagem de programação. Para esta primeira parte do trabalho, foi pedido que definíssemos a nossa linguagem com informações como nome, origem do nome, tipos de dados primitivos, comandos e seus funcionamentos, palavras chaves e reservadas e gramática lexical. Além disso foi solicitado que fizéssemos o analisador léxico utilizando o LEX.

Em primeiro lugar foi feita a definição do assunto ser abordado no trabalho. Para isso, foram propostas várias ideias de tema para a linguagem, como mitologia nórdica ou grega, filosofia, dialetos africanos e piratas. Após uma conversa, foi de decisão unânime a escolha do tema de piratas. A partir disso, pensamos em alguns nomes para a linguagem, como nomes de navios ou personagens históricos dessa referida época, e, depois da análise dos possíveis nomes, foi definido que a linguagem se chamaria ***Endurance***.



Figure 1: Ícone da linguagem de programação

O nome ***Endurance*** faz referência ao navio da primeira expedição cujo objetivo era fazer a primeira travessia terrestre do continente antártico. Em agosto de 1914, o Endurance partiu do porto de Plymouth, no sudoeste da Inglaterra, com destino a Buenos Aires, na Argentina. Ali, mais pessoas se juntaram à tripulação para realizar a viagem à Antártida.[1]

De lá, o navio partiu em dezembro de 1914, mas quando chegou em janeiro do ano seguinte ao mar de Weddell, enfrentando ventos inclementes por seis dias, o navio ficou preso no gelo comprimido ao seu redor pela força da corrente de ar. Em fevereiro de 1915, os 28 marinheiros a bordo do Endurance perceberam que o gelo ao redor do navio não permitia que ele se movesse. Tentaram várias manobras, mas sem sucesso.[1]

Nos 10 meses seguintes, a tripulação ficou à mercê de ventos fortes e correntes marítimas que moveram a imensa massa de gelo em que estavam presos, de um lado para o outro. Em outubro, a pressão exercida pelo gelo no navio era tal que a água começou a penetrar no navio. Assim, não houve outra alternativa senão abandoná-lo e acampar na superfície congelada ao redor.[1]

Outrossim, no ano de 2022, 107 anos após a expedição feita pelo capitão Ernest Henry Shackleton, o Endurance foi encontrado no fundo do mar por pesquisadores.

Posto isso, baseado nessa história, o grupo optou pela escolha desse nome por ser emblemático e pelo fato do Endurance representar uma situação histórica marcante relacionada à viagens marítimas.



Figure 2: Tripulação do Endurance jogando uma partida de futebol no meio do gelo

2 Desenvolvimento

Para desenvolver a linguagem *Endurance*, nos baseamos principalmente nas linguagens C e Python. Por exemplo, a gramática da linguagem foi baseada na gramática lexical de C, e alguns operadores como `&&`, `||` e `!` foram substituídos por lexemas equivalentes a *and*, *or* e *not* (assim como é feito na linguagem Python). Ademais, as expressões usadas para os comandos e tipos de dados foram, em sua maioria, baseados em expressões piratas e contextualizadas para os lexemas de nossa linguagem.

2.1 Tipos de dados primitivos

Os tipos de dados primitivos são tipos de dados básicos incorporados e normalmente fornecidos pela própria linguagem. Representam unidades fundamentais de dados e são utilizados para armazenar e manipular valores simples. É importante notar que os tipos de dados específicos e as suas características podem variar consoante as linguagens de programação. Estes tipos de dados primitivos servem como blocos de construção para estruturas de dados mais complexas e permitem que os programadores manipulem e operem com valores de dados básicos de forma eficiente. A linguagem Endurance suporta 5 tipos de dados primitivos, sendo eles:

- `int` = `jib` [2]. Termo designado a uma vela triangular que se posiciona à frente do mastro dianteiro de um veleiro. Essa associação foi realizada visto que ela deve estar sempre **inteira** para o navio de mover
- `float` = `boat` [3]. Trocadilho com a palavra *float*.
- `double` = `ship` [4]. O dobro do `float` (em questão de tamanho).
- `bool` = `addled` [5]. Termo designado a um pirata confuso/maluco.
 - `true` = `aye`. Significa **sim** na linguagem de pirata.
 - `false` = `arr`. Expressão designada a um acontecimento negativo.
- `char` = `sailor` [6]. Trocadilho ao termo em inglês, *character*, que faz referência a um personagem. Nesse contexto, pode-se designar um majuro.

2.2 Palavras chaves e Palavras reservadas

As palavras-chave são palavras que têm um significado especial e, neste trabalho, serão reservadas pela linguagem de programação. São utilizadas para definir a estrutura, o fluxo e o comportamento de um programa. Geralmente, as palavras-chave não podem ser utilizadas como nomes de variáveis ou outros identificadores e na linguagem Endurance não é diferente. Aqui todas as palavras chaves são palavras reservadas, e abaixo temos uma lista com uma tradução das palavras chaves nesta linguagem:

- unsigned = windward [7]. Representa algo contra o vento, indicando algo que é contra valores que possuem sinal.
- signed = leeward [7]. O contrário de *windward*, representando algo a favor do vento, que aceita valores que tem sinal.
- short = cutlass [8]. Indica uma lâmina curta, comumente utilizada por piratas.
- long = musket [9]. Ao contrário do item acima, este significa uma arma de cano longo, de longo alcance.
- include = inport [10]. Brincadeira com *importar* e porto, sendo que o porto é local onde os navios atracam.
- main = oggin [11]. Significa *mar*, o ambiente principal da história.
- break = anchor [12]. Ancora, utilizada para frear o navio.
- case = trade [13]. Provém da expressão pirata *sweet trade*, que faz referência ao *switch case*.
- const = rigging [14]. São cordas e amarrações usadas para manter controle, estrutura, estabilidade. Essencial para a operação, controle e segurança de um navio. Combina praticidade, habilidade náutica e conhecimento de manejo de velas para garantir uma navegação eficiente e sobrevivência no mar.
- continue = mast [15]. Mastro, guia para continuar com a próxima iteração de um loop (helm ship's wheel)
- default = pirate
- if = spyglass [16]. Luneta, utilizada para verificar alguma informação.
- else = parley [17] Era uma conferência de negociação, especialmente entre inimigos sobre os termos de uma trégua ([18]) ou outros assuntos. A raiz da palavra parley é parler, que é o verbo francês [19] "falar"; especificamente a conjugação parlez "você fala", seja como imperativo ou indicativo.
- enum = crew [20]. Tripulação, para definir um conjunto de valores nomeados.
- return = land_ho [5]. Terra a vista! Hora de voltar.
- struct = chest. Local utilizado para "armazenar" outros itens.

