UNIVERSIDADE TUIUTI DO PARANÁ

SABRINA ELOISE NAWCKI

VL-D+: UMA FERRAMENTA DE ESTUDO PARA COMPILADORES ATRAVÉS DE UMA INTERFACE

CURITIBA

2020

SABRINA ELOISE NAWCKI

VL-D+: UMA FERRAMENTA DE ESTUDO PARA COMPILADORES ATRAVÉS DE UMA INTERFACE

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Faculdade de Ciências Exatas e de Tecnologia, da Universidade Tuiuti do Paraná, como requisito à obtenção ao grau de Bacharel.

Orientador: Prof. Diógenes Cogo Furlan

CURITIBA

2020

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 **INTRODUÇÃO**

Os programas de computadores são entendidos pela máquina em uma linguagem de baixo nível, normalmente associada à notação binária, com sequências de símbolos 0 e 1 tendo diferentes significados.

Com a sofisticação da produção de programas de computadores estes deixaram de ser escritos diretamente em linguagem de máquina e passaram a ser construídos com o uso de abstrações e outras formas de signos relacionados à mente humana. Isto passou a exigir o uso de ferramentas para criar esta tradução entre as diferentes linguagens, da máquina e do homem. Uma destas ferramentas é chamada de compilador.

De forma simplificada, o compilador é a ferramenta que lê um programa, escrito em uma linguagem fonte, e o traduz em um programa equivalente em outra linguagem, mantendo a semântica original. (Aho, 1995).

Na disciplina de Compiladores são estudados técnicas e métodos para construir um compilador e por esta razão a disciplina se apresenta como uma das que possuem o maior grau de dificuldade de compreensão pelos discentes dos cursos de computação, pois envolve técnicas de difícil visualização e dependências de conteúdos de outras disciplinas.

Por causa desta dependência, o trabalho do docente em Compiladores é deveras complicado, pois exige do discente amplo conhecimento de disciplinas do currículo básico do curso, o que muitas vezes não é obtido com êxito.

Diante dessa premissa, devem-se criar meios que facilitem o aprendizado sem o auxílio direto de um docente, para que o discente obtenha um nível de introspecção maior do conteúdo da disciplina, de forma ampla e prática.

A internet dispõe de diversas formas comunicacionais, como imagens, vídeos, textos e outros, abrindo portas para o compartilhamento de informações. Assim, no meio didático pode-se criar ferramentas que auxiliem a construção do conhecimento com o seu uso.

Sendo assim, o objetivo desse trabalho de pesquisa é desenvolver e disponibilizar na *web* uma ferramenta que possa diminuir as possíveis dificuldades, que os discentes têm na disciplina de Compiladores.

O trabalho é dirigido às pessoas que se interessam em conhecer o funcionamento de um compilador e como o compilador pode ser implementado em uma ferramenta *web.*

A estrutura do trabalho é dividida em cinco partes: o capítulo 2 trata sobre teoria de compiladores; o capítulo 3 aborda a interface humano-computador; o capítulo 4 analisa os trabalhos relacionados e o capítulo 5 aborda a metodologia utilizada para o desenvolvimento da ferramenta VL-D+ (*Visual Learning D+*). A conclusão se encontra no capítulo 6. Os anexos pós-textuais se encontram após as referências.

2 **TEORIA DE COMPILADORES**

Este capítulo apresenta de forma resumida a teoria de compiladores, seu conceito, estrutura e os processos necessários para o seu funcionamento.

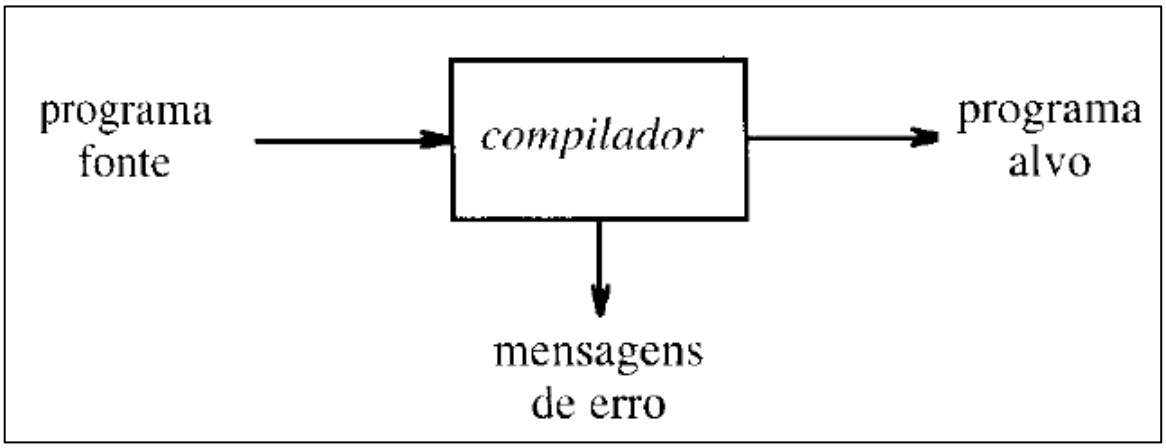
De acordo com Aho (1995), o compilador é um programa que processa comandos escritos numa linguagem fonte e os traduz para uma linguagem equivalente cuja notação possa ser executada no computador.

Para Santos (2018) um compilador é uma aplicação complexa que traduz uma descrição em determinada linguagem fonte para uma descrição equivalente em determinada linguagem de destino. Em outras palavras, um compilador traduz um programa em uma linguagem de programação para um programa executável em linguagem de máquina de uma arquitetura de destino.

Adicionalmente à tradução, os compiladores executam diversas funções auxiliares, que de acordo com Neto (2016) são atividades utilitárias para o usuário, encarregadas pela geração de listagens e a detecção, o diagnóstico e a emissão de mensagens e erros.

A figura 1 apresenta a definição de um compilador.

FIGURA 1 - Compilador



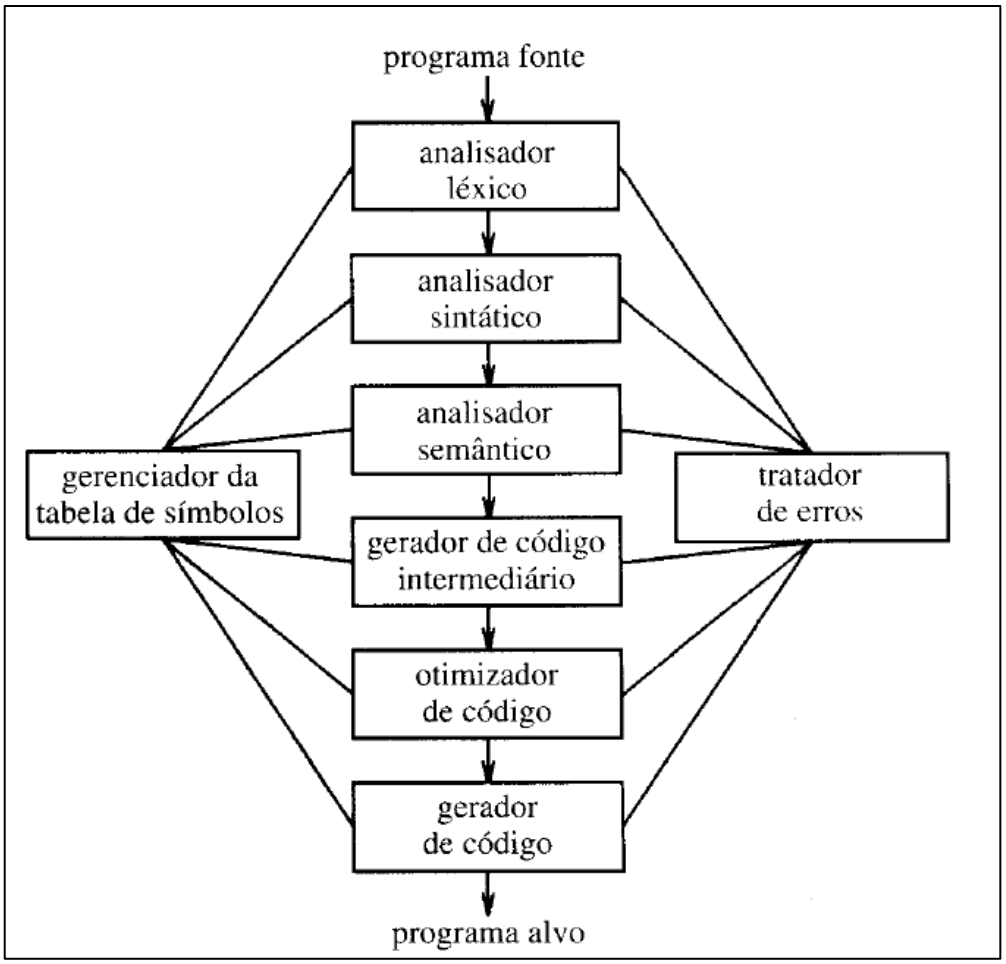
FONTE: AHO, SETHI, ULLMAN (1995)

Como pode ser visto na figura 1 o compilador recebe o programa fonte, que é o programa escrito pelo programador; em seguida é realizado o processo de tradução, que

caso seja bem-sucedido, retornará o programa alvo, e caso falhe, retornará as mensagens de erro ao programador.

Aho (1995) assume que o processo de tradução dos compiladores tipicamente é dividido nas fases: análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização de código e geração de código. A ordem e a relação destas fases são apresentadas na figura 2.

FIGURA 2 - Fases de um compilador



FONTE: AHO, SETHI, ULLMAN (1995)

Este capítulo é dividido da seguinte forma: a seção 2.1 apresenta o conceito de gramática junto à especificação da gramática D+, que é a gramática utilizada pelo compilador desenvolvido neste trabalho; as seções 2.2 a 2.7 apresentam as fases do compilador e a tabela de símbolos e a seção 2.8 apresenta a diferenciação entre compiladores e interpretadores.

2.1 GRAMÁTICA

O passo inicial na construção de um compilador é a definição da Linguagem de Programação (LP) que será utilizada na escrita dos programas fontes. A LP utilizada deve ser especificada corretamente e de preferência sem ambiguidades para que seja possível programar um compilador correspondente.

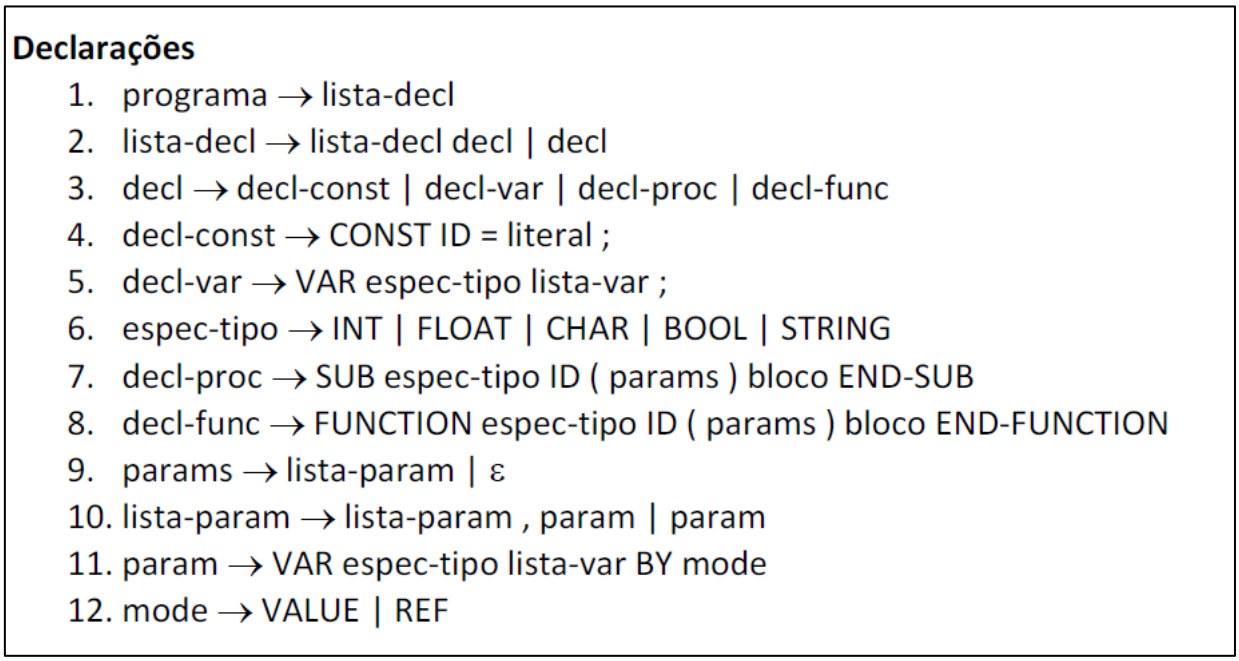
A formalização de uma LP é feita através de regras de formação. O conjunto destas regras é denominado de gramática. De acordo com Neto (2016) são definidas por gramáticas as linguagens construídas a partir de um conjunto de leis ou regras de formação que permitem a produção de textos sintaticamente corretos.

Para os compiladores, a gramática tem um papel importante de definir a estrutura hierárquica das construções da linguagem de programação e permitir a separação de sequências de caracteres como um único *token*, chamados de lexemas.

2.1.1 Gramática D+

A linguagem de programação D+, criada pelo professor Diógenes C. Furlan para a disciplina de compiladores, é uma linguagem de cunho didático, cuja sintaxe mescla elementos das linguagens BASIC, Pascal e C.

FIGURA 3 - Declarações na Gramática D+



FONTE: FURLAN (2019)

A figura 3 apresenta as regras gramaticais para as declarações de constantes, variáveis, procedimentos, funções e parâmetros da Linguagem D+.

Como pode ser visto nas regras 1 e 2, um programa é uma lista de declarações. A regra 3 apresenta as possíveis declarações que são especificadas nas regras 4, 5, 7 e 8. A regra 6 apresenta os possíveis tipos de variáveis que podem ser declarados. A regra 9 estabelece que os parâmetros podem ser uma lista de parâmetros individuais, podendo esta lista ser vazia. A regra 10 estabelece que uma lista de parâmetros pode ser formada por vários parâmetros separados por vírgula ou apenas um parâmetro. A regra 11 define como declarar um parâmetro. A regra 12 apresenta os possíveis modos de passagem dos parâmetros.

FIGURA 4 - Comandos na Gramática D+

Comandos

13. bloco → lista-com
14. lista-com → comando lista-com | ε
15. comando → cham-proc | com-atrib | com-selecao | com-repeticao | com-desvio | com-leitura | com-escrita | decl-var | decl-const
16. com-atrib → var = exp ;
17. com-selecao → IF exp THEN bloco END-IF | IF exp THEN bloco ELSE bloco END-IF
18. com-repeticao → WHILE exp DO bloco LOOP | DO bloco WHILE exp ; | REPEAT bloco UNTIL exp ; | FOR ID = exp-soma TO exp-soma DO bloco NEXT
19. com-desvio → RETURN exp ; | BREAK ; | CONTINUE ;
20. com-leitura → SCAN (lista-var) ; | SCANLN (lista-var) ;
21. com-escrita → PRINT (lista-exp) ; | PRINTLN (lista-exp) ;
22. cham-proc → ID (args) ;

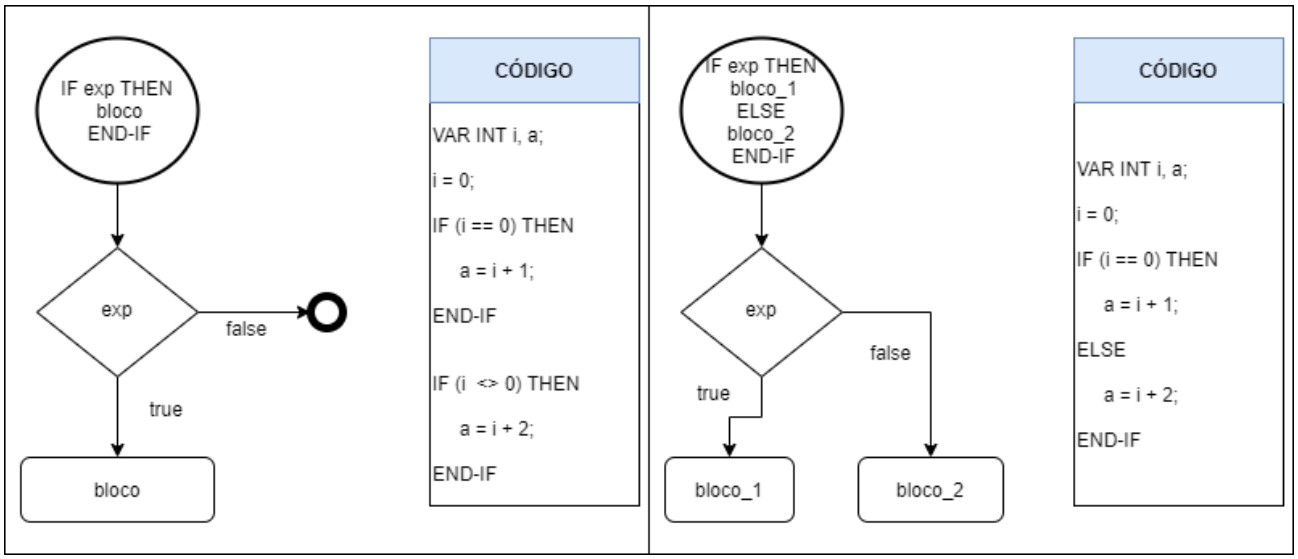
FONTE: FURLAN (2019)

A figura 4 apresenta as regras gramaticais dos comandos da gramática D+, sendo eles: chamada de procedimento, atribuição, seleção, repetição, desvio, leitura, escrita, declaração de variáveis e declaração de constantes.

Como pode ser visto nas regras 13 e 14, um bloco é uma lista de comandos. A regra 15 apresenta os possíveis comandos: chamada de procedimento, atribuição, seleção, repetição, desvio, leitura, escrita, declaração de variáveis e declaração de constantes. As regras 16 até a 22 apresentam como os comandos da regra 15 devem ser escritos.

A programação de um comando de seleção é possível de duas formas diferentes: somente com o caso verdadeiro, e com ambos os casos. A semântica destas formas é apresentada na figura 5.

FIGURA 5 - Comandos de seleção



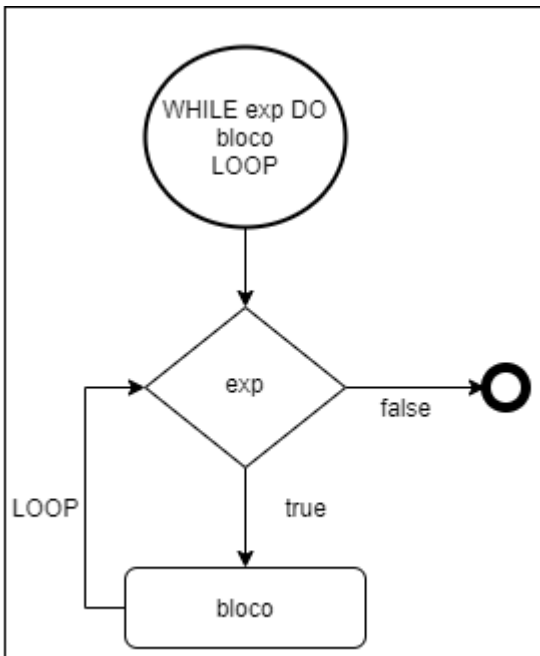
FONTE: a própria autora

A figura 5 apresenta um exemplo de como programar a mesma função utilizando os dois comandos de seleção da gramática D+. Ao lado esquerdo de cada quadrante é apresentado o diagrama do comando e ao lado direito é apresentado o código de exemplo.

Para a programação de um comando de repetição são disponibilizadas quatro maneiras diferentes: 1- enquanto uma condição for verdadeira faça alguma coisa; 2- faça alguma coisa enquanto uma condição for verdadeira; 3- repita alguma coisa até que uma condição seja verdadeira e 4- para um valor até uma expressão ser verdadeira faça alguma coisa.

As figuras 6, 7, 8 e 9 exemplificam a semântica de cada uma maneira de realizar o comando de repetição. Todos os exemplos implementam uma função que deve incrementar a variável ‘i’ até que o valor dela seja igual a ‘5’. Ao lado esquerdo de cada figura é apresentado o diagrama do comando e ao lado direito é apresentado o código de exemplo.

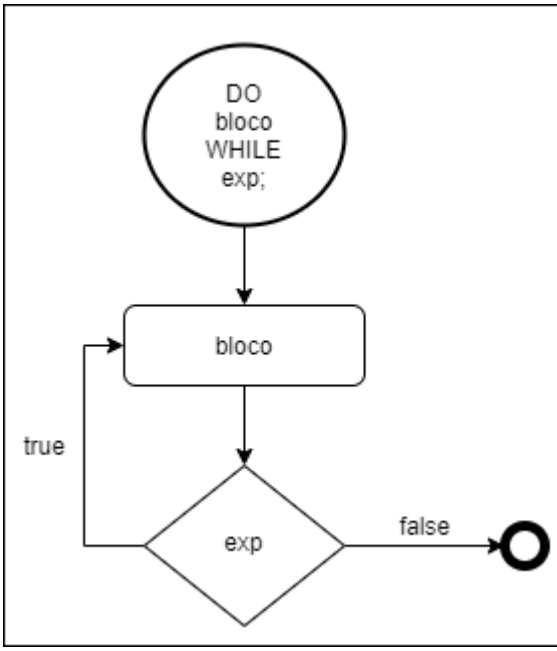
FIGURA 6 - Comando de repetição: enquanto



CÓDIGO	TESTE DE MESA	
	ID	Valor
VAR INT i;	i	0
i = 0;	i	1
WHILE (i < 5) DO	i	2
i = i + 1;	i	3
LOOP	i	4
	i	5

FONTE: a própria autora

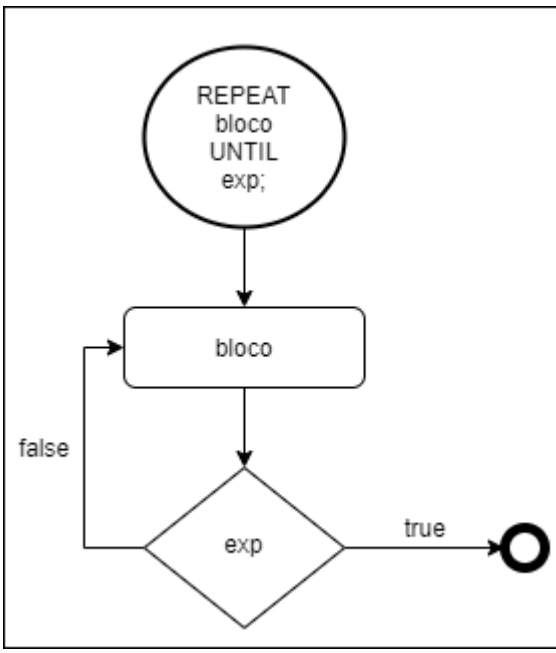
FIGURA 7 - Comando de repetição: faça



CÓDIGO	TESTE DE MESA	
	ID	Valor
VAR INT i;	i	0
i = 0;	i	1
DO	i	2
i = i + 1;	i	3
WHILE (i < 5);	i	4
	i	5

FONTE: a própria autora

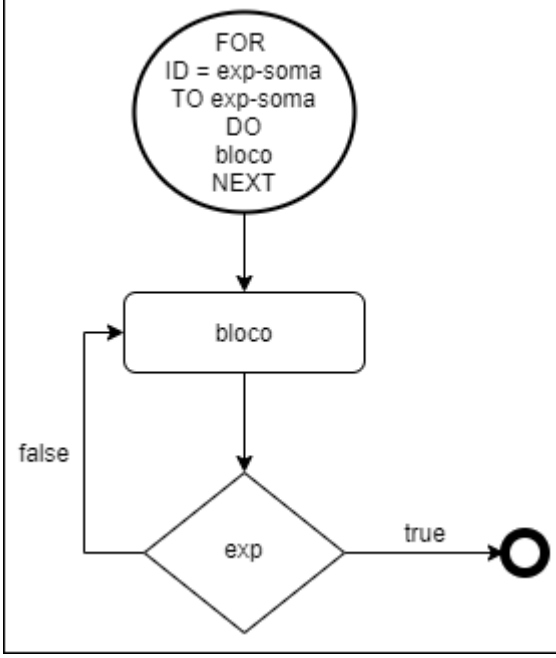
FIGURA 8 - Comando de repetição: repita



CÓDIGO	TESTE DE MESA	
	ID	Valor
VAR INT i;	i	0
i = 0;	i	1
REPEAT	i	2
i = i + 1;	i	3
	i	4
UNTIL (i == 5);	i	5

FONTE: a própria autora

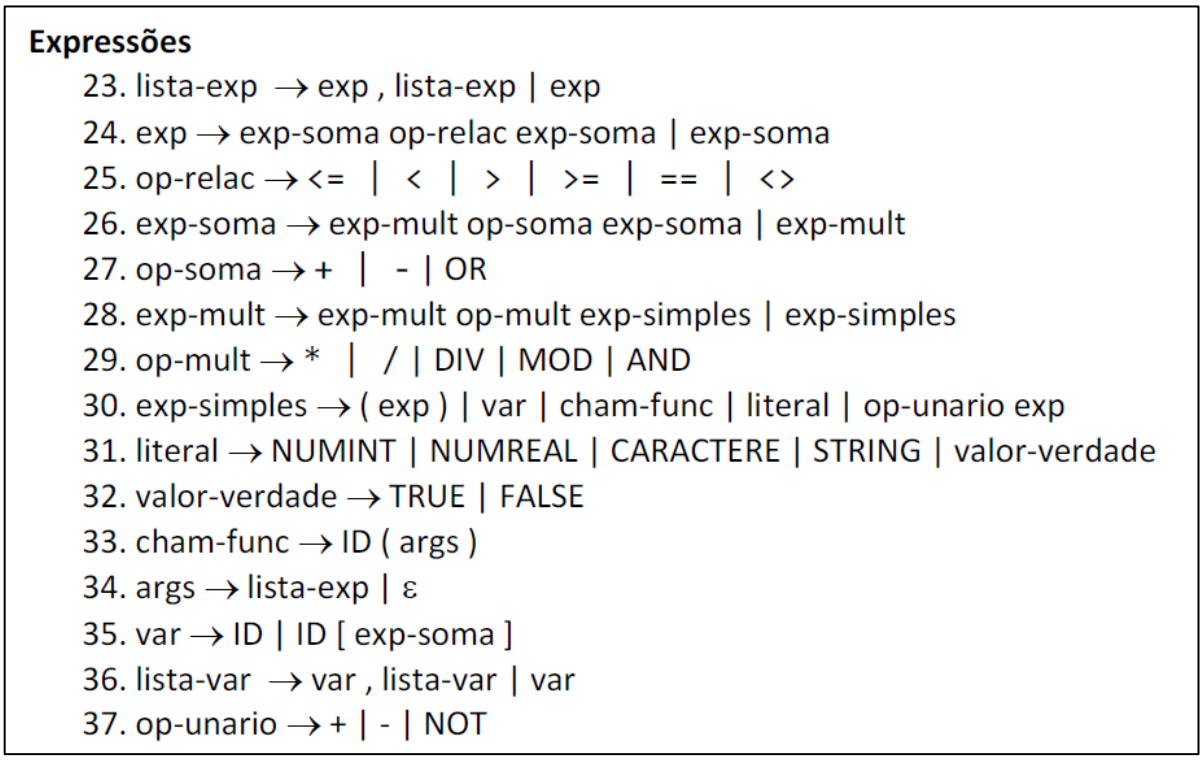
FIGURA 9 - Comando de repetição: para



CÓDIGO	TESTE DE MESA	
	ID	Valor
VAR INT i;	i	0
FOR i = 0 TO i < 5 DO	i	1
i = i + 1;	i	2
	i	3
NEXT	i	4
	i	5

FONTE: a própria autora

FIGURA 10 - Expressões na Gramática D+



FONTE: FURLAN (2019)

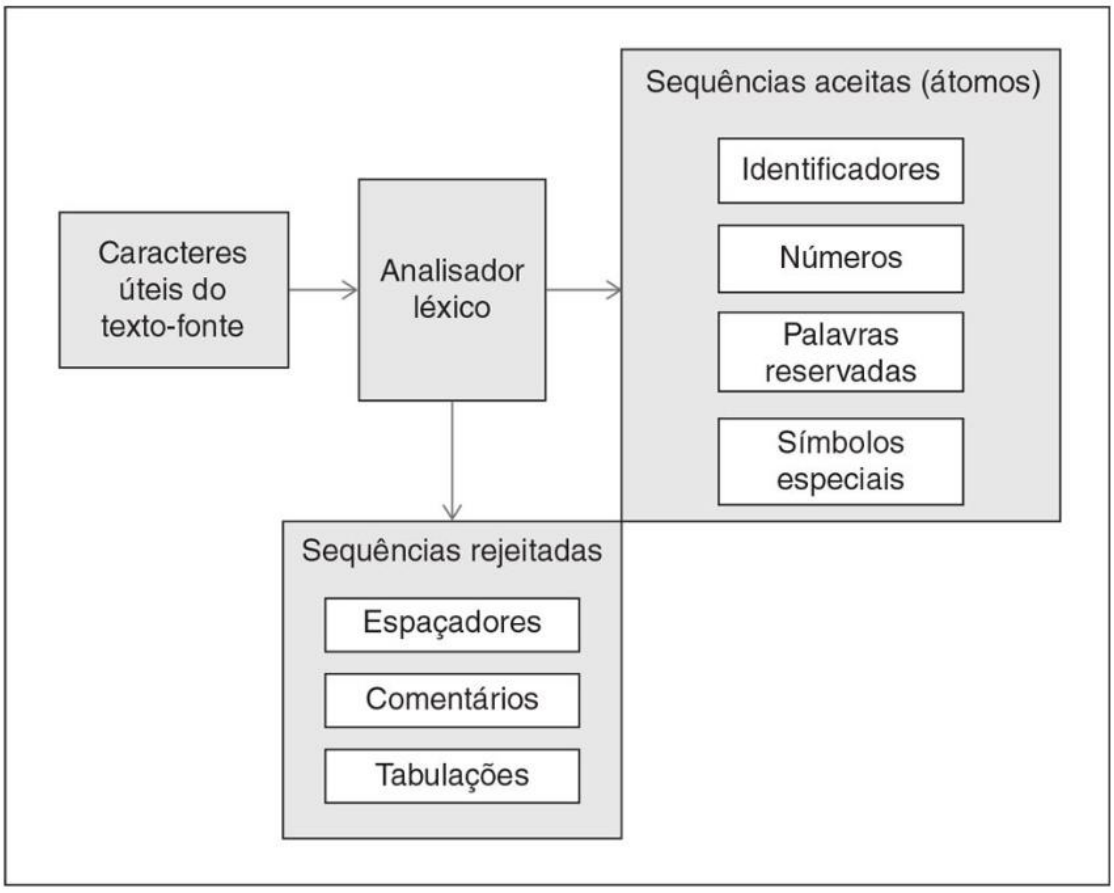
A figura 10 apresenta as regras gramaticais das expressões. As expressões podem conter operações de soma, operações de multiplicação, operações relacionais, operações unárias e chamadas de funções.

A regra 23 estabelece que uma lista de expressões é composta por uma ou mais expressões separadas por vírgula. As regras 24, 26, 28 e 30 definem as expressões. As regras 25, 27, 29 e 37 definem os operadores aceitos pela linguagem. A regra 31 apresenta os possíveis valores para um literal. A regra 32 apresenta os possíveis valores verdade. A regra 33 apresenta a chamada de função que deve ser composta por um identificador seguido por argumentos entre parênteses. A regra 34 apresenta que um argumento é composto por uma lista de expressões ou pode ser vazio. A regra 35 apresenta que uma variável por ser um identificador ou um identificador seguido por uma expressão de soma entre colchetes. A regra 36 apresenta que uma lista de variáveis pode ser composta por uma variável seguida por uma lista de variáveis separadas por vírgula ou por uma variável.

2.2 ANÁLISE LÉXICA

A análise léxica é a primeira fase do compilador e de acordo com Neto (2016) é a responsável pelo agrupamento dos caracteres válidos do programa fonte e pela classificação desses caracteres de acordo com a sua constituição.

FIGURA 11 - Análise léxica



FONTE: NETO (2016)

A figura 11 apresenta o processo do analisador léxico, que recebe como entrada o texto fonte e o separa em sequências aceitas e em sequências rejeitadas.

As sequências aceitas são identificadas de acordo com as regras definidas pela gramática da linguagem utilizada pelo programa fonte. As sequências rejeitadas são separadas pela função que descarta as sequências de caracteres que não contribuem na análise do programa fonte, como: tabulações, mudanças de linha, espaços, caracteres de controle, comentários e outros.

Além das funções principais posteriormente citadas, de acordo com Neto (2016) a análise léxica também conduz ações auxiliares, como: conversões numéricas, identificação de palavras reservadas e a criação e manutenção das tabelas de símbolos, definidas na seção 2.6, que são largamente utilizadas no processo da compilação.

2.2.1 Reconhecimento de *tokens*

No processo de compilação os termos "*token*", "padrão" e "lexema" são comumente utilizados. A análise léxica é a fase responsável pela identificação de cada sequência de caracteres como um desses termos.

Aho (1995) descreve o padrão como um conjunto de cadeias de entrada para as quais o *token* é produzido como saída e o lexema é o conjunto de caracteres que é reconhecido pelo padrão de um *token*. A figura 9 exemplifica o uso de cada um dos termos citados.

QUADRO 1 - Exemplo de *tokens*, lexema e padrão

Token	Lexemas Exemplo	Descrição Informal do Padrão
PR_CONST	pi	const travessão opcional seguido por letra seguida por letras, número e/ou travessões
PR_IF	if	if
OP_MAIOR	>	>
IDENTIFICADOR	contador	travessão opcional seguido por letra seguida por letras, número e/ou travessões
NUM_INT	120	qualquer constante numérica inteira
CARACTERE	'c'	qualquer símbolo entre aspas simples, exceto aspa simples

FONTE: AHO, SETHI, ULLMAN (1995) adaptado

No quadro 1 são apresentados diversos exemplos de *tokens* captados pela análise léxica de um programa escrito na linguagem D+ e seus respectivos lexemas e descrições informais, tais como: os *tokens* identificadores (*id*), que são os valores que começam com um travessão ou uma letra seguida por caracteres alfanuméricos ou travessões; os *tokens* numéricos (*num*), que são qualquer constante numérica inteira; entre outros.

A captura dos lexemas é feita caractere por caractere, de acordo com a sequência lida, e, da gramática da linguagem do programa fonte. Quando um lexema é completado, é possível inferir qual a sua classificação ou gerar o seu respectivo *token*.

2.2.2 Exemplo de programa e sua análise léxica

Para exemplificação do funcionamento da análise léxica, foram criados dois cenários de teste na linguagem D+.

O primeiro cenário de teste é um código funcional que efetua o cálculo de fatorial de um número, esse código é apresentado na figura 12.

FIGURA 12 - Código em D+ para cálculo de fatorial

```
1   /* Código na linguagem D+ para efetuar cálculo de fatorial de um número */
2   MAIN ()
3
4       VAR INT fatorial, numero;
5       numero = 3;
6       fatorial = 1;
7       WHILE numero > 1 DO
8           fatorial = fatorial * numero;
9           numero = numero - 1;
10      LOOP
11
12  END
```

FONTE: a própria autora

A análise léxica do código da figura 12 identifica vários *tokens* e seus respectivos lexemas, de acordo com a definição da gramática da linguagem D+, que podem ser vistos no quadro 2.

QUADRO 2 - *Tokens* e lexemas obtidos pela análise léxica do código de fatorial

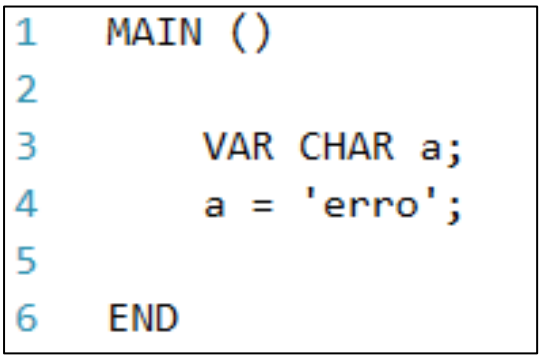
Token	Lexema
PR_MAIN	MAIN
SIN_PAR_A	(
SIN_PAR_F	)
PR_VAR	VAR
PR_INT	INT
IDENTIFICADOR	fatorial
SIN_V	,
IDENTIFICADOR	numero
SIN_PV	;
IDENTIFICADOR	numero
OP_ATRIBUI	=
NUM_INT	3
SIN_PV	;
IDENTIFICADOR	fatorial
OP_ATRIBUI	=
NUM_INT	1
SIN_PV	;
PR_WHILE	WHILE
IDENTIFICADOR	numero
OP_MAIOR	>
NUM_INT	1
PR_DO	DO
IDENTIFICADOR	fatorial
OP_ATRIBUI	=
IDENTIFICADOR	fatorial
OP_MULTI	*
IDENTIFICADOR	numero
SIN_PV	;
IDENTIFICADOR	numero
OP_ATRIBUI	=
IDENTIFICADOR	numero
OP_SUBTRAI	-
NUM_INT	1
SIN_PV	;
PR_LOOP	LOOP
PR_END	END

FONTE: a própria autora

O quadro 2 apresenta todos os *tokens* e seus respectivos lexemas obtidos através do código de cálculo de fatorial pela análise léxica. Porém, pode-se observar que espaços, quebra de linhas, tabulações e comentários são ignorados.

O segundo cenário de teste é um código que contém um erro léxico, esse código é apresentado na figura 13.

FIGURA 13 - Código em D+ com erro léxico



FONTE: a própria autora

A análise léxica do código apresentado na figura 13 deve disparar um erro na linha 4 do código, pois o tipo caractere (CHAR) de acordo com a gramática da linguagem D+ suporta apenas um símbolo ou nenhum símbolo entre aspas simples.

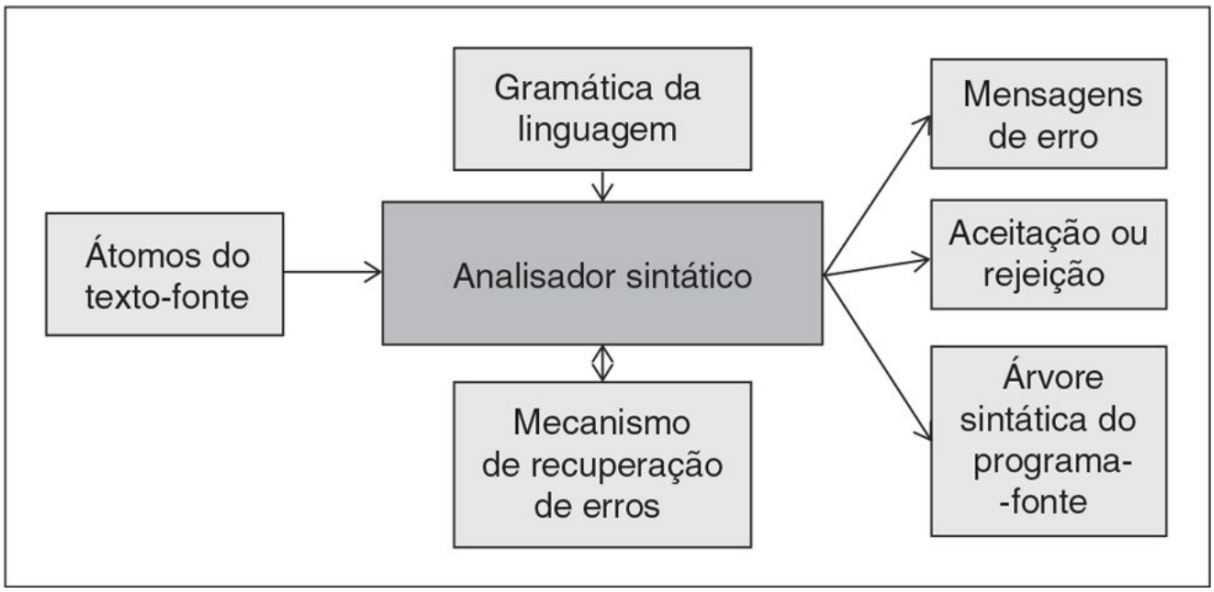
2.3 ANÁLISE SINTÁTICA

Após o programa fonte ser devidamente processado pelo analisador léxico, o analisador sintático obtém a sequência de *tokens* gerada na análise léxica e assegura que a sequência seja válida de acordo com a especificação sintática da linguagem fonte.

Para Neto (2016) a função do analisador sintático é construir uma árvore sintática a partir da sequência de *tokens* dada pela análise léxica, associando os ramos que a compõem aos elementos gramaticais correspondentes da linguagem fonte.

Durante o processo de verificação dos *tokens* também é feita a detecção de erros sintáticos encontrados no texto fonte. Os erros devem ser relatados de forma inteligível ao programador e o analisador sintático deve ser capaz de recuperar os erros e continuar o processo de verificação.