2.3 Comandos

Um comando refere-se a uma instrução ou declaração específica que executa uma ação ou operação. Os comandos são utilizados para controlar o fluxo de execução, manipular dados e interagir com vários componentes de um programa. Aqui temos uma divisão entre comando básicos e comando adicionais fornecidos pela Endurance.

2.3.1 Básicos

- Comentário = /ahoy belay/. Ahoy: “olá”; belay: “cala-te”.
- do = weight_anchor. Significa levantar âncora, indicando a realização de algo.
- printf = parrot. Trocadilho com o fato dos papagaios conseguirem repetir uma fala humana.
- scanf = plunder Saquear, para receber dados de entrada.
- for = voyage Significar viajar/percorrer.
- goto = compass. Compasso, mostra aonde deve-se ir.
- sizeof = rummage. Revistar, para obter o tamanho de um tipo ou variável.
- switch = sweet trade. ”Negócio fácil”.
- typedef = charter. Contrato, que pode ser usado para definir tipos personalizados.
- union = treasure. Tesouro, uma combinação de diferentes tipos de valores.
- while = squall. Rajada de vento forte, uma repetição condicional já que o vento pode ou não se repetir. Enquanto houver vento, continue.
- free = swab. Designação para o marinheiro que limpa o convés.
- pointer = harpoon. Algo que aponta para algum local.

2.3.2 Adicionais

- slice de array = gully [21]. Faca comum entre os marinheiros e frequentemente usada em motins simplesmente porque não havia mais nada.
- some = hoard [22]. Tesouro acumulado, para realizar operações em uma coleção.
- reduce = booty [23]. Saqueando as coisas de um array. Permite reduzir uma coleção de valores a um único valor, aplicando uma operação acumulativa. Isso é útil para calcular somas, produtos, médias, entre outros.
- filter = net. Rede, para filtrar elementos de uma coleção. Permite filtrar os elementos de uma coleção com base em uma condição especificada, retornando uma nova coleção contendo apenas os elementos que atendem à condição. Isso é útil para filtrar dados indesejados ou selecionar elementos específicos de uma coleção.
- map = chart. Permite aplicar uma função a cada elemento de uma coleção, retornando uma nova coleção com os resultados transformados. Isso facilita a manipulação e a transformação.

- *sort* = *plunderhaul*. Essa sugestão combina os termos "plunder" (saquear) e "haul" (arrastar). Os piratas eram conhecidos por saquear tesouros e arrastar sua pilhagem para seus navios. O termo "plunderhaul" reflete a ação de coletar, arrastar e organizar os elementos de forma ordenada, assim como os piratas organizavam e avaliavam seu saque.

2.4 Gramática da linguagem Endurance (para análise léxica)

A seguir é apresentada a gramática lexical da linguagem Endurance. Esta gramática está organizada em forma de itens e sub-itens, sendo que as produções podem ser entendidas como *item deriva em sub-item*. Além disso, foi seguido o padrão utilizado em aula, as variáveis da linguagem são apresentadas em itálico, enquanto os terminais estão em negrito. Também apresentamos uma breve explicação de alguns tópicos importantes.

2.4.1 Tokens

Primeiramente temos a definição da variável *token*, que pode derivar nas variáveis *identificador*, *palavra-chave*, *constante*, *sequencia-caractere* e *pontuador*. Essa variável *token* é a variável de partida e será utilizada para classificar a entrada para que seja gerado o respectivo token para a saída.

- *token*:
 - *identificador*
 - *palavra-chave*
 - *constante*
 - *sequencia-caractere*
 - *pontuador*

2.4.2 Identificadores

Já os identificadores, possuem um padrão específico para serem reconhecidos. Eles podem ser constituídos por letras, o *underline*(_) e dígitos mas não podem começar com dígitos, sendo assim, a primeira produção garante que o que será retornado sempre terá como primeiro caractere algo diferente de um dígito.

- *identificador*:
 - *caractere-nao-digito*
 - *identificador caractere-nao-digito*
 - *identificador digito*
- *caractere-nao-digito*:
 - _
 - Letras de A a Z minúsculas ou maiúsculas
- *digito*:
 - Dígitos de 0 a 9

2.4.3 Palavras Chave

As palavras chave da linguagem estão relacionadas a palavras que tem um significado pre-definido e com propósitos específicos dentro da linguagem. Todas essas palavras chave, presentes no corpo da produção, são terminais dessa gramática.

- *palavra-chave*:
 - jib
 - boat
 - ship
 - addled
 - aye
 - arr
 - sailor
 - windward
 - leeward
 - cutlass
 - musket
 - inport
 - oggin
 - anchor
 - trade
 - rigging
 - mast
 - pirate
 - spyglass
 - parley
 - crew
 - land ho
 - chest
 - weigh_anchor
 - parrot
 - plunder
 - voyage
 - compass
 - rummage
 - sweet
 - trade
 - charter
 - treasure
 - squall
 - swab
 - harpoon

2.4.4 Constantes

As constantes em uma linguagem de programação são valores imutáveis em uma linguagem. Quando uma variável recebe um valor de uma constante, essa variável passa a assumir exatamente esse valor até que sofra uma nova atribuição. Constantes podem ser dos tipos: inteira, decimal, caractere, identificador ou pontuador. Ao utilizar constantes, os programadores podem tornar o seu código mais legível, mais fácil de manter e menos propenso a erros causados por modificações acidentais de valores.

- *constante*:
 - *constante-inteira*
 - *constante-decimal*
 - *constante-caractere*
 - *constante-identificador*
 - *pontuador*
- *constante-inteira*:
 - *numero-inteiro*
- *constante-decimal*:
 - *numero-decimal*
- *constante-caractere*:
 - *”sequencia-caractere”*
 - *’sequencia-caractere’*
- *constante-identificador*:
 - *identificador*
- *sequencia-digitos*:
 - *digito*
 - *sequencia-digitos digito*
- *sinal*:
 - *+*
 - *-*
- *numero-inteiro*:
 - *sequencia-digitos*
 - *sinal sequencia-digitos*
- *numero-decimal*:
 - *sequencia-digitos.sequencia-digitos*
 - *sinal sequencia-digitos.sequencia-digitos*



2.4.5 Cadeia de Caracteres

As cadeias de caracteres são conjuntos de símbolos limitados, que no caso da Endurance devem estar entre aspas duplas ou simples. Esta cadeia tem como objetivo representar dados, mais especificamente dados de texto. São sequências de caracteres e as linguagens de programação fornecem várias operações e métodos para trabalhar com cadeias de caracteres, tais como concatenação, extração, pesquisa e manipulação. As cadeias de caracteres são normalmente imutáveis e o seu comprimento pode ser determinado utilizando funções ou métodos incorporados. Compreender o manuseamento de cadeias de caracteres é essencial para trabalhar com dados baseados em texto na programação.

- *caractere*
 - Qualquer membro do conjunto do caractere de origem, exceto as aspas, duplas (") ou simples ('), a barra invertida ou o caractere de nova linha.
- *sequencia-caractere*
 - *caractere*
 - *sequencia-caractere caractere*

2.4.6 Pontuadores

Os pontuadores numa linguagem de programação são símbolos ou caracteres que definem a sintaxe, a estrutura e a organização do código. Servem como separadores, delimitadores e operadores, ajudando a agrupar, definir e controlar diferentes partes do programa. Os pontuadores desempenham um papel crucial na garantia de uma sintaxe apropriada e na organização do código de uma forma que seja compreensível tanto para os programadores como para os compiladores/intérpretes. É válido ressaltar que neste trabalho, alguns dos pontuadores utilizados pela Endurance não foram escritos em sua forma simbólica e foram trocados pelos seus nomes.

- *pontuador*
 - [
 -]
 - (
 -)
 - 
 - ;
 - .
 - **harpoon**
 - ++
 - --
 - +
 - -
 - *
 - /
 - &

- |
- ~
- %
- <<
- >>
- <
- >
- <=
- >=
- ==
- !=
- ^
- **rum**
- **grog**
- **sober**
- ?
- :
- ;
- =
- +=
- -=
- *=
- /=
- %=
- <<=
- >>=
- &=
- |=
- ^=
- ,

2.5 Analisador léxico

O analisador léxico é o programa capaz de fazer uma análise léxica, que por sua vez é o processo que transforma um fluxo de caracteres em um fluxo de tokens. No presente trabalho, foi utilizada a ferramenta FLEX (Fast LEXical analyzer generator) para gerar o analisador que será usado para processar a linguagem Endurance.

Gerar o analisador utilizando o FLEX traz vantagens sobre a criação do código de forma manual, já que basta identificar o vocabulário da linguagem e escrever as expressões regulares, que a própria ferramenta irá construir o analisador. Portanto, foi criado o arquivo `lex.l` onde são especificadas as expressões regulares para cada terminal (token) da linguagem de programação

Endurance. A Figura 3 apresenta parte arquivo lex contendo expressões regulares e algumas regras de tradução.

```

8  /* definicoes regulares */
9  ws                [ \t\n]
10
11  digit             [0-9]
12  positive          [+]?{digit}+
13  negative          [-]{digit}+
14  decimal_pos       [+]?{digit}+[.]{digit}+
15  decimal_neg       [-]{digit}+[.]{digit}+
16
17  low               [a-z]
18  upper             [A-Z]
19  word1q            ['"](^\\n|"\\\\"")*['"]
20  word2q            [""](^\\n|"\\\\"")*[""]
21  word              {word1q}|{word2q}
22
23  identifier        [A-Za-z_][A-Za-z0-9_]*
24
25  %%
26  \\ahoy.*belay\\    { printf("COMMENTS; LEXEMA: %s\\n", yytext); }
27
28  windward          { printf("UNSIGNED; LEXEMA: %s\\n", yytext); }
29  leeward           { printf("SIGNED; LEXEMA: %s\\n", yytext); }
30  cutlass           { printf("SHORT; LEXEMA: %s\\n", yytext); }
31  musket            { printf("LONG; LEXEMA: %s\\n", yytext); }
32  inport            { printf("INCLUDE; LEXEMA: %s\\n", yytext); }
33  oggin             { printf("MAIN; LEXEMA: %s\\n", yytext); }
34  anchor            { printf("BREAK; LEXEMA: %s\\n", yytext); }
35  trade             { printf("CASE; LEXEMA: %s\\n", yytext); }
36  rigging           { printf("CONST; LEXEMA: %s\\n", yytext); }
37  mast              { printf("CONTINUE; LEXEMA: %s\\n", yytext); }
38  pirate            { printf("DEFAULT; LEXEMA: %s\\n", yytext); }
39  spyglass          { printf("IF; LEXEMA: %s\\n", yytext); }
40  parley            { printf("ELSE; LEXEMA: %s\\n", yytext); }
41  crew              { printf("ENUM; LEXEMA: %s\\n", yytext); }
42  land_ho           { printf("RETURN; LEXEMA: %s\\n", yytext); }

```

Figure 3: Parte do analisador léxico

```

1  jib oggin() ⚓
2  /ahoy parrot("Caravel") belay/
3
4  'Galleon'
5
6  sailor endurance = "a";
7
8  spyglass(davy_jones != "b") ⚓
9  |   parrot("Blackbeard");
10 | ⚓; parley ⚓
11 |   parrot("Davy Jones");
12 | ⚓;
13 | ⚓;