FIGURA 14 - Análise sintática



FONTE: NETO (2016)

A figura 14 apresenta o processo do analisador sintático. A entrada do analisador sintático é a sequência de *tokens* do texto fonte e a gramática da linguagem; junto ao analisador sintático existe um mecanismo de recuperação de erros; e a saída do analisador são as mensagens de erro, a aceitação ou rejeição de cada *token* e a árvore sintática do programa fonte.

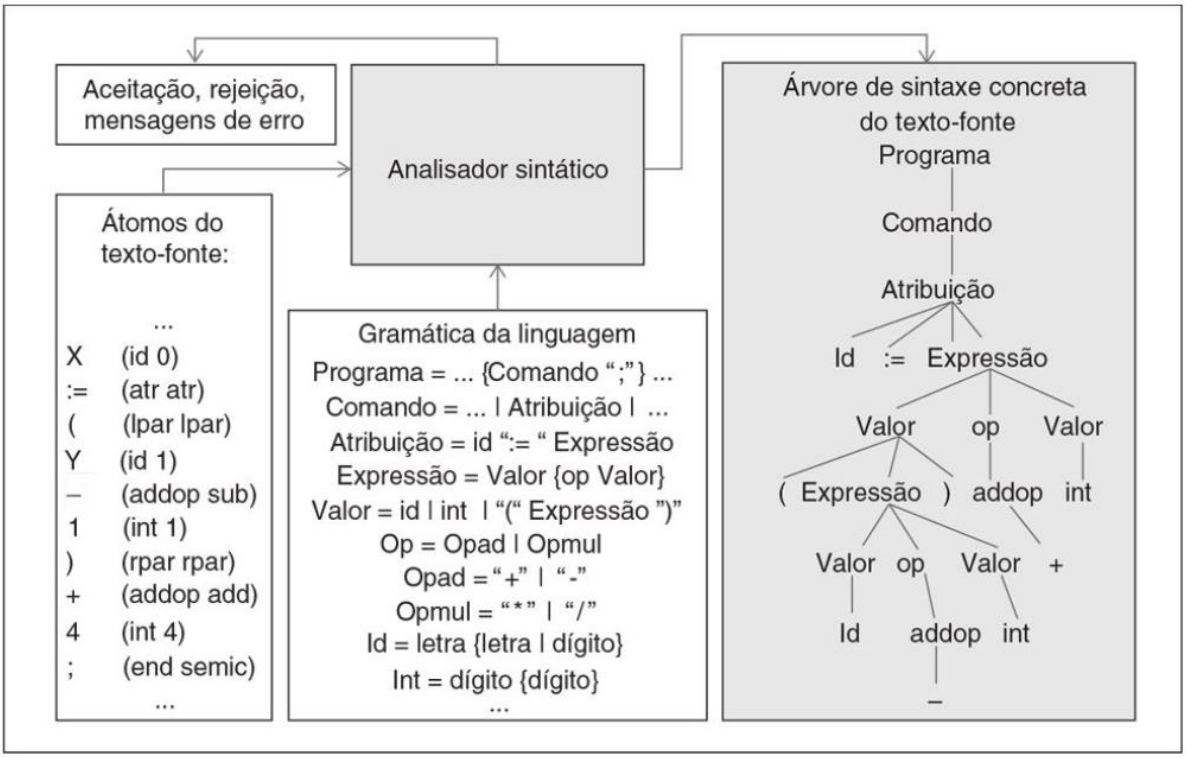
2.3.1 Árvore sintática

A árvore sintática descreve a estrutura do texto fonte de acordo com a gramática da linguagem correspondente ao programa fonte.

A construção da árvore sintática pode ser realizada de diversas formas conforme a necessidade do compilador. Usualmente os compiladores constroem uma árvore sintática que exibe apenas as informações essenciais à tradução do programa fonte.

De acordo com Neto (2016) uma árvore sintática montada com base no texto fonte e na gramática da linguagem é denominada *árvore sintática concreta* e ela reflete fielmente a estrutura e a exata composição do texto fonte do programa.

FIGURA 15 - Árvore sintática concreta

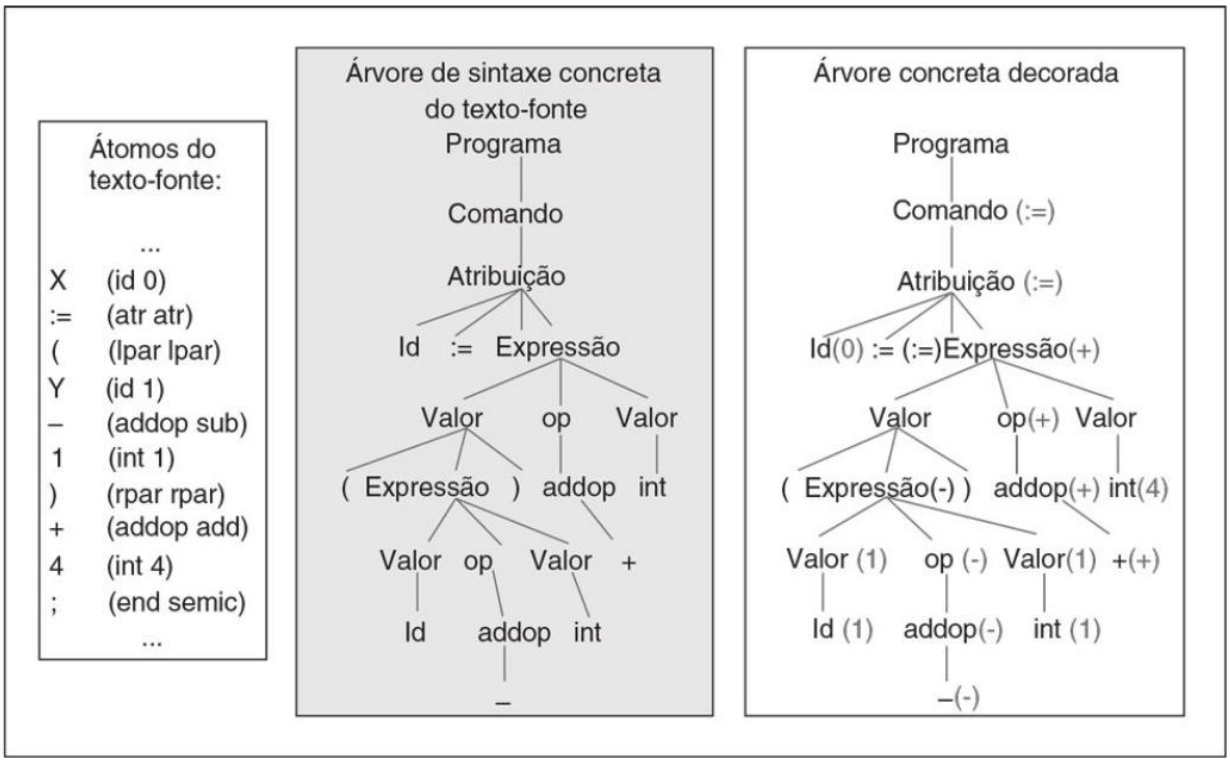


FONTE: NETO (2016)

A figura 15 apresenta um exemplo de processo do analisador sintático. A entrada do analisador sintático é a sequência de *tokens* do texto fonte: identificador 'X', atribuição ':=', abre parênteses '(', identificador 'Y', subtração '-', número inteiro '1', fecha parênteses ')', adição '+', número inteiro '4', ponto e vírgula ';' e a gramática da linguagem; e a saída do analisador são as mensagens de erro, a aceitação ou rejeição de cada *token* e a *árvore sintática concreta* gerada através da gramática e do texto fonte recebidos na entrada do analisador.

Quando a *árvore sintática concreta* é acrescida de informação de cunho semântico ela traz consigo toda a informação disponível sobre o programa que representa e passa a ser chamada de *árvore sintática concreta decorada*.

FIGURA 16 - Árvore concreta decorada

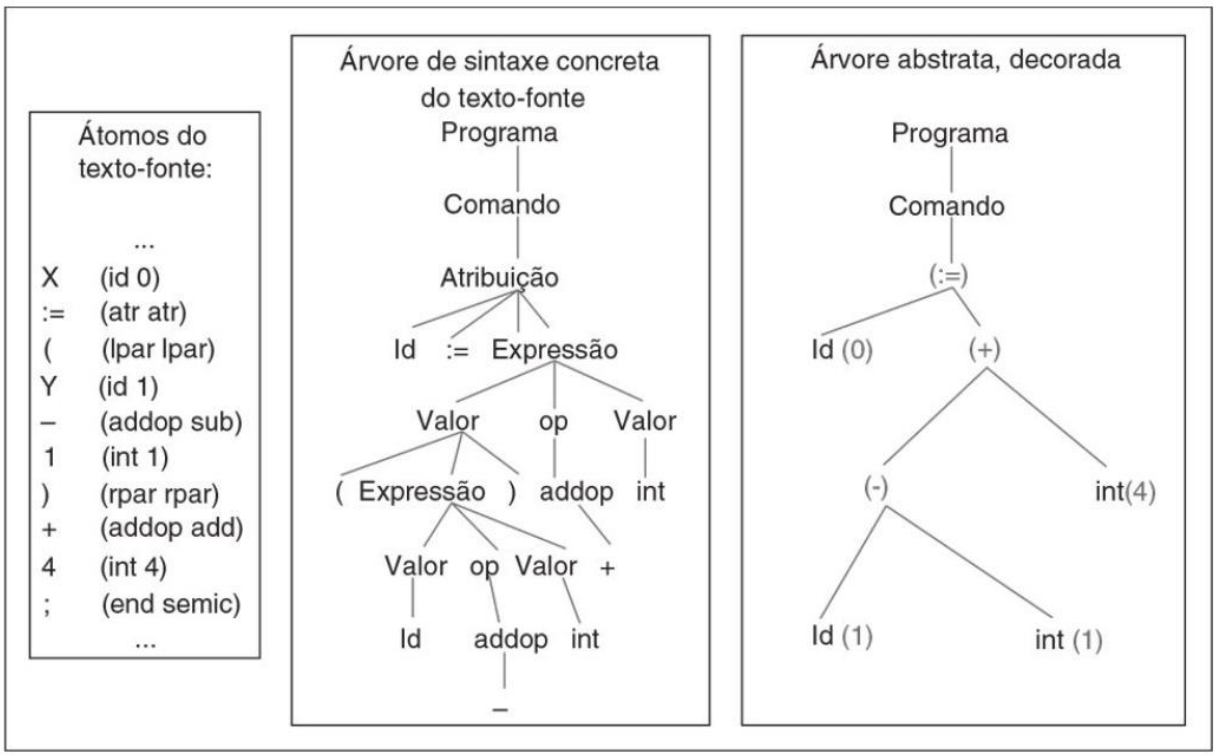


FONTE: NETO (2016)

A figura 16 apresenta um exemplo de construção de uma *árvore sintática concreta* e de uma *árvore sintática concreta decorada*. As duas árvores são construídas a partir da sequência de *tokens* do texto fonte e pode-se observar que a diferença entre elas é que a árvore decorada traz consigo o lexema de cada *token*.

Apesar de a *árvore sintática concreta* apresentar informação abundante sobre o programa fonte ela não é tão útil ao compilador quanto faz parecer. A *árvore sintática concreta* traz consigo inúmeros elementos sintáticos irrelevantes para o compilador, cujo propósito é aumentar o conforto do programador. Pensando em reduzir o espaço de memória e o tempo de processamento a ser executado pelo compilador para produzir a árvore sintática, existe a *árvore sintática abstrata*, uma versão de árvore sintática que elimina todo elemento sintático irrelevante e traz consigo apenas a informação essencial para a construção do código objeto.

FIGURA 17 - Árvore abstrata



FONTE: NETO (2016)

A figura 17 apresenta um exemplo de construção de uma *árvore sintática concreta* e de uma *árvore sintática abstrata decorada*. As duas árvores são construídas a partir da sequência de *tokens* do texto fonte e pode-se observar que a maior diferença entre elas é que a árvore abstrata por conter apenas os nós correspondentes à expressão, é substancialmente menor e traz consigo poucos dados.

2.3.2 Exemplo prático

Para exemplificar a análise sintática foi criado um código na linguagem D+ com diversos erros sintáticos, com o intuito de apontar qual deveria ser o resultado do compilador quanto a eles.

FIGURA 18 - Código em D+ com erros sintáticos

```
1   /* Código na linguagem D+ para efetuar cálculo de fatorial de um número */
2   MAIN
3
4       VAR INT fatorial, numero
5       numero = 3
6       fatorial = 1
7       WHILE int > 1
8           fatorial = fatorial * numero
9           numero =   1 -
10      LOOP
11
12  END
```

FONTE: a própria autora

A análise sintática do código apresentado na figura 18 deve disparar os seguintes erros: erro de declaração de *main* na linha 2 do código, por falta dos parenteses; erro nas linhas 4, 5, 6, 8 e 9 por falta de ponto e vírgula; erro de comando *while* na linha 7 por falta do comando *do;* e erro na linha 9 pois a operação está incompleta.

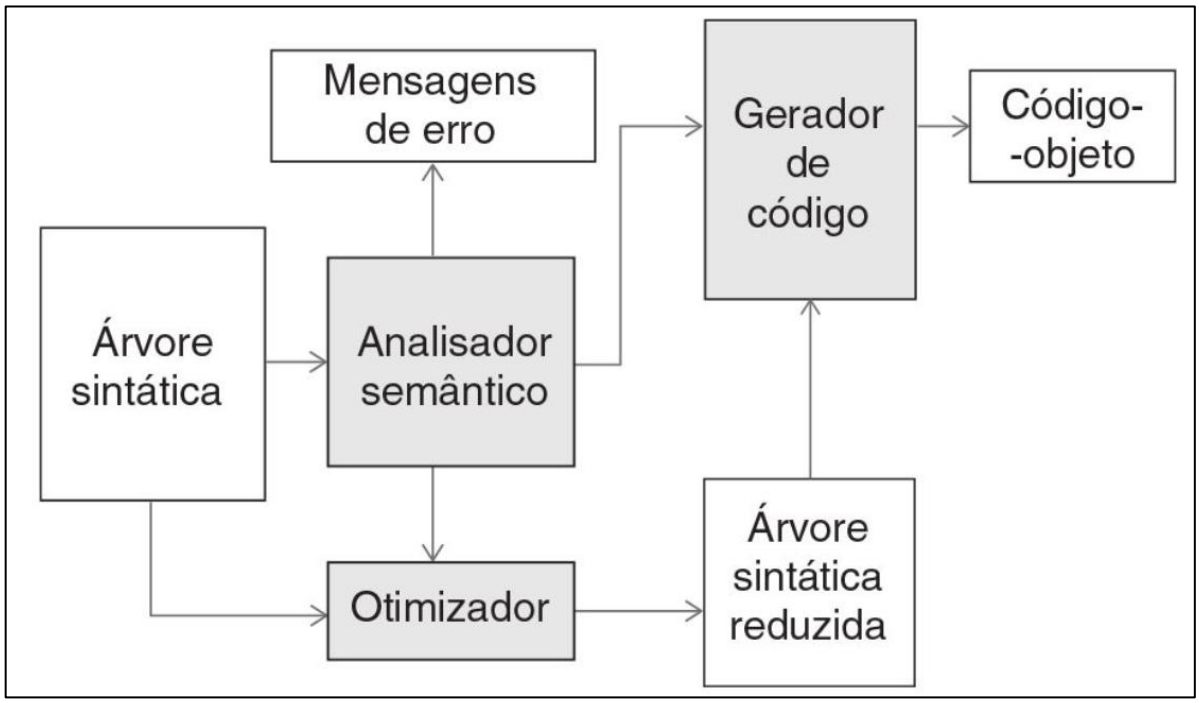
2.4 ANÁLISE SEMÂNTICA

Na terceira fase do compilador é realizada a análise da árvore sintática abstrata gerada na fase anterior, em busca de possíveis ineficiências que possam ser removidas sem comprometer o programa.

Essa atividade, de acordo com Neto (2016) é denominada análise semântica e nela podem ser levantadas informações que permitam ao compilador decidir sobre as alterações que podem ou devem ser feitas na árvore.

Para Aho (1995) a análise semântica verifica os erros semânticos no programa fonte e captura as informações de tipo utilizando a estrutura hierárquica determinada pelo analisador sintático, a fim de identificar os operadores e operandos das expressões e enunciados para a fase de geração de código.

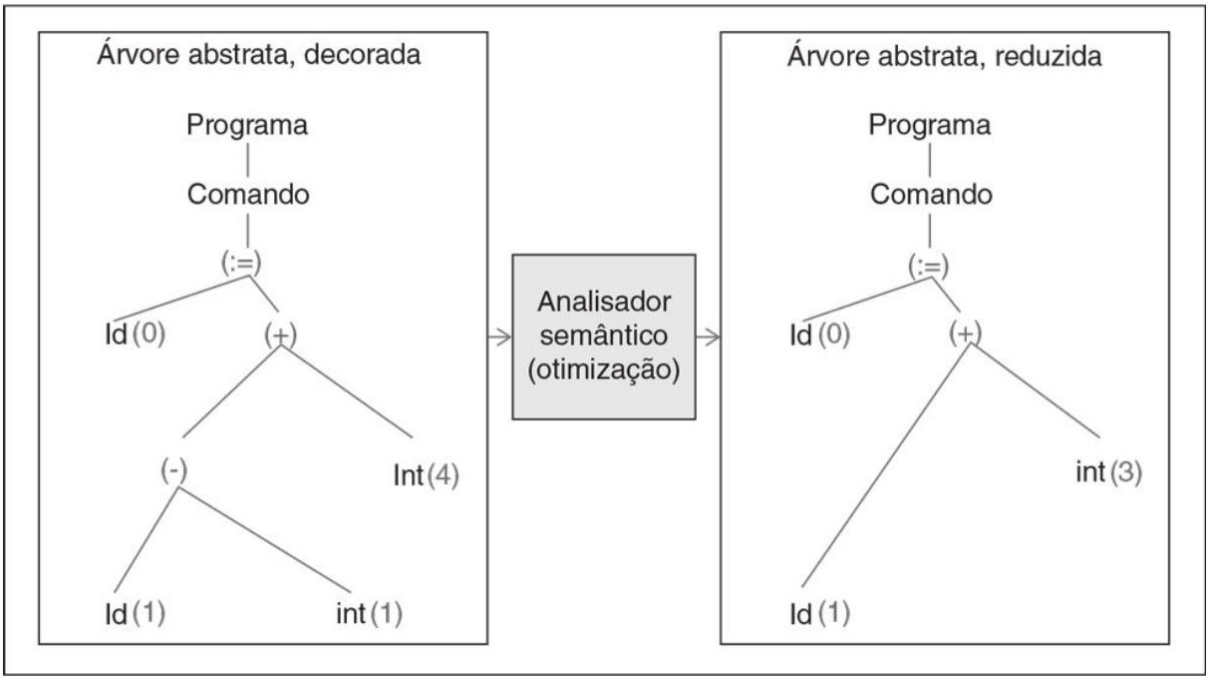
FIGURA 19 - Análise semântica



FONTE: NETO (2016)

A figura 19 apresenta o processo do analisador semântico. O analisador semântico pode realizar dois caminhos diferentes para a geração do código-objeto: o melhor caminho é o compilador enviar a árvore sintática e a análise semântica realizada para um otimizador que remove redundâncias e ineficiências do programa e gera a *árvore sintática abstrata reduzida* e a partir dessa nova árvore enviá-la para o gerador de código, dessa forma, o código-objeto terá menos funções e portanto será mais eficiente; o outro caminho é o resultado do analisador semântico ir direto para o gerador de código e resultar no código-objeto, não otimizado. Caso haja erros nessas etapas, a saída do analisador serão as mensagens de erro.

FIGURA 20 - Árvore sintática abstrata reduzida



FONTE: NETO (2016)

A figura 20 apresenta o analisador semântico recebendo de entrada uma árvore abstrata decorada e retornando uma árvore abstrata reduzida equivalente.

2.5 TABELA DE SÍMBOLOS

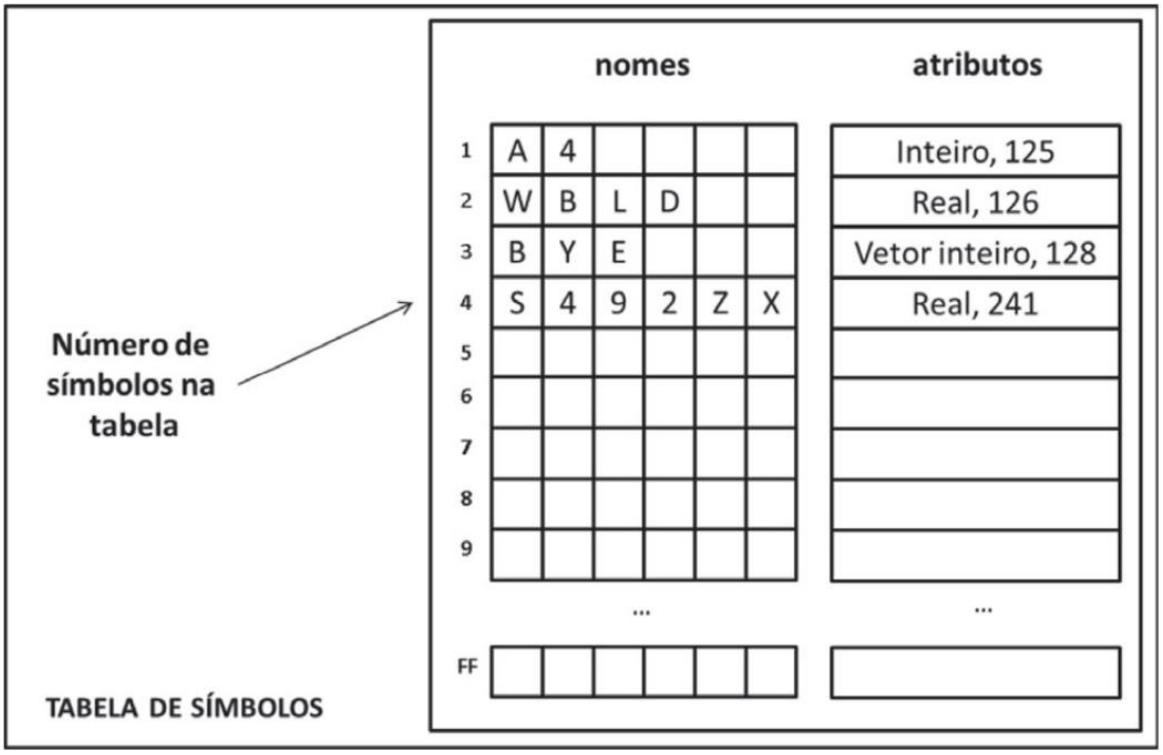
A criação e manutenção de tabelas de símbolos usualmente é executada pelo analisador léxico. Porém, de acordo com Neto (2016) alguns compiladores constroem os analisadores léxicos e sintáticos limitados estritamente às suas operações mínimas, com o intuito de poder reaproveitá-los. Portanto, as atividades que não podem ser reutilizadas são transferidas para outra parte do compilador.

Neto (2016) define a tabela de símbolos como uma coleção dos identificadores encontrados pelo compilador ao longo da inspeção do texto fonte, relacionados aos atributos necessários para a geração de código.

Para Aho (1995) o compilador utiliza a tabela de símbolos para controlar as informações de escopo e para manter atualizadas as informações acerca de um novo

nome ou de uma nova informação. Santos (2018) relaciona a classe do identificador, o tipo de dados e o tipo de retorno como informações essenciais a serem memorizadas pela tabela de símbolos.

FIGURA 21 - Tabela de símbolos



FONTE: NETO (2016)

A figura 21 apresenta um exemplo de uma tabela de símbolos. Ao lado esquerdo da tabela são colocados os nomes dos identificadores encontrados no texto fonte e ao lado direito são colocados seus respectivos atributos.

Como uma tarefa de controle do compilador, a tabela de símbolos é encarregada de consultar símbolos para o compilador a qualquer momento. De acordo com Neto (2016), caso o símbolo conste na tabela é feita uma consulta em seus atributos com a finalidade de coletar dados; e se o símbolo não consta na tabela, geralmente é uma situação de erro.

Sendo assim, a criação e manutenção da tabela de símbolos são operações fundamentais para o correto funcionamento do compilador.

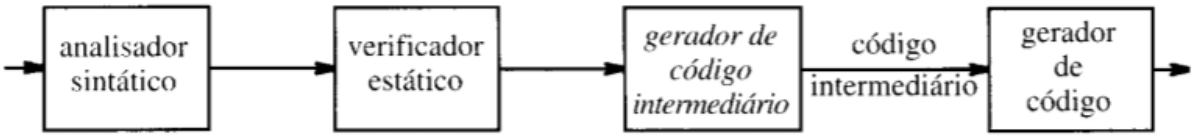
2.6 GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Em alguns modelos de compiladores a penúltima fase do compilador é a geração de código intermediário.

De acordo com Aho (1995) no modelo de análise e síntese de um compilador, o programa fonte é traduzido numa representação intermediária.

Santos (2018) afirma que o código intermediário consiste em um processo de linearização de código que gera uma sequência linear de instruções intermediárias a partir de uma árvore sintática ou qualquer outro tipo de representação intermediária do programa a ser compilado.

FIGURA 22 - Código intermediário



FONTE: AHO, SETHI, ULLMAN (1995)

A figura 22 apresenta em qual local no fluxo de processos do compilador é construído e utilizado o código intermediário.

De acordo com Santos (2018) o principal propósito de um compilador é gerar um código para algum processador e para isso é necessário conhecer os tipos de processadores, ver todos os tipos de representações intermediárias possíveis e definir uma arquitetura que permita gerar, de forma simples e direta, o código não otimizado.

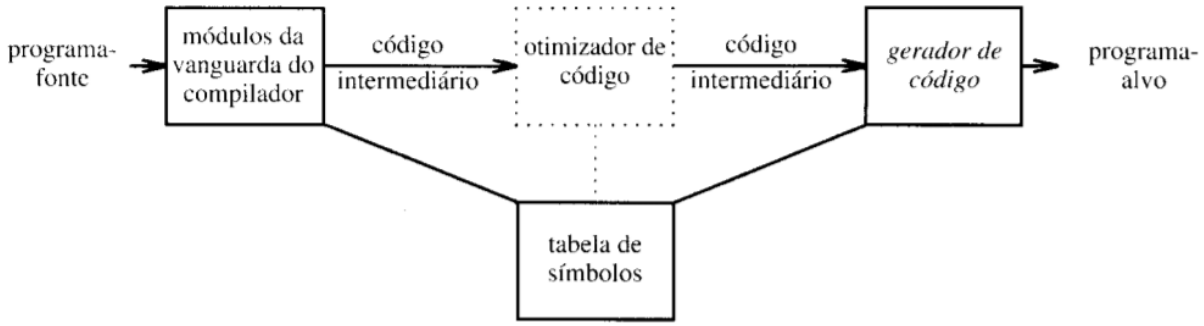
Vide este problema de otimização, para a criação de um código otimizado os compiladores utilizam o código intermediário que pode ser manipulado de modo a gerar instruções mais eficientes para um processador específico.

Aho (1995) afirma que uma vantagem da criação de código intermediário é que outros computadores podem reutilizá-lo, precisando executar o processo de compilação a partir da geração de código.

2.7 GERAÇÃO DE CÓDIGO

A fase final do compilador é a tradução propriamente dita do programa fonte para a forma do código objeto, conhecida como geração de código. A entrada do gerador de código é a representação intermediária do programa fonte e a saída é o programa alvo, como indicado na figura 23.

FIGURA 23 – Gerador de código



FONTE: AHO, SETHI, ULLMAN (1995)

A figura 23 apresenta o fluxo completo do compilador dando enfoque à etapa de geração de código. Inicialmente o compilador recebe o programa fonte e o processa até a geração do código intermediário; em seguida o código intermediário pode passar por um otimizador de código; a última etapa é o gerador de código que recebe o código intermediário e a tabela de símbolos como entrada e retorna o programa alvo.

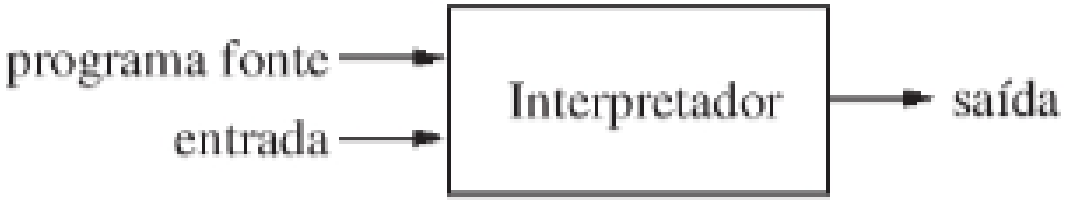
Neto (2016) aponta duas maneiras pelas quais a geração de código pode ser realizada: na primeira maneira, o processo de geração de código é realizado com uma interpretação literal do programa fonte gerando um código objeto de baixa qualidade; e na segunda maneira o código é formatado para que possa ser tratado por uma seção de otimização disponível no compilador.

2.8 INTERPRETADOR X COMPILADOR

O interpretador é outro tipo de processador de linguagem. De acordo com Aho (2008), diferentemente do compilador o interpretador executa diretamente as operações

especificadas no programa fonte sobre as entradas fornecidas pelo usuário, como apresentado na figura 24.

FIGURA 24 - Interpretador



FONTE: AHO, LAM, SETHI, ULLMAN (2008)

Como pode ser visto na figura 24 o interpretador recebe o programa fonte, que é o programa escrito pelo programador e a entrada fornecida pelo usuário; em seguida são executadas as operações do programa fonte sobre a entrada e é retornado o programa objeto como saída.

Aho (2008) argumenta que um compilador normalmente produz um programa objeto muito mais rápido no mapeamento de entradas para saídas que um interpretador. Porém, Aho (2008) aponta que um interpretador, ao executar o programa fonte instrução por instrução, é capaz de oferecer um diagnóstico de erro melhor que um compilador.

3 **INTERAÇÃO HUMANO-COMPUTADOR**

Este capítulo aborda alguns princípios da interação humano-computador que, quando considerados no desenvolvimento de um sistema interativo, aumentam a qualidade do sistema.

De acordo com Benyon (2011) o *design* de sistemas interativos é focado na entrega de um sistema de alta qualidade, onde os produtos e serviços oferecidos pelo sistema combinam com as pessoas e com seus modos de vida.

Nas seções 3.1, 3.2, 3.3 são apresentados os conceitos de acessibilidade, usabilidade e aceitabilidade respectivamente; na seção 3.4 são apresentados importantes considerações para o desenvolvimento de uma interface na *web.*

3.1 ACESSIBILIDADE

De acordo com Sobral (2019) a acessibilidade em uma interface tem a função de diminuir as barreiras de interação entre o usuário e o sistema estabelecendo qualidade de interação para a realização de tarefas.

Devido ao aumento crescente de usuários de computadores e tecnologias é cada vez mais comum a necessidade de softwares acessíveis. Para esse requisito Benyon (2011) aponta duas abordagens do *design* que visam a acessibilidade: o *design* universal e o *design* inclusivo.

O *design* universal é um conjunto de conceitos de acessibilidade que um *designer* precisa ter ao desenvolver um produto para torná-lo acessível a qualquer usuário. Os princípios do *design* universal são baseados na abordagem filosófica do *The Center for Universal Design* – numa tradução livre “O Centro do *Design* Universal” apresentados no quadro 3.

QUADRO 3 - Princípios do Design Universal

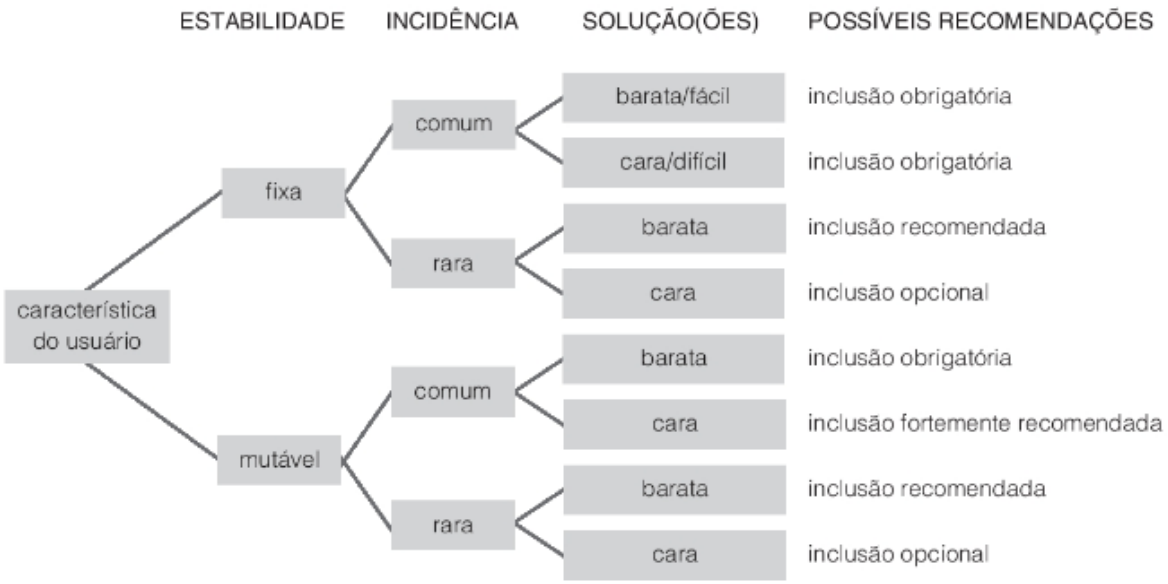
Princípios	Descrição
Uso equitativo	O *design* não prejudica ou estigmatiza nenhum usuário
Flexibilidade no uso	O *design* acomoda uma variedade de preferências e habilidades individuais
Uso intuitivo	O uso do *design* é fácil de entender independentemente do conhecimento do usuário
Informação perceptível	O *design* transite a informação corretamente ao usuário, independentemente do ambiente ou das habilidades sensoriais do usuário
Tolerância ao erro	O *design* minimiza consequências de ações acidentais
Baixo esforço físico	O *design* não necessita de muito esforço físico
Tamanho e espaço para acesso e uso	O espaço fornecido para aproximação, uso, alcance e manipulação deve ser apropriado independentemente do tamanho do corpo do usuário, postura ou mobilidade

FONTE: The Center for Universal Design adaptado

De acordo com Benyon (2011) o *design* inclusivo é baseado nas seguintes premissas: as diferenças nas habilidades são uma característica comum do ser humano visto que as mudanças físicas e intelectuais ocorrem ao longo da vida; se o *design* é eficaz para pessoas com deficiências será eficaz para todos; a autoestima, identidade e bem-estar das pessoas são profundamente afetados pela capacidade delas de funcionar no ambiente físico, com conforto, independência e controle.

Para Benyon (2011) o *design* inclusivo é uma abordagem que frequentemente por razões técnicas ou financeiras é inatingível. A figura 25 apresenta uma árvore de decisão que demonstra visualmente as condições e probabilidades para alcançar a inclusividade dado um determinado cenário.

FIGURA 25 - Árvore de decisão para análise de inclusividade



FONTE: BENYON (2011)

A figura 25 apresenta uma árvore de decisões para analisar a inclusão necessária no *design* de acordo com as características do usuário com que o software irá trabalhar. Por exemplo, o início da árvore é a característica do usuário, se a característica tiver estabilidade fixa, a incidência de usuários necessitados de inclusão for rara e a solução de software for cara a inclusão de software é opcional.

3.2 USABILIDADE

A usabilidade é a principal busca da interação humano-computador. De acordo com Benyon (2011) um sistema com alto grau de usabilidade é eficiente na questão de esforço que o usuário precisa exercer, contém funções e conteúdo de informações adequadas e organizadas, é fácil de aprender e de lembrar, é seguro e entrega alto grau de utilidade para o seu propósito.

QUADRO 4 - Critérios de Usabilidade

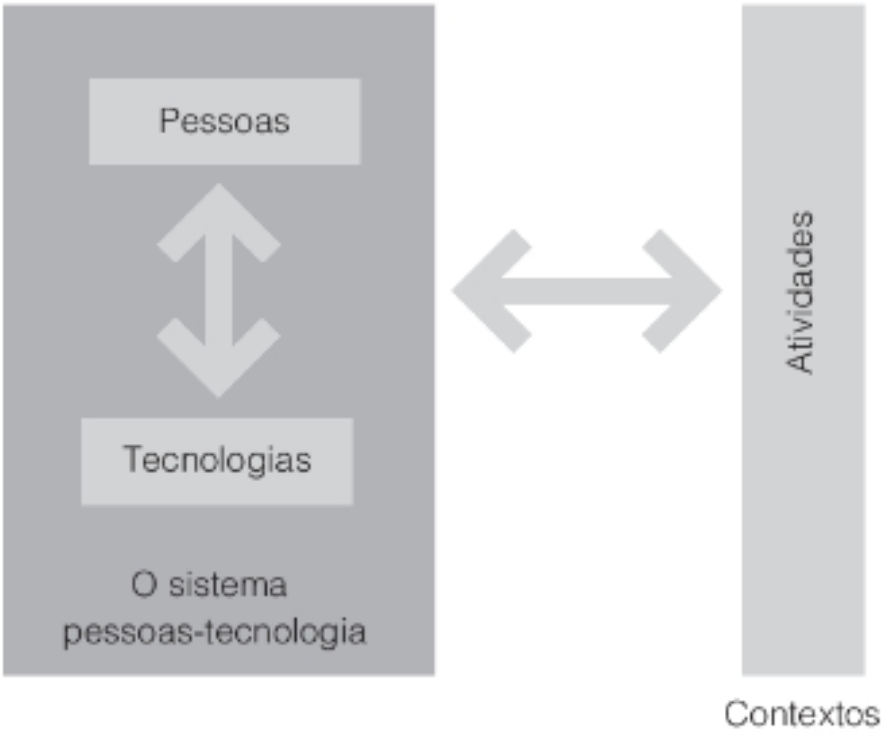
Critério	Descrição
Fácil de aprender	O usuário deve aprender o sistema sem dificuldades
Fácil de lembrar	O sistema deve ser facilmente memorizado de tal forma que um usuário esporádico consiga utilizá-lo após certo período sem a necessidade de reaprendê-lo
Eficiência	Depois que um usuário souber utilizar o sistema a interface deve ser eficiente
Poucos erros	O sistema deve ter baixa taxa de erros durante a interação
Satisfação subjetiva	O sistema deve ser agradável ao uso

FONTE: SOBRAL (2019) adaptado

O quadro 4 apresenta os critérios de um sistema com alto grau de usabilidade. De acordo com Sobral (2019) um sistema seguindo esses critérios fará com que o usuário precise utilizar menos carga de trabalho cognitivo, gerando facilidade de aprendizagem, diminuição dos erros, eficiência na interação e satisfação.

Para Benyon (2011) a usabilidade tem o objetivo de equilibrar os quatro principais fatores do *design* de sistemas interativos centrados no ser humano: pessoas, atividades que as pessoas desejam realizar, contextos nos quais a interação acontece e as tecnologias.

FIGURA 26 - Usabilidade



FONTE: BENYON (2011)

A figura 26 ilustra uma característica importante da interação humano computador. Do lado esquerdo é apresentada a interação entre pessoas e as tecnologias num sistema que as pessoas utilizam; do outro lado é apresentada a relação das pessoas e as atividades que estão sendo realizadas e os contextos dessas atividades.

3.3 INTERFACE NA *WEB*

De acordo com Garret (2000) originalmente a *web* era um espaço de troca de informações hipertextuais. Porém, com o crescente desenvolvimento de tecnologias a *web* começou a ser utilizada como uma interface de software remoto.

O desenvolvimento de uma interface interativa de acordo com (2019) foca em encontrar uma solução que promova a interação da interface com o usuário entre o hardware e o software. Para alcançar uma boa interação é necessário saber quais são as funções do software, quem é o usuário que irá utilizar o software e que tipo de tarefa ele

deseja realizar. Além disso, o desenvolvimento de uma interface interativa deve levar em conta a utilização de usuários de níveis iniciante, intermediário e avançado.

Garret (2000) define considerações-chave que fazem parte do desenvolvimento da experiência do usuário na *web*.