```

Figure 4: Exemplo de entrada do analisador léxico.

```
• deneribeiro10@LAPTOP-DENER:endurance$ make test-third
./build/a.out < resources/third.end
INT; LEXEMA: jib
MAIN; LEXEMA: oggin
OPEN_PAREN; LEXEMA: (
CLOSE_PAREN; LEXEMA: )
BLOCK_OPEN; LEXEMA: ⚓
COMMENTS; LEXEMA: /ahoy parrot("Caravel") belay/
Foi encontrado uma string. LEXEMA: 'Galleon'
CHAR; LEXEMA: sailor
Foi encontrado um identificador. LEXEMA: endurance
ASSIGN; LEXEMA: =
Foi encontrado uma string. LEXEMA: "a"
SEMI_COLON; LEXEMA: ;
IF; LEXEMA: spyglass
OPEN_PAREN; LEXEMA: (
Foi encontrado um identificador. LEXEMA: davy_jones
NE; LEXEMA: !=
Foi encontrado uma string. LEXEMA: "b"
CLOSE_PAREN; LEXEMA: )
BLOCK_OPEN; LEXEMA: ⚓
PRINTF; LEXEMA: parrot
OPEN_PAREN; LEXEMA: (
Foi encontrado uma string. LEXEMA: "Blackbeard"
CLOSE_PAREN; LEXEMA: )
SEMI_COLON; LEXEMA: ;
BLOCK_CLOSE; LEXEMA: ⚓;
ELSE; LEXEMA: parley
BLOCK_OPEN; LEXEMA: ⚓
PRINTF; LEXEMA: parrot
OPEN_PAREN; LEXEMA: (
Foi encontrado uma string. LEXEMA: "Davy Jones"
CLOSE_PAREN; LEXEMA: )
SEMI_COLON; LEXEMA: ;
BLOCK_CLOSE; LEXEMA: ⚓;
BLOCK_CLOSE; LEXEMA: ⚓;
• deneribeiro10@LAPTOP-DENER:endurance$
```

Figure 5: Exemplo de execução do analisador léxico.

2.6 Gramática da linguagem Endurance (para análise sintática)

A gramática da linguagem Endurance necessitou de alterações e correções para que possa realizar corretamente a análise léxica e sintática de um programa Endurance. Esta gramática possui algumas definições regulares, como:

- **identifier** para **id**
- **positive**, **negative**, **decimal_pos** e **decimal_neg** para **number**
- **word** para **string**

Todas essas definições regulares podem ser encontradas no Lex, foram somente renomeadas para facilitar o entendimento da gramática. Abaixo apresentamos a gramática da linguagem Endurance:

$program \rightarrow stmts$

$stmt \rightarrow conditional$
 $\quad | repetition$
 $\quad | func$
 $\quad | var$
 $\quad | commands$
 $\quad | expr ;$
 $\quad | \text{\textbf{\char"26}} stmts \text{\textbf{\char"26}};$

$conditional \rightarrow \textbf{spyglass} (expr) stmt$
 $\quad | \textbf{spyglass} (expr) stmt \textbf{parley} stmt$
 $\quad | \textbf{sweet} (expr) \text{\textbf{\char"26}} caselist \text{\textbf{\char"26}};$

$caselist \rightarrow caselist \textbf{trade} term : stmts$
 $\quad | caselist \textbf{pirate} : stmts$
 $\quad | \epsilon$

$repetition \rightarrow \textbf{squall} (expr) stmt$
 $\quad | \textbf{voyage} (optexpr ; optexpr ; optexpr) stmt$
 $\quad | \textbf{weight_anchor} stmt \textbf{squall} (expr) ;$

$var \rightarrow type \textbf{id} vector ;$
 $\quad | type * pointer ;$

$pointer \rightarrow \textbf{id} vector$
 $\quad | * pointer$

$funcid \rightarrow \textbf{id}$
 $\quad | \textbf{oggin}$
 $\quad | \textbf{parrot}$
 $\quad | \textbf{plunder}$
 $\quad | \textbf{gully}$
 $\quad | \textbf{hoard}$
 $\quad | \textbf{booty}$
 $\quad | \textbf{net}$
 $\quad | \textbf{chart}$
 $\quad | \textbf{plunderhaul}$
 $\quad | \textbf{swab}$

$func \rightarrow type funcid (opttypelist) \text{\textbf{\char"26}} stmts \text{\textbf{\char"26}};$
 $\quad | type funcid (opttypelist) ;$

$typelist \rightarrow typelist , type \textbf{id} vector$
 $\quad | type \textbf{id} vector$

$termlist \rightarrow termlist , term$
 $\quad | term$

$opttypelist \rightarrow typelist$
 $\quad | \epsilon$

$optterm\ list \rightarrow term\ list$
 $\mid \epsilon$

$commands \rightarrow \text{land_ho } optexpr ;$
 $\mid \text{anchor ;}$
 $\mid \text{mast ;}$
 $\mid \text{charter } type\ id\ vector;$
 $\mid \text{inport string}$
 $\mid \text{chest } id\ \text{\textbf{\char"26}}\ varlist\ \text{\textbf{\char"26}};$
 $\mid \text{crew } id\ \text{\textbf{\char"26}}\ idlist\ \text{\textbf{\char"26}};$
 $\mid \text{treasure } id\ \text{\textbf{\char"26}}\ varlist\ \text{\textbf{\char"26}};$
 $\mid \text{compass } id :$
 $\mid id :$
 $\mid \text{rummage (} type\);$

$varlist \rightarrow varlist\ var$
 $\mid \epsilon$

$idlist \rightarrow id\ ,\ idlist$
 $\mid id$

$optexpr \rightarrow expr$
 $\mid \epsilon$

$stmts \rightarrow stmts\ stmt$
 $\mid \epsilon$

$expr \rightarrow attr\ assign\ expr$
 $\mid term\ op\ expr$
 $\mid term\ rel\ expr$
 $\mid term\ cond\ expr$
 $\mid \text{sober } expr$
 $\mid (expr)$
 $\mid term$