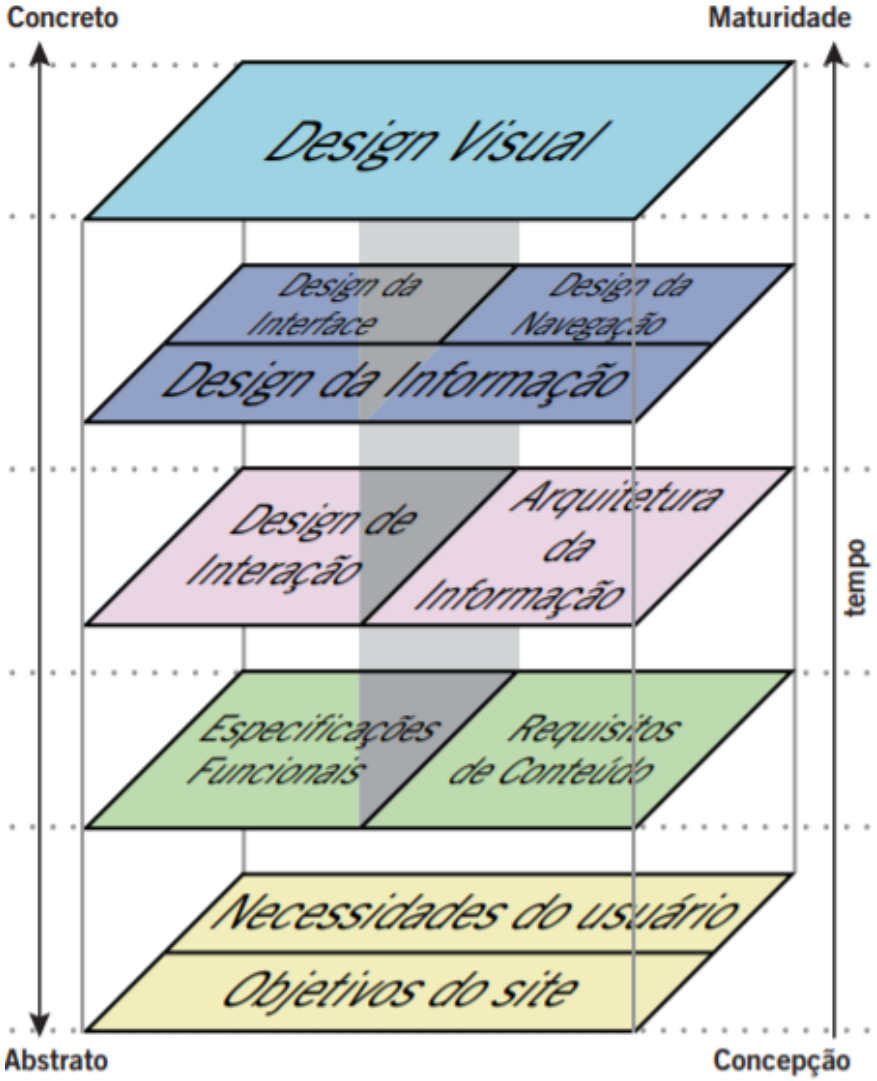
As considerações-chave ambíguas tanto para a *web* como interface de software quanto para a *web* como um sistema de hipertexto são: o *design* visual, responsável pelo tratamento dos elementos gráficos do site; as necessidades do usuário que são identificadas através de pesquisas; e os objetivos do site.

As demais considerações-chave para a *web* como interface de software de acordo com Garret (2000) são: o *design* da interface que trabalha com os elementos da interface para facilitar a interação do usuário com as funcionalidades; o *design* da informação que se preocupa em facilitar a compreensão da informação; o *design* de interação que define como o usuário interage com as funcionalidades do sistema; as especificações funcionais que descrevem detalhadamente as funcionalidades que o site deve incluir para satisfazer as necessidades do usuário.

Para a *web* como um sistema de hipertexto as demais considerações-chave de acordo com Garret (2000) são: o *design* de navegação que facilita a movimentação do usuário em meio a arquitetura da informação; o *design* de informação que se encarrega de apresentar a informação de forma que facilite a compreensão do usuário; a arquitetura da informação que estrutura a informação para facilitar o acesso intuitivo do conteúdo; os requisitos de conteúdo que definem quais são os conteúdos necessários ao site em relação às necessidades do usuário.

A figura 27 apresenta o esquema dos elementos da experiência do usuário definido por Garret (2000).

FIGURA 27 - Elementos da experiência do usuário



FONTE: GARRET (2000)

4 **TRABALHOS RELACIONADOS**

Esse capítulo apresenta em suas seções a síntese dos trabalhos relacionados ao tema do presente projeto. O quadro 5 apresenta um comparativo entre os trabalhos relacionados.

QUADRO 5 - Comparação dos Trabalhos Relacionados

Título	**Possui saída visual?**	**Ilustra a Análise Léxica?**	**Ilustra a Análise Sintática?**	**Ilustra a Tabela de Símbolos?**	**Ilustra código intermediário?**	**Ilustra os erros de compilação?**	**É *off-line* ou *on-line*?**	**Ilustração estática ou dinâmica?**
Auxílio no ensino em compiladores: software simulador como ferramenta de apoio na área de compiladores	SIM	NÃO	NÃO	SIM	SIM	SIM	OFF	ESTÁTICA
C-gen – ambiente educacional para geração de compiladores	SIM	SIM	SIM	NÃO	NÃO	NÃO	OFF	ESTÁTICA
Compilador educativo verto: ambiente para aprendizagem de compiladores	SIM	SIM	SIM	SIM	SIM	NÃO	OFF	ESTÁTICA
Interpretador da linguagem D+	SIM	SIM	SIM	SIM	NÃO	SIM	OFF	ESTÁTICA
SCC: um compilador C como ferramenta de ensino de compiladores	SIM	NÃO	SIM	SIM	NÃO	NÃO	OFF	ESTÁTICA

FONTE: a própria autora

4.1 AUXÍLIO NO ENSINO EM COMPILADORES: SOFTWARE SIMULADOR COMO FERRAMENTA DE APOIO NA ÁREA DE COMPILADORES

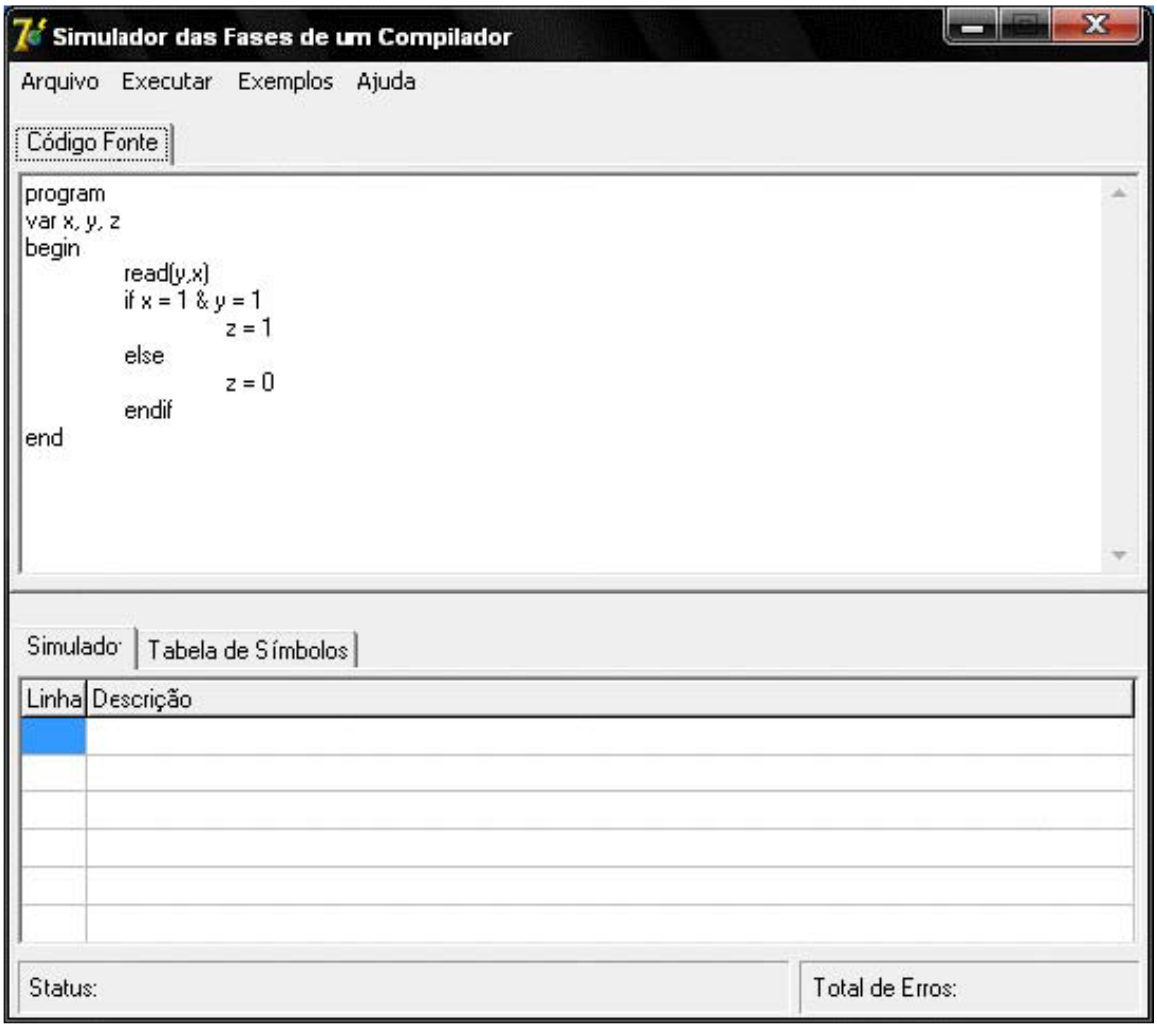
Os pesquisadores Costa, Silva e Britto declararam em seu projeto a necessidade de elaborar uma técnica que pudesse facilitar a compreensão dos discentes, mais especificamente da disciplina de Compiladores, dos cursos que envolvem computação. Com essa premissa, eles projetaram e desenvolveram um software capaz de simular claramente o funcionamento interno das fases de um compilador.

A ferramenta *CompilerSim* foi criada para executar um processo de análise de código baseado na linguagem de programação Pascal, e, posteriormente, para converter

o código para a linguagem de baixo nível Assembly x86. O desenvolvimento da ferramenta se baseou na série de artigos "*Let's Build a Compiler*", escritos pelo cientista Ph.D. Jack W. Crenshaw na década de 80. A partir dos artigos foi extraído o código original e o mesmo passou por modificações e adaptações com o intuito de que as etapas do processo de compilação fossem demonstradas através de uma interface gráfica.

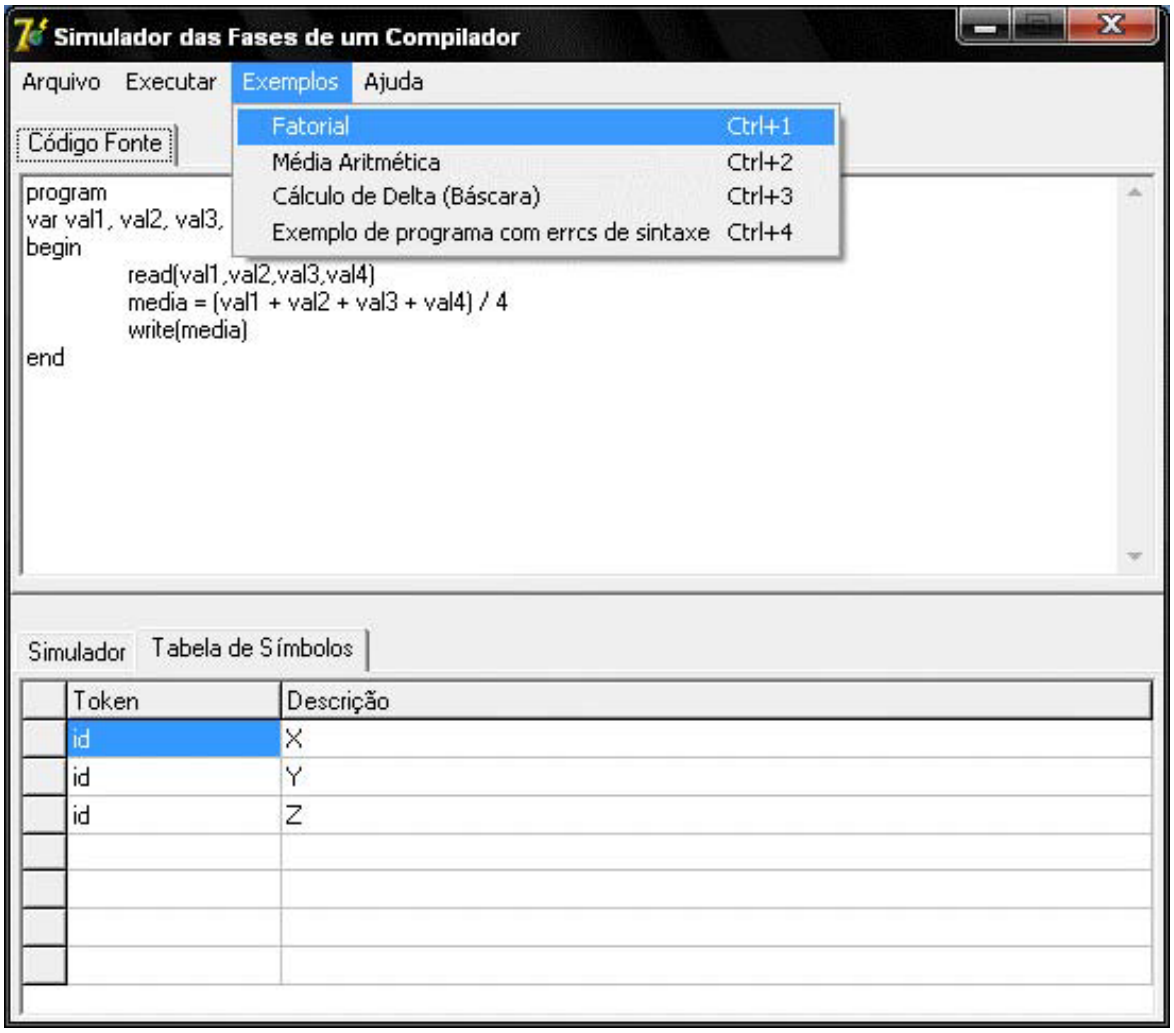
A interface gráfica do software apresenta algumas funcionalidades para o usuário. A primeira etapa da aplicação consiste em inserir o código fonte no programa, podendo ser feito manualmente pelo usuário (figura 28) ou carregado a partir da aba "Exemplos" (figura 29).

FIGURA 28 - Interface padrão do simulador *CompilerSim*



FONTE: COSTA, SILVA, BRITTO (2008)

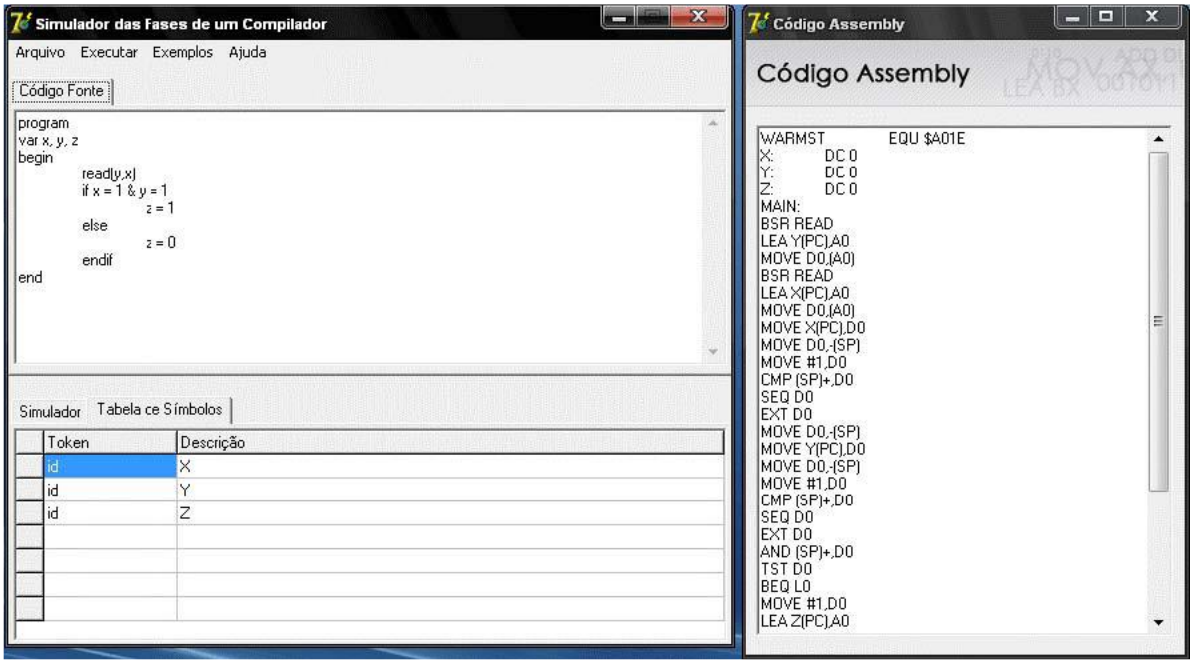
FIGURA 29 - Exemplos prontos para execução



FONTE: COSTA, SILVA, BRITTO (2008)

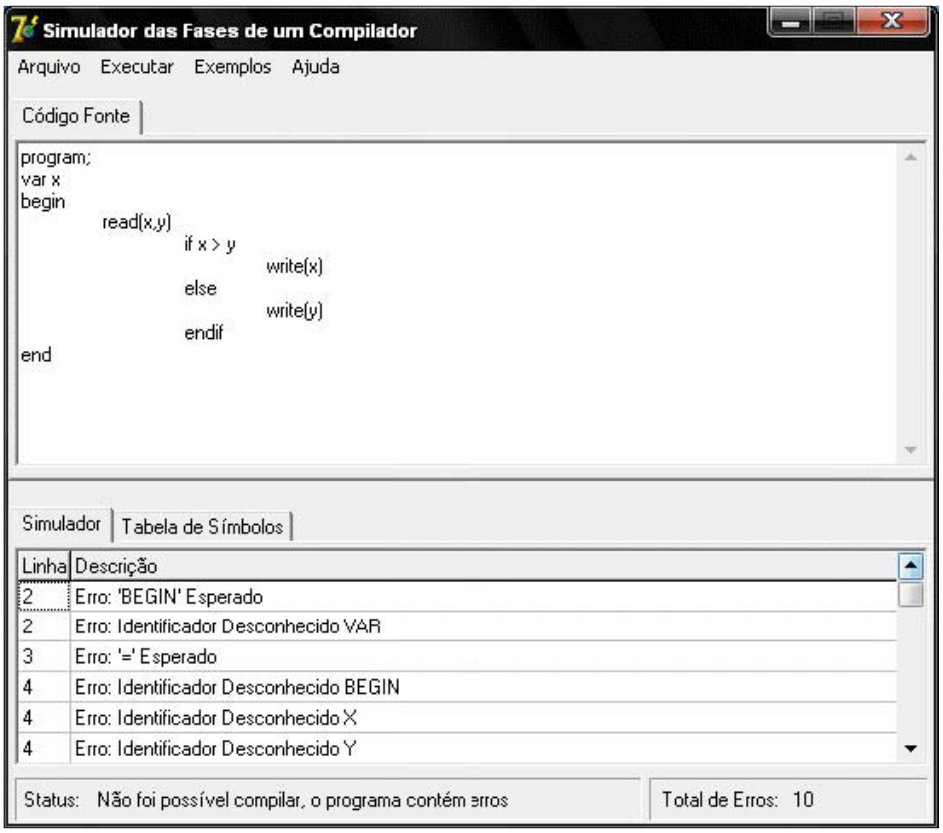
Após a execução do código, a ferramenta apresenta os seguintes resultados: os *tokens* gerados a partir da análise léxica do código, a interpretação e geração de código intermediário na linguagem Assembly, e, quando cabível, os erros gerados na compilação. A visualização desses itens pode ser observada nas figuras 30 e 31 respectivamente.

FIGURA 30 - Análise da linguagem e resultados abordados



FONTE: COSTA, SILVA, BRITTO (2008)

FIGURA 31 - Execução de um código e a indicação de erros



FONTE: COSTA, SILVA, BRITTO (2008)

Os autores do projeto *CompilerSim* concluíram através da utilização do software simulador em salas de aula que a ferramenta pode influenciar positivamente e significativamente o processo de ensino e aprendizagem na disciplina de compiladores.

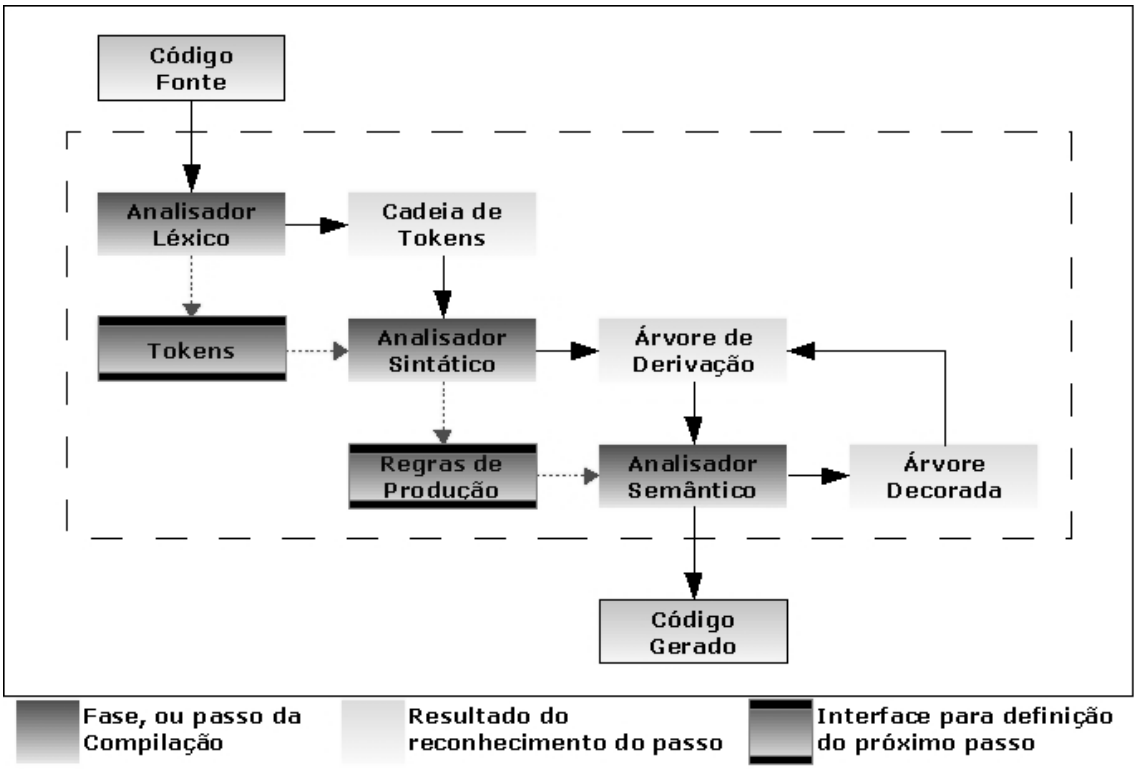
4.2 C-GEN – AMBIENTE EDUCACIONAL PARA GERAÇÃO DE COMPILADORES

O *C-gen* é uma ferramenta, criada pelos pesquisadores Backes e Dahmer, com o intuito de sanar a carência de ferramentas com interface gráfica possíveis de serem utilizadas para orientar os discentes da disciplina de Compiladores, exibindo o funcionamento de todo o processo de compilação.

A ferramenta desenvolvida permite que o discente a utilize enquanto aprende, possibilitando que a teoria de compiladores seja visualizada, assim facilitando a compreensão do conteúdo. Para isso, a ferramenta atende alguns requisitos, dentre eles: possibilitar a definição e disponibilizar uma interface adequada para os analisadores léxicos, sintáticos e semânticos; permitir o acompanhamento do processo de reconhecimento das fases da compilação, passo a passo.

A arquitetura do sistema é organizada para que cada passo do processo de compilação se comunique com o passo posterior através de uma representação intermediária, permitindo ser executado independentemente, conforme apresentado na figura 32.

FIGURA 32 - Arquitetura do protótipo



FONTE: BACKES, DAHMER (2006)

As fases da compilação são implementadas com *plug-ins* que reconhecem as suas respectivas entradas e produzem as saídas correspondentes.

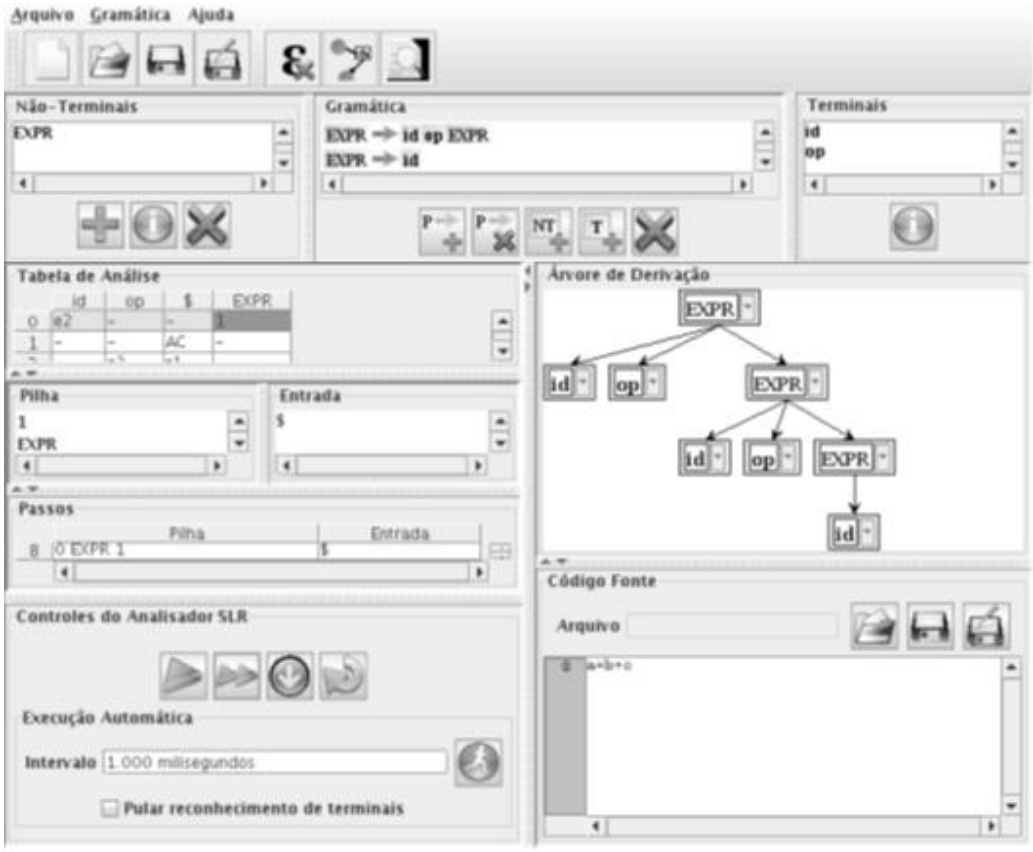
O *plug-in* responsável pelo analisador léxico foi implementado de forma que seja produzido um analisador através da definição de autômatos. A definição é informada pelo usuário, que é responsável pela criação de estados e transições do autômato e os caracteres que o mesmo deve reconhecer em cada transição. Após a edição do autômato, quando iniciado o processo de reconhecimento, a ferramenta converte os dados em um autômato finito determinístico e cria a tabela de transições, conforme ilustra a figura 33.

FIGURA 33 - Interface de edição de um autômato e reconhecimento

FONTE: BACKES, DAHMER (2006)

A análise sintática é feita através da especificação da gramática, considerando como terminais os *tokens* gerados pelo analisador léxico. A gramática é gerada pelo usuário no editor, que deve criar os símbolos não-terminais. O propósito do usuário gerar sua própria gramática é justamente para que ele não precise aprender uma nova notação para utilizar a ferramenta. A interface para essa confecção e de resultado da análise é ilustrada pela figura 34.

FIGURA 34 - Interface de edição de gramáticas e reconhecimento



FONTE: BACKES, DAHMER (2006)

Em suma, além do programa auxiliar os discentes no processo de aprendizagem da disciplina de Compiladores, a estrutura do *C-gen* permite expansões de funcionalidades via *plug-ins* ou através da contribuição voluntária de desenvolvedores, visto que o programa é um software livre.

4.3 COMPILADOR EDUCATIVO VERTO: AMBIENTE PARA APRENDIZAGEM DE COMPILADORES

No Centro Universitário Feevale foi notada a necessidade de desenvolver uma ferramenta de apoio pedagógico, para a disciplina de Compiladores após a mesma enfrentar desmotivação e dificuldades de compreensão dos discentes.

Os compiladores são, de forma geral, tradutores de alguma linguagem para um programa. De acordo com os pesquisadores Schneider, Passerino e Oliveira, a

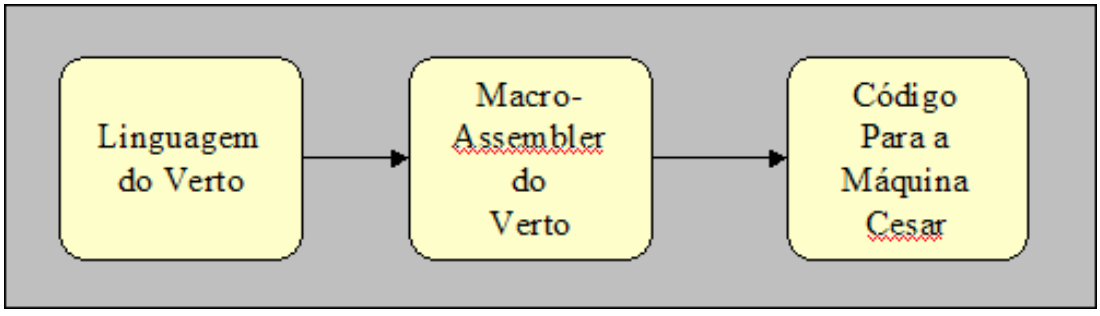
aprendizagem de compiladores consiste em absorver um processo composto por etapas que exigem estratégias e métodos específicos para aprender o processo de compilação, sendo ele: análise léxica, sintática e semântica, tabela de símbolos e geração do código objeto.

A pesquisa ressalta também que, a aprendizagem é a construção de uma representação pessoal de um conteúdo que é objeto de aprendizagem (Coll, 1998 *apud* SCHNEIDER, 2005). Para atingir esse nível de compreensão os estudantes devem ser capacitados a compreender as fases do compilador, sobretudo as fases finais de geração de código intermediário e objeto.

Visto a dificuldade dos discentes e a disciplina ser lecionada em apenas um semestre, foi desenvolvido o Compilador Educativo *Verto*, que faz parte de um ambiente de aprendizagem para compiladores. O compilador é composto por diversas ferramentas computacionais, um docente mediador, os discentes que o utilizam, as sequências didáticas que são planejadas no plano de ensino e aprendizagem apoiada pela metodologia de ensino embasada pelo docente.

O processo de compilação do *Verto* inicia-se com a geração do código intermediário em um formato *macro-assembler*. O formato dispõe de instruções simplificadas para facilitar a compreensão das estruturas compiladas. Após essa etapa, gera-se o arquivo final que contém as instruções no formato *César*, uma linguagem objeto criada com fins didáticos, permitindo que o estudante execute e analise o algoritmo. Na figura 35 pode-se observar o esquema de tradução do compilador *Verto*.

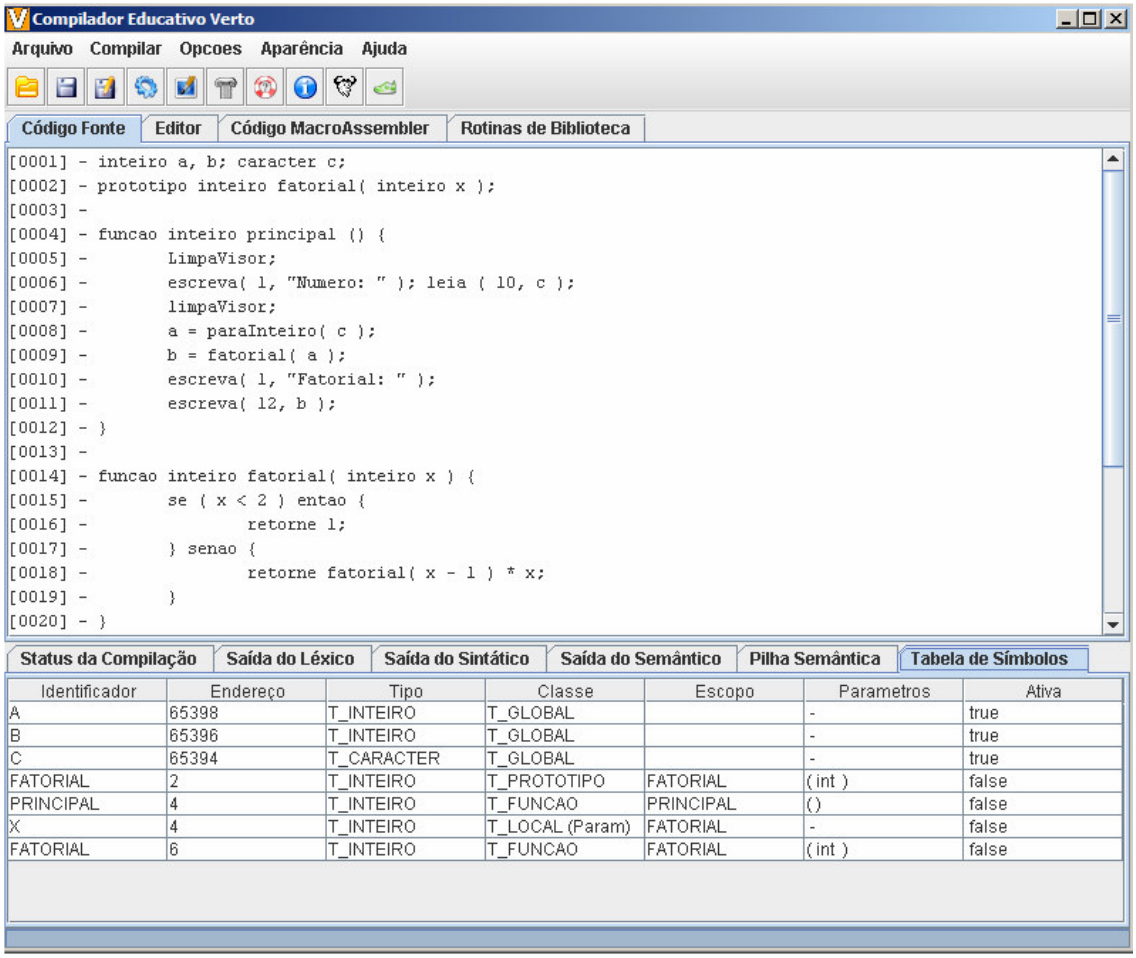
FIGURA 35 - Esquema de Tradução do Compilador *Verto*



FONTE: SCHNEIDER, PASSERINO, OLIVEIRA (2007)

Para o uso inicial da ferramenta, dispõe-se de uma tela para a edição de textos fonte escritos na linguagem *César*. Na figura 36 é possível observar a tela de edição e a tabela de símbolos gerada a partir do código inserido.

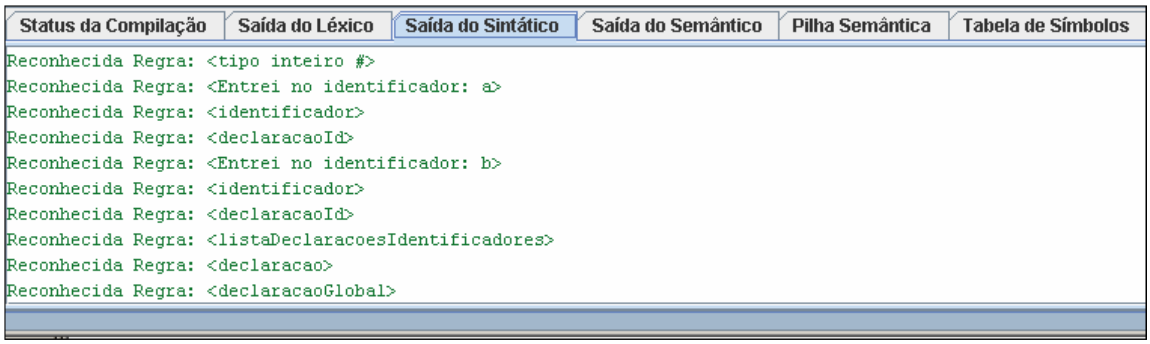
FIGURA 36 - Interface de Edição do Compilador *Verto*



FONTE: SCHNEIDER, PASSERINO, OLIVEIRA (2007)

A ferramenta focou as etapas finais da compilação, utilizando uma técnica de análise léxica simples e um método de análise sintática. A análise sintática, onde é feita a geração de erros, análise semântica e geração do código-objeto, é feita de forma recursiva, onde cada regra sintática reconhecida pode levar o compilador a disparar uma rotina de ação semântica ou de geração de código. A ampliação da saída do analisador sintático é ilustrada pela figura 37, onde é registrada a sequência de regras reconhecidas pelo compilador.

FIGURA 37 - Aba: Saída do Sintático



FONTE: SCHNEIDER, PASSERINO, OLIVEIRA (2007)

Os autores da ferramenta *Verto* a descrevem como uma ferramenta de auxílio de múltiplas disciplinas, sendo elas: compiladores, arquitetura de computadores e paradigmas de linguagens de programação.

4.4 INTERPRETADOR DA LINGUAGEM D+

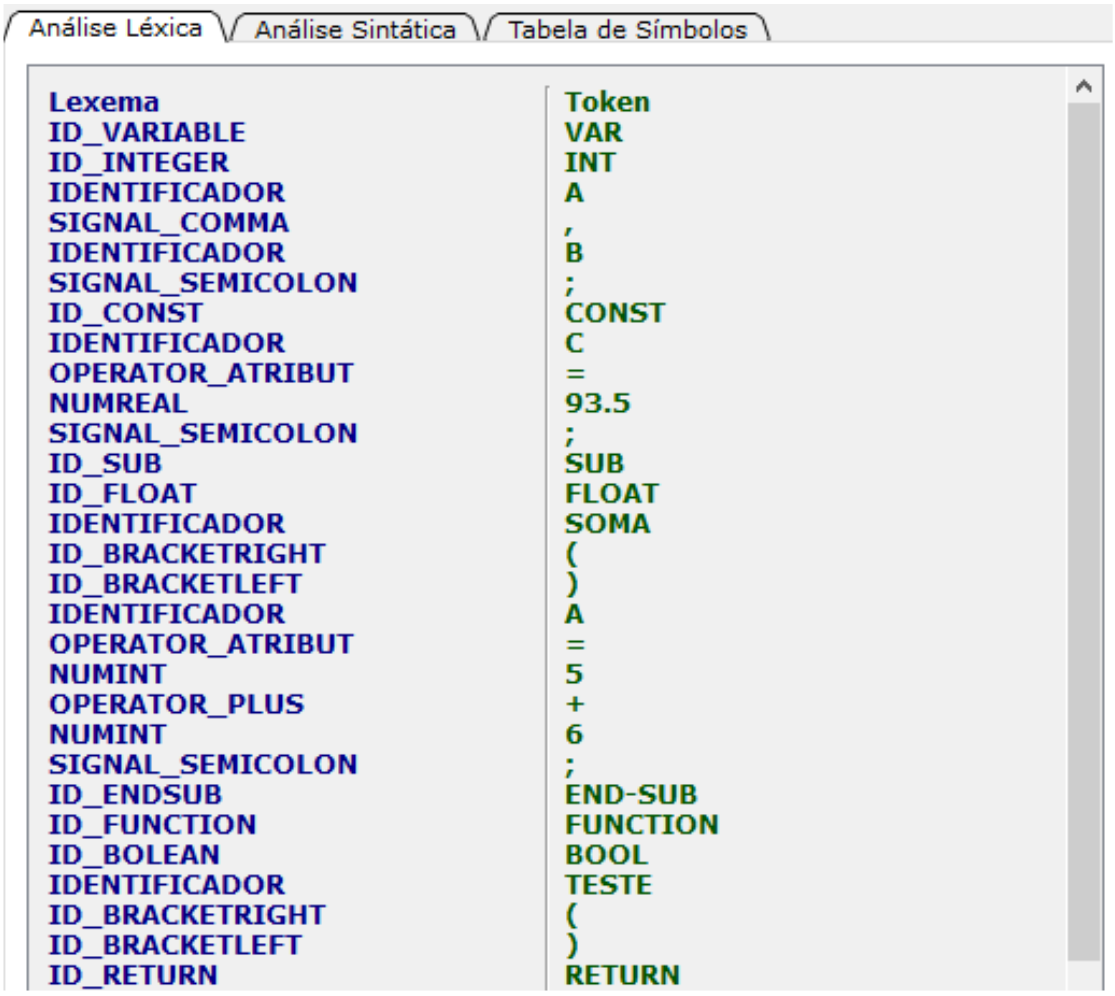
O interpretador da linguagem D+ é um software proposto e desenvolvido por Gabriel Ribeiro (2019). Ribeiro enfatiza em seu projeto a dificuldade de aprendizagem dos discentes da disciplina de Compiladores no Bacharelado em Ciência da Computação, ele descreve a disciplina como muito abrangente e profunda. Em sua pesquisa, o autor indica que mais da metade dos estudantes de compiladores tem grande dificuldade de aprender a disciplina.

Visto o problema da aprendizagem dos estudantes, Ribeiro (2019) desenvolveu um software integrado a uma interface com o intuito de amenizar a dificuldade dos discentes no aprendizado e auxiliá-los na absorção do conteúdo de compiladores. O software consiste em uma interface de fácil utilização, onde o usuário insere o código em linguagem D+, pressiona o botão para executar a compilação, e o resultado já é gerado pelo programa contemplando: análise léxica, análise sintática e tabela de símbolos.

A análise léxica é demonstrada tanto por uma tabela de *tokens* e lexemas, quanto por um autômato finito que contém todos os autômatos possíveis para a linguagem D+.