$assign \rightarrow =$
 $\mid +=$
 $\mid -=$
 $\mid /=$
 $\mid *=$
 $\mid \% =$
 $\mid << =$
 $\mid >> =$
 $\mid \& =$
 $\mid |=$
 $\mid ^=$

$op \rightarrow +$
 $\mid -$
 $\mid *$

- | /
- | %
- | &
- | |
- | ~
- | <<
- | >>
- | ^

rel → ==

- | !=
- | <
- | <=
- | >
- | >=

cond → **rum**

- | **grog**

term → **number**

- | *attr*
- | *boolean*
- | *funcid* (*optterm**list*)
- | **string**

attr → *id* **vector**

- | **id** *vector* . *attr*
- | **id** *vector* **harpoon** *attr*

boolean → **aye**

- | **arr**

vector → *vector* [*expr*]

- | €

type → *modifier* **jib**

- | *modifier* **sailor**
- | *modifier* **boat**
- | *modifier* **ship**
- | *modifier* **addled**
- | **chest** **id**
- | **crew** **id**

modifier → **windward**

- | **leeward**
- | **cutlass**
- | **musket**
- | **rigging**
- | €

2.7 Analisador sintático

Para a análise sintática, primeiro foi feito arquivos de exemplos tirados do guia disponibilizado para que fosse possível entender o funcionamento do Yacc. Tendo feito isso, primeiro o grupo fez a gramática correspondente da linguagem no arquivo `.y` com algumas pequenas alterações ao longo dos testes. Ademais, fizemos as inclusões dos arquivos `.h` necessários, bem como a definição do *union*, a declaração dos *tokens* e as *regras de precedência*.

Por fim, foi criada a tabela de símbolos e, no arquivo `translate.y` foi feita a inserção dos identificadores, a criação e exclusão dos blocos e a impressão da tabela de símbolos após a execução do programa, bem como a declaração dos tipos. Vale ressaltar, que essa segunda parte pode ser observada pelos comandos que estão entre chaves (`{}`), essas ações entre chaves servem para que o programa produza alguma saída com base em uma entrada. Em uma gramática Yacc/Bison, uma regra gramatical pode ter uma ação composta de declarações C. Cada vez que o analisador reconhece uma correspondência para essa regra, a ação é executada.

2.8 Tabela de símbolos

A tabela de símbolos é a estrutura de dados responsável por armazenar as informações dos identificadores presentes na linguagem, além de outras informações como tipo e localização na memória. A implementação da tabela de símbolos foi feita em linguagem C utilizando listas encadeadas, onde cada elemento da lista é uma segunda lista encadeada.

Essa implementação foi utilizada pois permite a criação de uma tabela de símbolos multi-nível. Assim, foi possível fazer uma representação bidimensional onde cada bloco é representado por uma linha e cada símbolo é representado como um elemento na linha. A Figura 6 mostra um exemplo de código com três blocos.

```
1  /ahoy início do bloco 1 belay/
2  jib a;
3  boat x;
4  ⚓ /ahoy início do bloco 2 belay/
5      sailor a;
6      ⚓ /ahoy início do bloco 3 belay/
7          boat z;
8          sailor y;
9          jib x;
10     ⚓; /ahoy fim do bloco 3 belay/
11 ⚓; /ahoy fim do bloco 2 belay/
12 /ahoy fim do bloco 1 belay/
```

Figure 6: Exemplo de declarações de variáveis multi-nível.

A partir do código exemplo apresentado na Figura 6 obtemos a tabela de símbolos teórica apresentada na Figura 7, onde são mostradas as listas encadeadas com dois tipos de nós, os de blocos (nós redondos) e os de símbolos (nós quadrados).

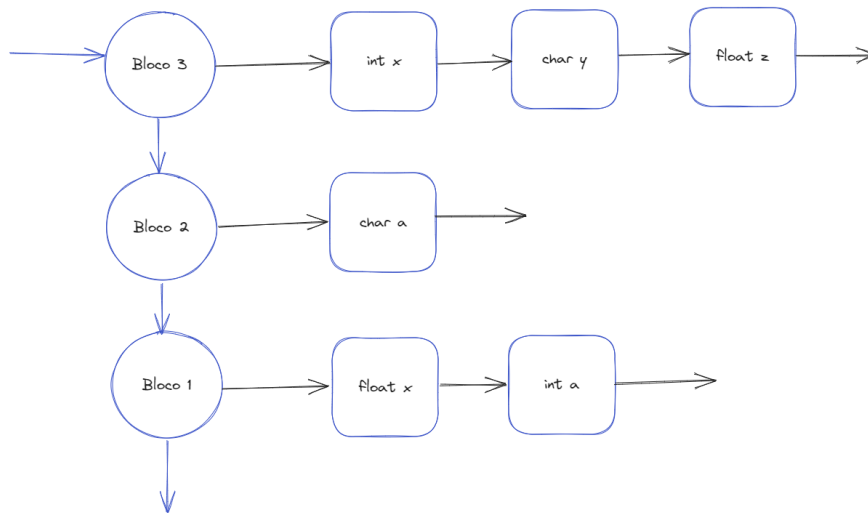


Figure 7: Tabela de símbolos visualizada como lista encadeada de listas encadeadas.

A figura 8 mostra o resultado que é apresentado no terminal para o usuário quando o código da Figura 6 é analisado sintaticamente. Note que esse é o resultado da tabela de símbolos antes do fechamento do Bloco 3. Em outros pontos do código, como ao final do Bloco 2 por exemplo, a tabela de símbolos conteria apenas 2 blocos (Bloco 1 e Bloco 2) e as variáveis x, y e z, definidas no bloco 3, não estariam presentes.

ID	Type	Address
x	INT	0x555b8ab66940
y	CHAR	0x555b8ab668a0
z	FLOAT	0x555b8ab66800
ID	Type	Address
a	CHAR	0x555b8ab66740
ID	Type	Address
x	FLOAT	0x555b8ab66680
a	INT	0x555b8ab665e0

Figure 8: Tabela de símbolos mostrada no terminal do usuário ao executar o analisador sintático.

A Figura 8 contém 3 tabelas, cada tabela representa um nível da tabela de símbolos, sendo que o primeiro nível é o bloco mais recentemente adicionado. Como a tabela de símbolos varia com o processo de análise do programa, optamos por mostrar a tabela de símbolos sempre que um bloco é fechado. Ou seja, será mostrado no terminal todas as variáveis acessíveis neste bloco logo antes deste ser fechado.

2.8.1 Implementação da tabela de símbolos em linguagem C

O arquivo de cabeçalho “symbolTable.h” é o arquivo que deve ser incluído para que se tenha acesso às funcionalidades e estruturas da tabela de símbolos, tais informações são listadas

abaixo:

- **SymbolTable**: Estrutura de dados da tabela de símbolos.
- **symbolTableNew()**: Aloca uma nova tabela.
- **symbolTableCreateBlock()**: Adiciona uma nova linha, ou seja, um novo bloco foi criado no programa fonte.
- **symbolTableDeleteBlock()**: Remove a linha mais recentemente adicionada, ou seja, foi fechado um bloco no programa fonte.
- **symbolTableInsert()**: Insere um ***Símbolo***.
- **symbolTableFind()**: Busca um Símbolo na tabela inteira, tendo como prioridade a busca em blocos mais recentemente adicionados.
- **symbolTableFindInBlock()**: Busca um símbolo no bloco atual.
- **symbolTableDelete()**: Desaloca a tabela de símbolos.
- **symbolTableShow()**: Escreve a tabela de símbolos em um arquivo de saída.

Além disso, o ***Símbolo*** foi implementado também como estrutura de dados e são listadas abaixo suas funcionalidades e estruturas:

- **Symbol**: Estrutura de dados de um Símbolo
- **Enumtypes**: Os tipos de dados que o um identificador pode representar no código, por exemplo, valor inteiro, decimal, nome de função, etc.
- **symbolNew()**: Aloca um novo símbolo.
- **symbolDelete()**: Desaloca o símbolo.

2.9 Compilando o projeto

Para compilar o projeto, basta digitar ***make*** no terminal na pasta raiz do projeto e um executável será gerado na pasta build. No ***makefile*** foram configurados alguns testes para facilitar a execução do programa, sendo eles:

- test-first
- test-second
- test-third
- test-input
- test-error

Assim, para executar os testes basta digitar ***make*** seguido do teste desejado, exemplo: ***make test-first***.

Os quatro primeiros testes e estão sintaticamente corretos e irão fazer a execução corretamente, mostrando o código-fonte e a tabela de símbolos ao final da execução. Já o *test-error* irá mostrar erro de sintaxe quando executado, irá mostrar também a linha do erro léxico do programa.

Além dos arquivos de exemplo, é possível compilar um exemplo qualquer seguindo os seguintes passos:

1. Execute o comando **make** na pasta raiz do projeto.
2. Execute o comando **make run FILE=<caminho_do_arquivo_de_entrada>**

2.10 Docker

Para facilitar a execução do projeto, é possível utilizá-lo com o Docker. O Docker é uma plataforma que permite empacotar e distribuir aplicações em ambientes isolados, chamados de contêineres. Isso garante que o software executará da mesma forma em diferentes sistemas operacionais.

2.10.1 Build

Para criar a imagem do Docker, execute o seguinte comando:

```
docker build . -t endurance
```

Isso criará uma imagem chamada *endurance* com base no Ubuntu 18.04 e instalará os pacotes necessários.

2.10.2 Run

Para executar o projeto dentro do contêiner Docker, utilize o comando:

```
docker run endurance
```

Isso executará o contêiner e rodará o projeto.

2.10.3 Estrutura do Dockerfile

A estrutura do Dockerfile utilizada para criar a imagem Docker é a seguinte:

```
# Imagem base: minima
FROM ubuntu:18.04

# Variável de ambiente para codificação
ENV LANG=C.UTF-8

# Instalando pacotes necessários
RUN apt-get update \
    && apt-get install -y flex gcc make bash bison \
    && rm -rf /var/lib/apt/lists/*

# Copiando arquivos fonte
COPY . /usr/src/endurance
WORKDIR /usr/src/endurance

RUN make

# Roda o Make por padrão
CMD ["bash", "./run.sh"]
```

Este Dockerfile utiliza a imagem base do Ubuntu 18.04 e instala os pacotes necessários (**flex**, **gcc**, **make**, **bash** e **bison**). Em seguida, copia os arquivos fonte do projeto para o diretório `/usr/src/endurance`, define o diretório de trabalho e executa o comando **make** para compilar o projeto. Por fim, o contêiner será iniciado executando o comando **bash run.sh** por padrão.

2.10.4 Exemplos

```
./build/a.out < resources/first.end
1: jib oggin() ⚓
2:   addled black_beard;
3:   black_beard = aye;
4:   sweet (black_beard) ⚓
5:     trade aye:
6:       /ahoy belay/
7:       anchor;
8:     trade arr:
9:       /ahoy belay/
10:      anchor;
11:    pirate:
12:      /ahoy belay/
13:  ⚓ ;
14: ⚓ ;
```

ID	Type	Address
black_beard	BOOL	0x55960eb55b70

ID	Type	Address
----	------	---------

Programa sintaticamente correto

Figure 9: Primeiro Teste

```

./build/a.out < resources/second.end
1: jib oggin() ⚓
2:   /ahoy  belay/
3:
4:   boat davy_jones;
5:   davy_jones = 3.33;
6:
7:   squall (davy_jones > 0) ⚓
8:     parrot("bora");
9:
10:    davy_jones-=1;
11:  ⚓ ;
12:
13:   land_ho 0;
14: ⚓ ;