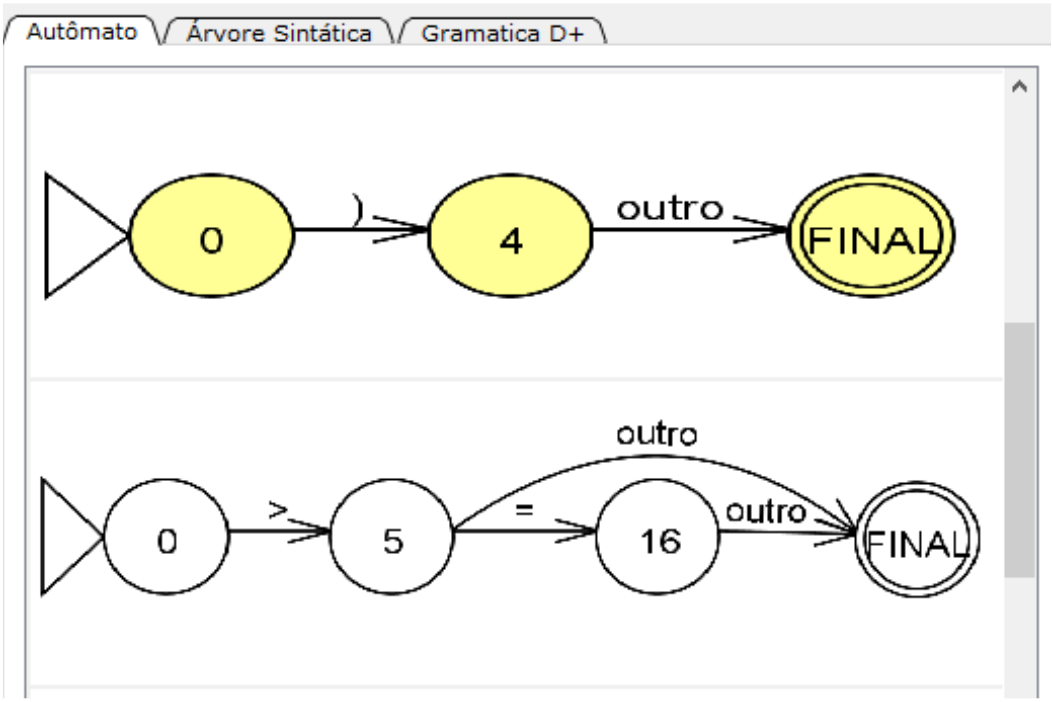
Após a compilação do código inserido pelo usuário, os autômatos utilizados são preenchidos com fundo amarelo. A figura 38 ilustra a tabela de saída da análise léxica e a figura 39 ilustra um autômato colorido quando é acessado, e outro em preto e branco quando não é acessado.

FIGURA 38 - Saída da análise sintática



FONTE: RIBEIRO (2019)

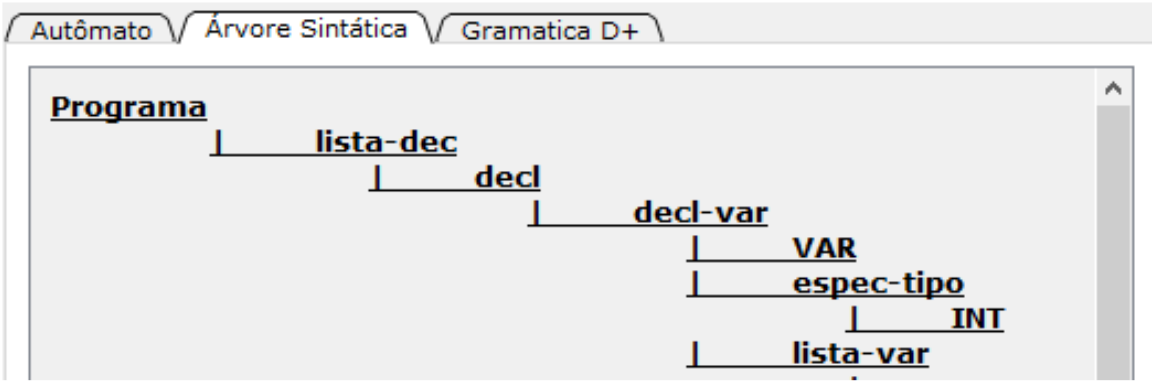
FIGURA 39 - Interface com os autômatos da análise sintática



FONTE: RIBEIRO (2019)

A análise sintática é representada por uma árvore sintática conforme ilustra figura 40.

FIGURA 40 - Árvore Sintática montada



FONTE: RIBEIRO (2019)

A tabela de símbolos ilustra os identificadores localizados pelo interpretador e traz informações que auxiliam na compreensão de cada palavra inserida no código, conforme apresentado na figura 41.

FIGURA 41 - Tabela de símbolos

FONTE: RIBEIRO (2019)

As tecnologias utilizadas para o desenvolvimento do interpretador de D+ foram a linguagem de programação C++ para a codificação do compilador, QT *Creator* para criação da interface e JFLAP para a criação dos autômatos finitos.

Para testes de eficiência, o interpretador de D+ foi distribuído junto a um questionário sobre o uso da ferramenta, para discentes da disciplina de Compiladores, que gerou resultados positivos, afirmando que a ferramenta ajuda na compreensão das etapas do compilador. O projeto também constou que para trabalhos futuros poderiam ser implementadas na interface a: análise semântica, geração do código de máquina, dinamicidade na apresentação da informação e o *log* da análise semântica.

4.5 SCC: UM COMPILADOR C COMO FERRAMENTA DE ENSINO DE COMPILADORES

O compilador SOIS C *Compiler* (SCC), de Foleiss, Assunção, Cruz, Gonçalvez e Feltrim (2009), se destaca entre os compiladores de C por contemplar todo o conjunto de instruções de um compilador, assim proporcionando um serviço de ajuda no ensino de disciplinas que abordam o processo de compilação.

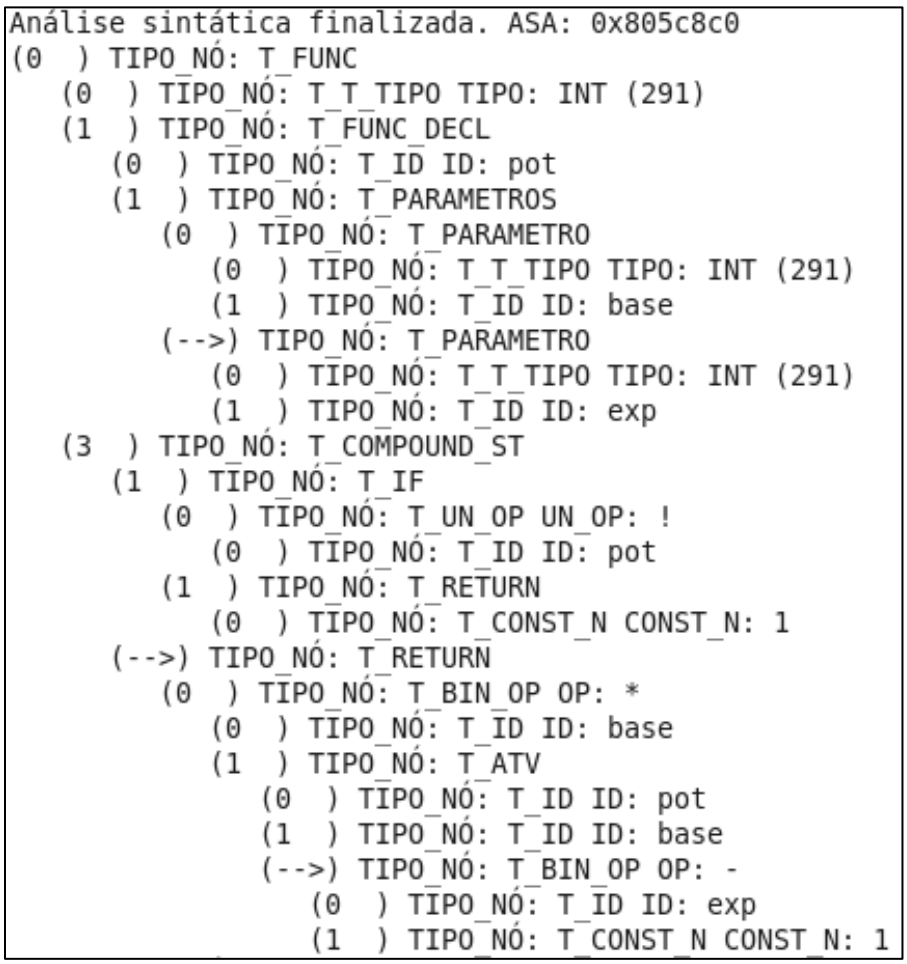
O Sistema Operacional Integrado Simulado (SOIS), foi escolhido justamente por ser um ambiente que permite a escrita, execução e depuração de programas em um ambiente simulado. Por sua vez, o montador SASM possui modos de depuração que podem demonstrar todas as fases do processo de compilação, a inter-relação dos seus artefatos e componentes e os resultados obtidos.

O projeto foi desenvolvido com os objetivos de gerar código que permita que o simulador do ambiente execute programas da linguagem C e de ser uma ferramenta de ensino, tendo funcionalidades que auxiliem a compreensão de áreas específicas do processo de um compilador real.

A pesquisa escolheu o processo de compilação proposto por Louden que possui cinco etapas: análise léxica, análise sintática, análise semântica, geração de código intermediário e geração de código final. No SCC, são geradas: a árvore sintática, a árvore de símbolos, a árvore sintática abstrata anotada e um analisador semântico.

A análise sintática tem como principal artefato a árvore sintática abstrata, nela é mostrada a estrutura sintática do programa, sendo possível visualizar o mapeamento do programa-fonte. A figura 42 ilustra um trecho de uma árvore sintática gerada pelo SCC.

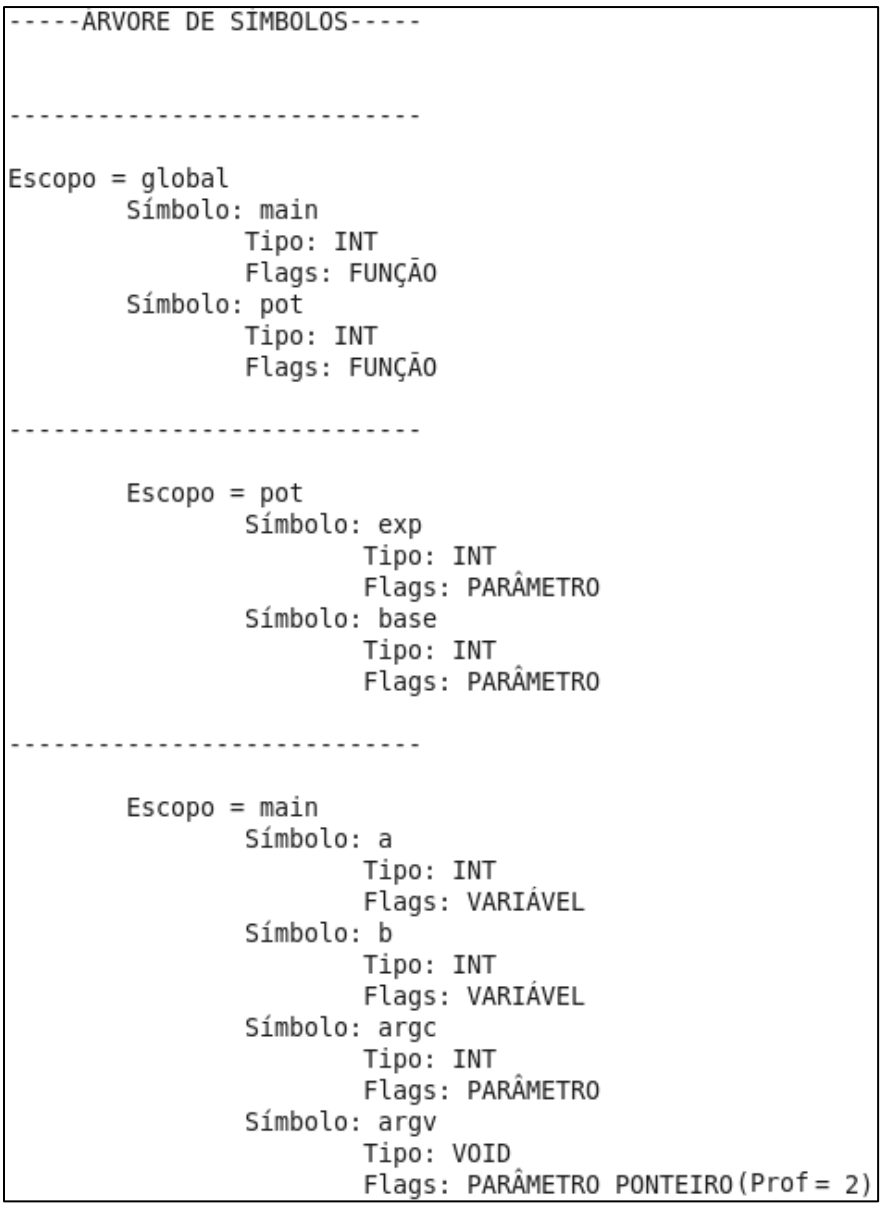
FIGURA 42 - Árvore Sintática



FONTE: FOLEISS, ASSUNÇÃO, CRUZ, GONÇALVEZ, FELTRIM (2009)

A árvore de símbolos é outro artefato que explora o entendimento do compilador, é ela a responsável por transcrever a tabela de símbolos gerada a partir do código fonte. A figura 43 ilustra a árvore de símbolos, que é dividida ente os escopos existentes no código.

FIGURA 43 - Árvore de símbolos



FONTE: FOLEISS, ASSUNÇÃO, CRUZ, GONÇALVEZ, FELTRIM (2009)

A proposta do SCC é apresentar, aos discentes da disciplina de Compiladores, um compilador que mostre os desafios presentes na implementação de um compilador, utilizando métodos empregados no desenvolvimento de compiladores profissionais.

Além dos modos de depuração o SCC provê ao usuário interfaces com algumas partes do processo de compilação. O projeto também citou a possibilidade de incluir um módulo de otimização no sistema e a visualização gráfica da árvore sintática como trabalhos futuros.

5 **METODOLOGIA**

Esse capítulo apresenta a metodologia utilizada e os recursos necessários para o desenvolvimento da ferramenta VL-D+ proposta no presente projeto.

5.1 TECNOLOGIAS UTILIZADAS

Nessa seção são apresentados os softwares e linguagens utilizadas como base para o desenvolvimento do sistema proposto, tais como: ASP.NET Core, Visual Studio, Razor, JavaScript, C#, Canvas e Diagrams.net.

5.1.1 ASP.NET Core

De acordo com a documentação da Microsoft (2020) o ASP.NET Core é uma estrutura de software livre de plataforma cruzada, de alto desempenho que fornece uma estrutura para construção de aplicativos modernos, baseados em nuvem e conectados à internet, como aplicativos *web* e aplicativos *IoT*.

Para o desenvolvimento da ferramenta VL-D+ será utilizado o ASP.NET Core MVC, que de acordo com a documentação da Microsoft (2020) é uma estrutura avançada para a criação de aplicativos *web* e APIs usando o padrão e *design* Modelo-Exibição-Controlador.

O padrão Modelo-Exibição-Controlador (MVC) ajuda a tornar as APIs *web* e os aplicativos testáveis e definem um modelo para a codificação.

A Microsoft (2020) aponta vários benefícios da utilização do ASP.NET Core, tais quais: o *Blazor* permite a utilização das linguagens de programação C# e *JavaScript* para construção da lógica do aplicativo do lado cliente e do servidor; o *Razor Pages* facilita a codificação e a torna mais produtiva em cenários focados em interfaces de páginas *web*; existe um *pipeline* de solicitação HTTP leve, modular e de alto desempenho.

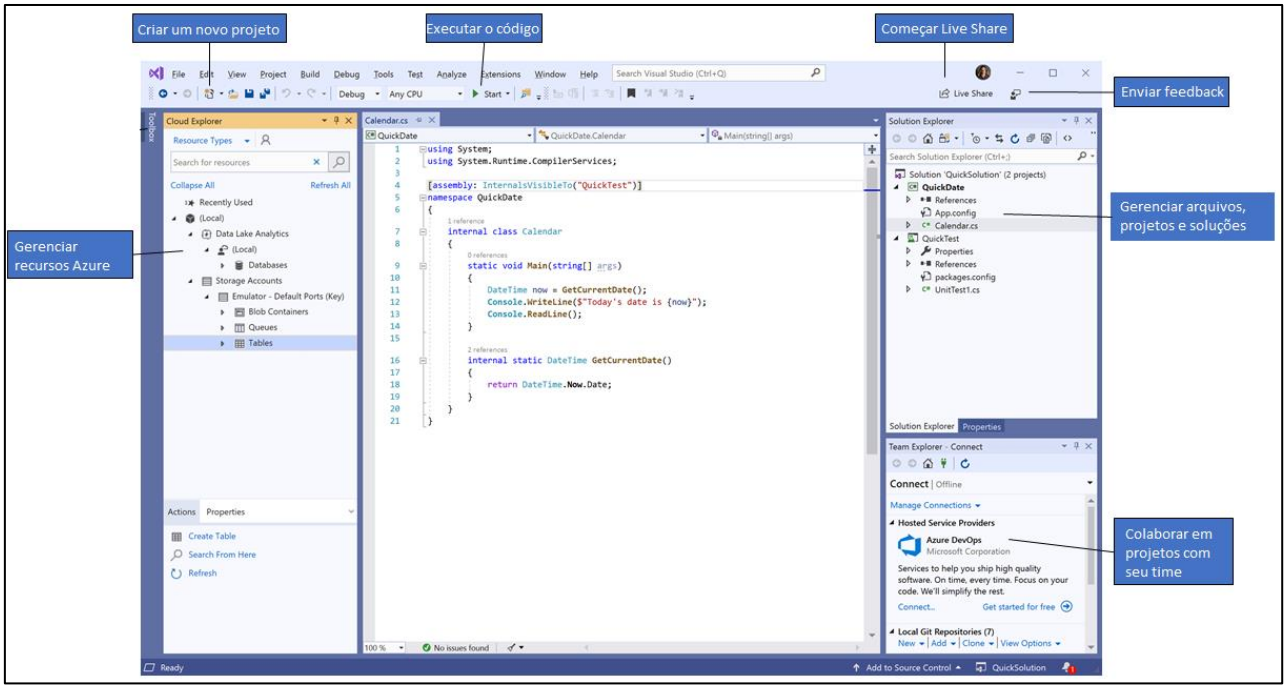
5.1.2 Visual Studio

Para utilização do ASP.NET Core é necessário o ambiente de desenvolvimento integrado (IDE) Visual Studio.

De acordo com a documentação da Microsoft (2019) o Visual Studio é um painel de inicialização criativo que permite ao desenvolvedor editar, depurar e compilar o código e, em seguida publicar um aplicativo.

Além do editor e do depurador já fornecidos em diversas IDE's, a Microsoft (2019) reforça que o Visual Studio inclui compiladores, ferramentas de preenchimento de código, *designers* gráficos e outros recursos que facilitam o processo de desenvolvimento do software.

FIGURA 44 - Janela da IDE do Visual Studio



FONTE: Microsoft (2019) adaptada

A figura 44 apresenta o Visual Studio como um projeto aberto e várias janelas de ferramentas que a IDE dispõe. Dois recursos principais que podem ser vistos na figura são: na parte superior direita se encontra o gerenciador de soluções que permite exibir, navegar e gerenciar os arquivos de código e organiza o código agrupando seus arquivos

em soluções e projetos; e na parte central da imagem aparece a janela do editor onde o desenvolvedor edita o código.

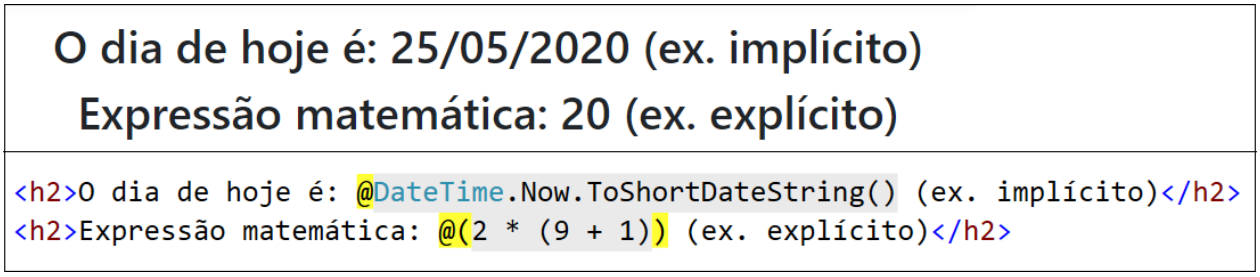
5.1.3 Razor

De acordo com a Microsoft (2020) o *Razor* é uma sintaxe de marcação para inserir código baseado em servidor em páginas da *web*.