```

```

=====
| ID          | Type          | Address          |
=====

```

```

=====
| ID          | Type          | Address          |
=====

```

```

| davy_jones  | FLOAT         | 0x5601a541cb70  |
=====

```

```

=====
| ID          | Type          | Address          |
=====

```

```

=====
| ID          | Type          | Address          |
=====

```

```

| davy_jones  | FLOAT         | 0x5601a541cb70  |
=====

```

```

=====
| ID          | Type          | Address          |
=====

```

Programa sintaticamente correto

Figure 10: Segundo Teste

```
./build/a.out < resources/third.end
1: jib oggin() ⚡
2:   /ahoy parrot("Caravel") belay/
3:
4:   sailor endurance;
5:   endurance = "a";
6:
7:   spyglass(endurance != "b") ⚡
8:     parrot("Blackbeard");
9:   ⚡ ; parley ⚡
10:    parrot("Davy Jones");
11:   ⚡ ;
12: ⚡ ;
```

ID	Type	Address
endurance	CHAR	0x5605dab33b70

ID	Type	Address
endurance	CHAR	0x5605dab33b70

ID	Type	Address
endurance	CHAR	0x5605dab33b70

Figure 11: Terceiro Teste

```
error test

./build/a.out < resources/error.end
1: jib oggin() ⚡
2:
3:     /ahoy Exemplo com erro belay/
4:
5:     jib num_1;
6:     jib num_2;
7:     jib sum;
8:
9:     {}
10:     sum = num_1 ++ num_2;
11:
12:     land_ho sum;
13: ⚡ ;
Token não reconhecido. LEXEMA: {
Token não reconhecido. LEXEMA: }
Erro próximo a linha 10: syntax error
```

Figure 12: Teste com erro

3 Extensão Visual Studio Code

A linguagem de programação desenvolvida possui uma extensão para o Visual Studio Code (VSCode) que já está em produção e disponível para download no Marketplace de extensões. É possível acessá-la através do seguinte link: [24].

Esta extensão adiciona suporte completo à linguagem de programação Endurance ao ambiente de desenvolvimento do VSCode, incluindo recursos como realce de sintaxe e snippets, que são trechos de código pré-definidos que podem ser inseridos automaticamente.

3.1 Snippets

A extensão Endurance para o VSCode fornece realce de sintaxe para facilitar a leitura e edição de código Endurance. Além disso, ela inclui vários snippets úteis que podem ser inseridos no código com apenas alguns cliques ou teclas de atalho. Cada snippet tem um prefixo associado que pode ser digitado no editor para acionar o snippet correspondente.

A seguir, explicarei cada um dos snippets disponíveis e sua finalidade, sendo que "prefixo" é como executamos o snippet, assim como já explicado:

3.1.1 "block"

- **Prefixo:** "block"
- **Descrição:** Insere um bloco de código Endurance.

3.1.2 "comment"

- **Prefixo:** "comment"
- **Descrição:** Gera um comentário no formato "/ahoy belay/".

3.1.3 "for"

- **Prefixo:** "for"
- **Descrição:** Cria um laço "for" em Endurance.

3.1.4 "if-else"

- **Prefixo:** "if-else"
- **Descrição:** Trecho de código para uma declaração "if-else" em Endurance.

3.1.5 "switch-case"

- **Prefixo:** "switch-case"
- **Descrição:** Trecho de código para uma declaração "switch-case" em Endurance.

3.1.6 "while"

- **Prefixo:** "while"
- **Descrição:** Trecho de código para um laço "while" em Endurance.

3.1.7 "main"

- **Prefixo:** "main"
- **Descrição:** Trecho de código para a função "main" em Endurance.

3.1.8 "enum"

- **Prefixo:** "enum"
- **Descrição:** Trecho de código para um enum em Endurance.

3.1.9 "print"

- **Prefixo:** "print"
- **Descrição:** Trecho de código para uma instrução de impressão em Endurance.

3.1.10 "struct"

- **Prefixo:** "struct"
- **Descrição:** Trecho de código para uma estrutura (struct) em Endurance.

Esses snippets são apenas alguns exemplos das funcionalidades fornecidas pela extensão Endurance para o VSCode. Eles podem ser muito úteis para acelerar o desenvolvimento de código Endurance, permitindo que você insira blocos de código comuns de forma rápida e fácil.

3.2 Sintaxe

A sintaxe na extensão do VSCode é definida em seções. Vamos analisar cada seção a seguir:

1. A primeira seção contém metadados sobre a extensão e a linguagem associada:
 - O campo **"name"** especifica o nome da linguagem, que é "Endurance" neste caso.
 - O campo **"scopeName"** define o escopo da linguagem, que é "source.endurance".
 - O campo **"fileTypes"** contém uma lista de extensões de arquivo associadas à linguagem, neste caso, apenas ".end".
 - Os campos **"patterns"** e **"repository"** são onde as definições da sintaxe da linguagem são especificadas, e vamos analisá-los em detalhes.
2. A seção **"patterns"** contém uma lista de padrões que serão aplicados para destacar elementos específicos do código-fonte:
 - Cada objeto na lista tem uma chave **"include"** que faz referência a um identificador correspondente a um padrão definido na seção **"repository"**. Os identificadores incluídos são **"comments"**, **"numbers"**, **"keywords"**, **"types"**, **"operators"**, **"strings"**, **"variables"**, **"functions"** e **"control-flow"**. Esses identificadores representam as diferentes categorias de elementos que serão destacados na sintaxe.
3. A seção **"repository"** contém as definições dos padrões mencionados acima:
 - Cada chave na seção **"repository"** corresponde a um identificador usado na seção **"patterns"**. Vamos analisar cada um desses identificadores e suas definições.
 - **comments**: Define o padrão de destaque para comentários. O padrão é definido usando a chave **"patterns"**, que contém uma lista de objetos. Neste caso, há apenas um objeto que define o início e o fim de um comentário em Endurance, usando as chaves **"begin"** e **"end"**. Além disso, o objeto tem a chave **"name"** que define o nome do token de destaque para comentários, que é "comment.endurance".
 - **numbers**: Define o padrão de destaque para números. Este identificador define dois padrões de destaque para números. O primeiro padrão é para números inteiros, usando a chave **"match"** e uma expressão regular. O segundo padrão é para números de ponto flutuante, usando a mesma chave **"match"** e uma expressão regular. Ambos os padrões têm a chave **"name"** que define o nome do token de destaque para números, que é "constant.numeric.endurance".
 - **keywords**: Define o padrão de destaque para palavras-chave. Este identificador define vários padrões de destaque para palavras-chave. Cada objeto na lista **"patterns"** contém uma chave **"match"** com uma expressão regular correspondente à palavra-chave e uma chave **"name"** que define o nome do token de destaque para palavras-chave específicas. Por exemplo, a palavra-chave "spyglass" é destacada com o token "keyword.control.endurance".
 - **types**: Define o padrão de destaque para tipos. Este identificador define padrões de destaque para tipos. Assim como as palavras-chave, os padrões são definidos usando as chaves **"match"** e **"name"**. Por exemplo, o tipo "jib" é destacado com o token "storage.type.endurance".
 - **operators**: Define o padrão de destaque para operadores. Este identificador define padrões de destaque para operadores. Os operadores são definidos como expressões regulares usando a chave **"match"**. Por exemplo, os operadores "+", "-", "—" e "/" são destacados com o token "keyword.operator.endurance".

- **strings**: Define o padrão de destaque para strings. Este identificador define padrões de destaque para strings. Existem dois padrões definidos: um para strings delimitadas por aspas simples e outro para strings delimitadas por aspas duplas. Cada padrão é definido com as chaves "**begin**" e "**end**", que especificam os caracteres de início e fim da string, e a chave "**name**", que define o nome do token de destaque para strings. O segundo padrão também contém um padrão interno, destacando strings delimitadas por aspas simples dentro de strings delimitadas por aspas duplas.
- **variables**: Define o padrão de destaque para variáveis. Este identificador define padrões de destaque para variáveis. O padrão é definido usando a chave "**match**" e uma expressão regular que corresponde a nomes de variáveis válidos em Endurance. As variáveis são destacadas com o token "variable.other.endurance".
- **functions**: Define o padrão de destaque para funções. Este identificador define padrões de destaque para funções. O padrão é definido usando a chave "**match**" e uma expressão regular que corresponde a nomes de função válidos em Endurance, seguidos por um parêntese de abertura. As funções são destacadas com o token "entity.name.function.endurance".
- **control-flow**: Define o padrão de destaque para fluxo de controle. Este identificador define padrões de destaque para palavras-chave de fluxo de controle, como "if", "else" e "while". Os padrões são definidos usando a chave "**match**" e uma expressão regular correspondente. Essas palavras-chave são destacadas com o token "keyword.control.flow.endurance".

Essas são as principais definições presentes no arquivo **syntax.json** para a linguagem Endurance. Elas especificam os padrões de destaque para diferentes elementos da sintaxe da linguagem, como comentários, números, palavras-chave, tipos, operadores, strings, variáveis, funções e fluxo de controle.

References

- [1] Endurance. <https://www.bbc.com/portuguese/internacional-60721784>. (Accessed on 19/05/2023).
- [2] Jib. <https://en.wikipedia.org/wiki/Jib>. (Accessed on 19/05/2023).
- [3] Boat. <https://en.wikipedia.org/wiki/Boat>. (Accessed on 19/05/2023).
- [4] Ship. <https://en.wikipedia.org/wiki/Ship>. (Accessed on 19/05/2023).
- [5] Pirate speak - expressions. https://piratesonline.fandom.com/wiki/Pirate_Speak_-_Expressions. (Accessed on 19/05/2023).
- [6] Sailor. <https://en.wikipedia.org/wiki/Sailor>. (Accessed on 19/05/2023).
- [7] windward. https://en.wikipedia.org/wiki/Windward_and_leeward. (Accessed on 19/05/2023).
- [8] cutlass. <https://en.wikipedia.org/wiki/Cutlass>. (Accessed on 19/05/2023).
- [9] musket. <https://en.wikipedia.org/wiki/Musket>. (Accessed on 19/05/2023).
- [10] port. <https://en.wikipedia.org/wiki/Port>. (Accessed on 19/05/2023).
- [11] Dicionario piratas. https://pt.wikibooks.org/wiki/Piratas/Dicion%C3%A1rio_das_Express%C3%B5es_dos_Piratas. (Accessed on 19/05/2023).
- [12] anchor. <https://en.wikipedia.org/wiki/Anchor>. (Accessed on 19/05/2023).
- [13] trade. <https://en.wikipedia.org/wiki/Trade>. (Accessed on 19/05/2023).
- [14] Rigging. <https://en.wikipedia.org/wiki/Rigging>. (Accessed on 19/05/2023).
- [15] Mast. [https://en.wikipedia.org/wiki/Mast_\(sailing\)](https://en.wikipedia.org/wiki/Mast_(sailing)). (Accessed on 19/05/2023).
- [16] Spyglass. <https://en.wikipedia.org/wiki/Spyglass>. (Accessed on 19/05/2023).
- [17] Parley. <https://en.wikipedia.org/wiki/Parley>. (Accessed on 19/05/2023).
- [18] Truce. <https://pirates.fandom.com/wiki/Truce>. (Accessed on 19/05/2023).
- [19] French. <https://pirates.fandom.com/wiki/French>. (Accessed on 19/05/2023).
- [20] Crew. <https://en.wikipedia.org/wiki/Crew>. (Accessed on 19/05/2023).
- [21] Gully. <https://pirates.hegewisch.net/gully.html>. (Accessed on 19/05/2023).
- [22] Hoards. <https://boattoursjohnspass.com/pirates-and-treasure-the-search-for-lost-pirate-hoards/>. (Accessed on 19/05/2023).
- [23] booty. <https://www.urbandictionary.com/define.php?term=booty%20pirate>. (Accessed on 19/05/2023).
- [24] Extensão endurance. <https://marketplace.visualstudio.com/items?itemName=endurance.endurance>. (Accessed on 18/06/2023).