A página *Razor* é composta, de acordo com a documentação da Microsoft (2019), por um par de arquivos: um arquivo *cshtml* que contém a marcação HTML com o código C# usando a sintaxe *Razor* e um arquivo *cshtml.cs* que contém o código C# que manipula os eventos de página.

A linguagem padrão do *Razor* é o HTML, porém, é possível escrever códigos na linguagem C# em conjunto do HTML, isto é definido pela Microsoft (2020) como expressões implícitas e explícitas.

FIGURA 45 - Exemplo de expressões em Razor



FONTE: a própria autora

A figura 45 apresenta um exemplo de uso de expressão implícita e explícita. Na metade superior tem-se um exposto retirado de uma página *web* e na metade inferior têm-se o código correspondente escrito em HTML e em C#. A primeira linha do código começa com o código escrito em HTML e a expressão C# é realizada em sequência do '@', a expressão é implícita, pois a expressão C# termina assim que é inserido um separador (quebra de linha, espaçadores ou tabulações); a segunda linha do código começa com o código escrito em HTML, porém se difere da primeira linha de código, pois a expressão C# pode conter separadores enquanto estiver dentro do '@()'.

5.1.4 JavaScript

Para a criação de uma página *web* interativa, dinâmica, responsiva e em tempo real o uso da linguagem de programação *JavaScript* é indispensável para a aplicação.

De acordo com a MDN *web docs* (2019) o *JavaScript* é uma linguagem de *scripting* baseada em protótipos, multi-paradigma e dinâmica, que suporta estilos orientado a objetos, imperativo e funcional.

O *JavaScript* contém diversas funções para manipulação de eventos realizados pelo usuário em alguma parte da página *web*, os mais usados são: o evento *onClick* e o evento *onChange*, geralmente aplicados em botões e em caixas de texto respectivamente.

De acordo com a MDN *web docs* (2019) as capacidades dinâmicas do *JavaScript* incluem a construção de objetos em tempo de execução, listas de parâmetros, variáveis de funções, criação dinâmica de scripts, introspecção de objetos, e recuperação de código fonte. Além disso, é possível obter qualquer valor de um objeto da tela e mudá-lo se necessário.

5.1.5 C Sharp

Além do código C# (C Sharp) que pode ser utilizado nas páginas *Razor*, para as fases da compilação que não foram projetadas para ter uma ilustração dinâmica na interface foi utilizada a linguagem de programação C#.

O C# é uma linguagem de programação criada pela Microsoft como parte da plataforma .NET. A sintaxe do C# é parecida com o Java e C++ e é uma linguagem de programação orientada a objetos.

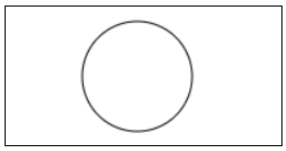
5.1.6 Canvas

Para a criação de uma interface dinâmica e em tempo real capaz de ilustrar o funcionamento do compilador, foi escolhida a Interface de Programação de Aplicações (API) *Canvas*.

De acordo com a MDN *web docs* (2020) o *Canvas* é uma API que provê maneiras de desenhar gráficos via *JavaScript* e via elemento HTML.

O *Canvas* é focado em gráficos 2D e pode ser utilizado para animações, gráficos de jogos, visualizações de dados, manipulações de fotos e processamentos de vídeos em tempo real.

FIGURA 46 - Exemplo do Canvas



```
<canvas id="canvas" width="200" height="100" style="border:1px solid black;"></canvas>

<script>
    var canvas = document.getElementById("canvas");
    var contexto = canvas.getContext("2d");
    contexto.beginPath();
    contexto.arc(95, 50, 40, 0, 2 * Math.PI);
    contexto.stroke();
</script>
```

FONTE: w3schools adaptado

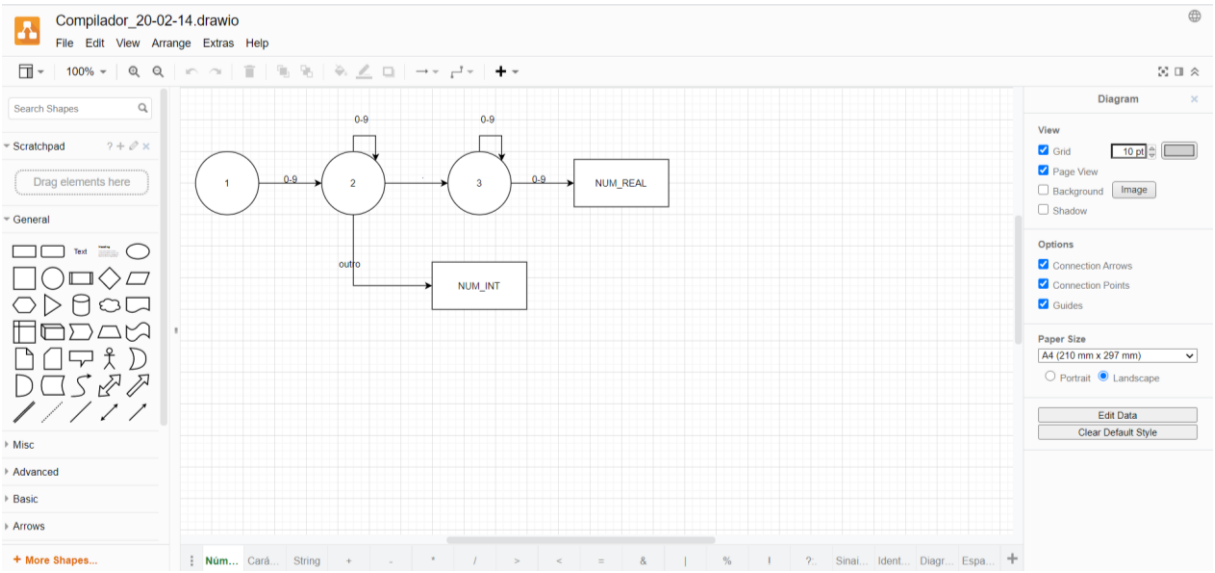
A figura 46 apresenta um exemplo de uso do *Canvas*. A metade superior mostra um exposto de uma página *web* e a metade inferior mostra o código que foi utilizado para criar o exposto. A *tag* HTML ‘<canvas>’ possui todas as propriedades necessárias para gerar o retângulo onde será manipulada a imagem que se deseja criar, como a altura e largura que o retângulo deve ter, o estilo da borda e o *id* para que o *JavaScript* consiga obter o elemento; a *tag* ‘<script>’ é onde o código *JavaScript* se responsabiliza por desenhar o círculo.

5.1.7 Diagrams.net

O *diagrams.net* é um software gratuito para a geração de diversos tipos de diagramas.

Para o presente trabalho, o *diagrams.net* foi utilizado na criação dos autômatos finitos da linguagem D+ que servem para ilustrar a análise léxica do compilador desenvolvido.

FIGURA 47 - Diagrams.net



FONTE: a própria autora

A figura 47 apresenta a interface do *diagrams.net.* No lado esquerdo são disponibilizados diversos itens de modelagem; no centro da imagem é onde cria-se o diagrama desejado; no lado direito têm-se outras configurações para os itens de modelagem; e no inferior é disponibilizado abas para a criar diagramas de um projeto separadamente.

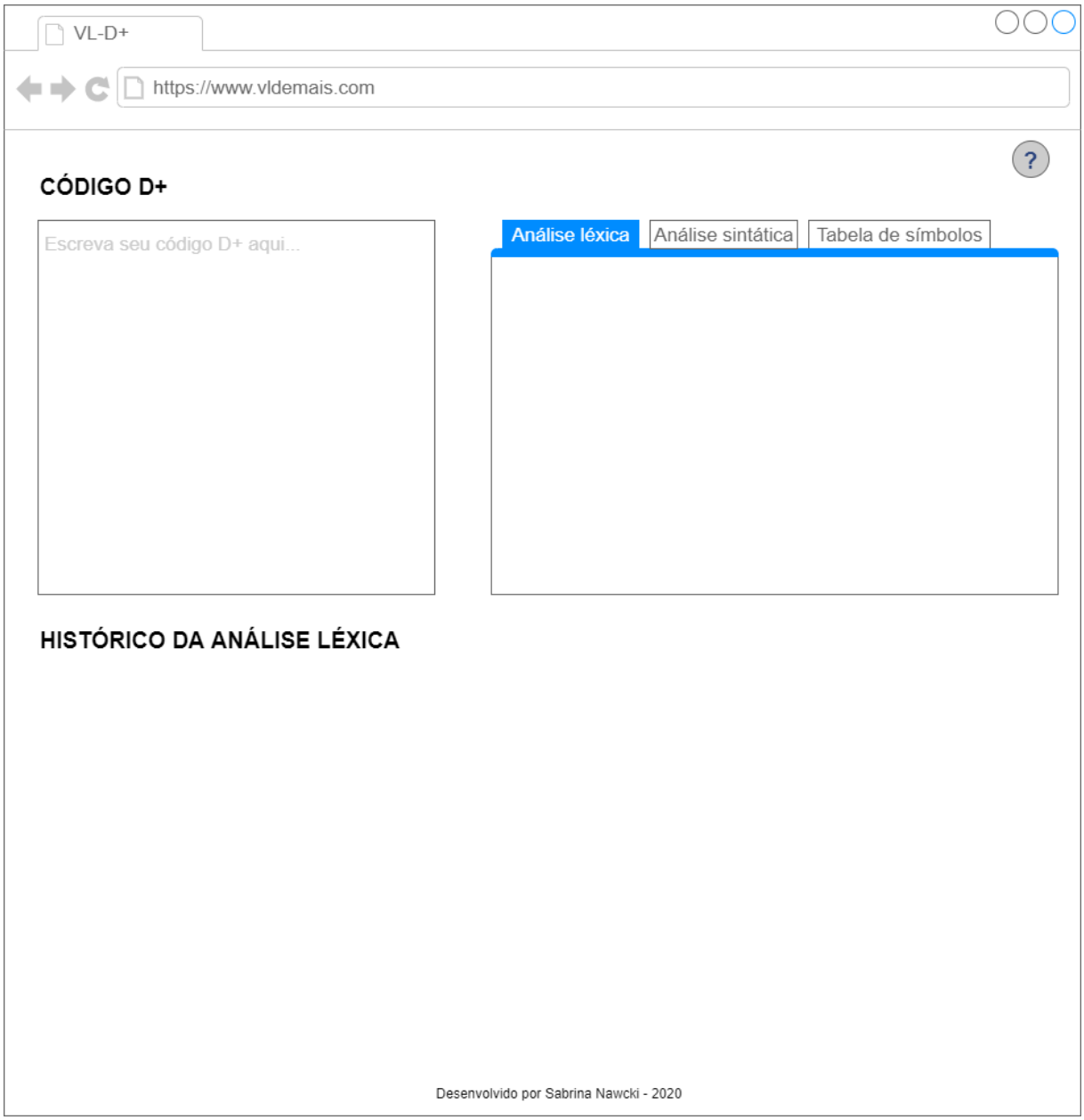
5.2 INTERFACE DA FERRAMENTA VL-D+

Nessa seção são apresentados os modelos, definidos como *mockup's* na área de *design*, das possíveis telas da ferramenta VL-D+ a serem desenvolvidas, tais como: a

tela inicial sem nenhuma interação do usuário e as telas da análise léxica, análise sintática e tabela de símbolos, todas apresentando uma interação feita pelo usuário.

A construção da interface da ferramenta VL-D+ foi pensada na usabilidade do usuário, dando-lhe liberdade de visualizar as fases do compilador, disponibilizadas pela ferramenta, quando quiser e sem ter a necessidade de reescrever o código ou abrir outra guia.

FIGURA 48 - *Mockup* da tela inicial

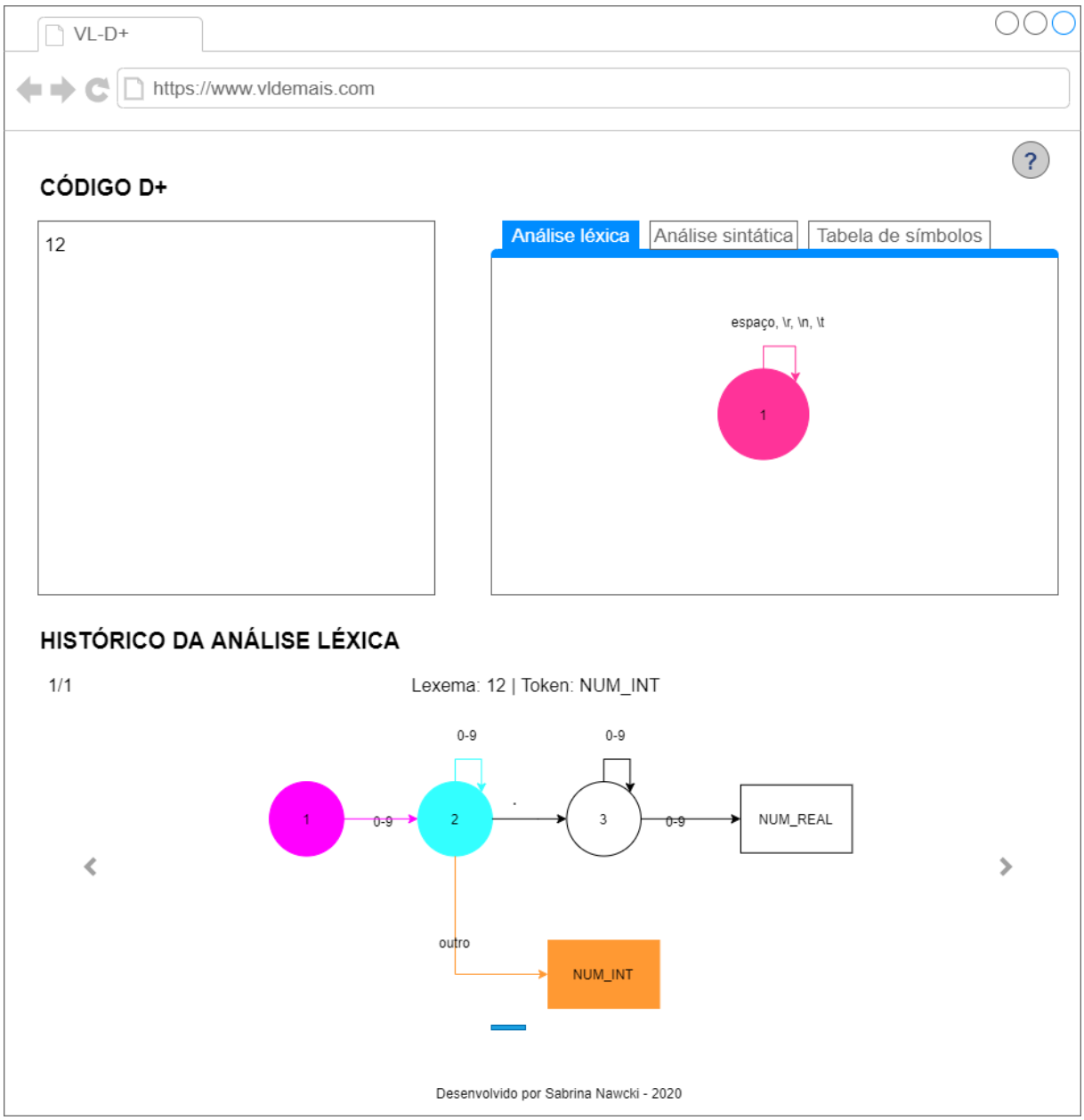


FONTE: a própria autora

A figura 48 apresenta o *mockup* da tela inicial da ferramenta VL-D+ sem nenhuma interação do usuário. Na parte superior da tela à esquerda tem-se uma caixa de texto onde o usuário deve digitar o código na linguagem D+; na parte superior da tela à direita tem-se um espaço destinado às resultantes do compilador separado pelas abas: análise léxica, análise sintática e tabela de símbolos.

Para a representação visual da análise léxica, optou-se a mostrar os autômatos finitos da linguagem D+ correspondentes ao código digitado pelo usuário e para a melhor usabilidade da ferramenta, foi vista a necessidade de manter um histórico com os autômatos finitos gerados pela análise léxica.

FIGURA 49 - *Mockup* da análise léxica

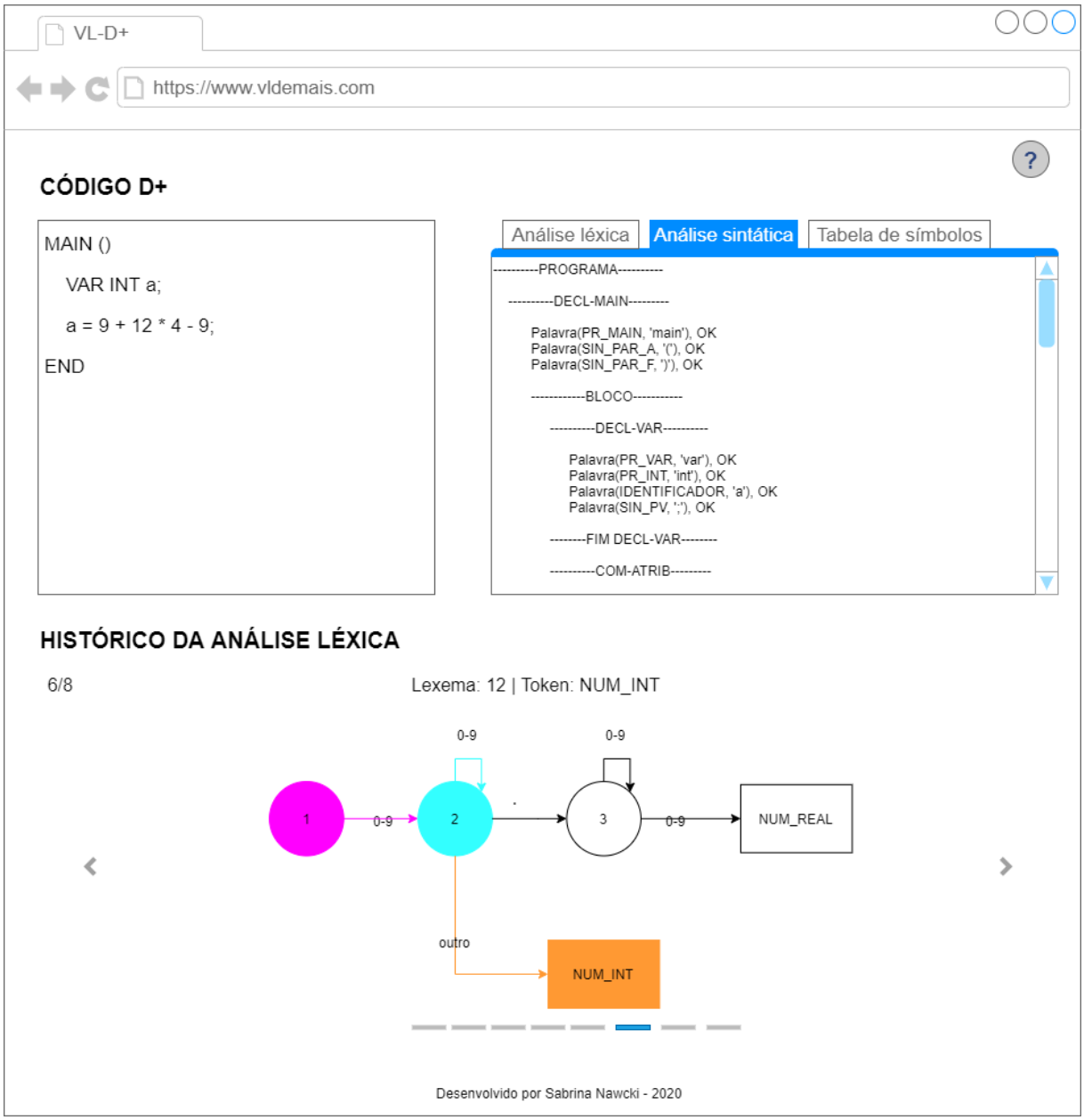


FONTE: a própria autora

A figura 49 apresenta o *mockup* da tela inicial após uma interação fictícia do usuário na aba da análise léxica. No exemplo o usuário digita na caixa de texto, na parte superior à esquerda da tela, o valor ‘12 ’ e como resultado da análise léxica são gerados dois autômatos finitos: o primeiro autômato finito pode ser visualizado no *slide show* do histórico da análise léxica, onde as cores simbolizam o percurso que a análise léxica tomou para concluir que ‘12’ é um número inteiro (NUM_INT); e como o espaço ‘ ’ é o valor corrente na caixa de texto, o segundo autômato finito gerado pode ser visualizado

na parte superior à direita da tela, como a análise léxica ignora separadores o autômato finito só representa um *loop* no ponto inicial.

FIGURA 50 - *Mockup* da análise sintática

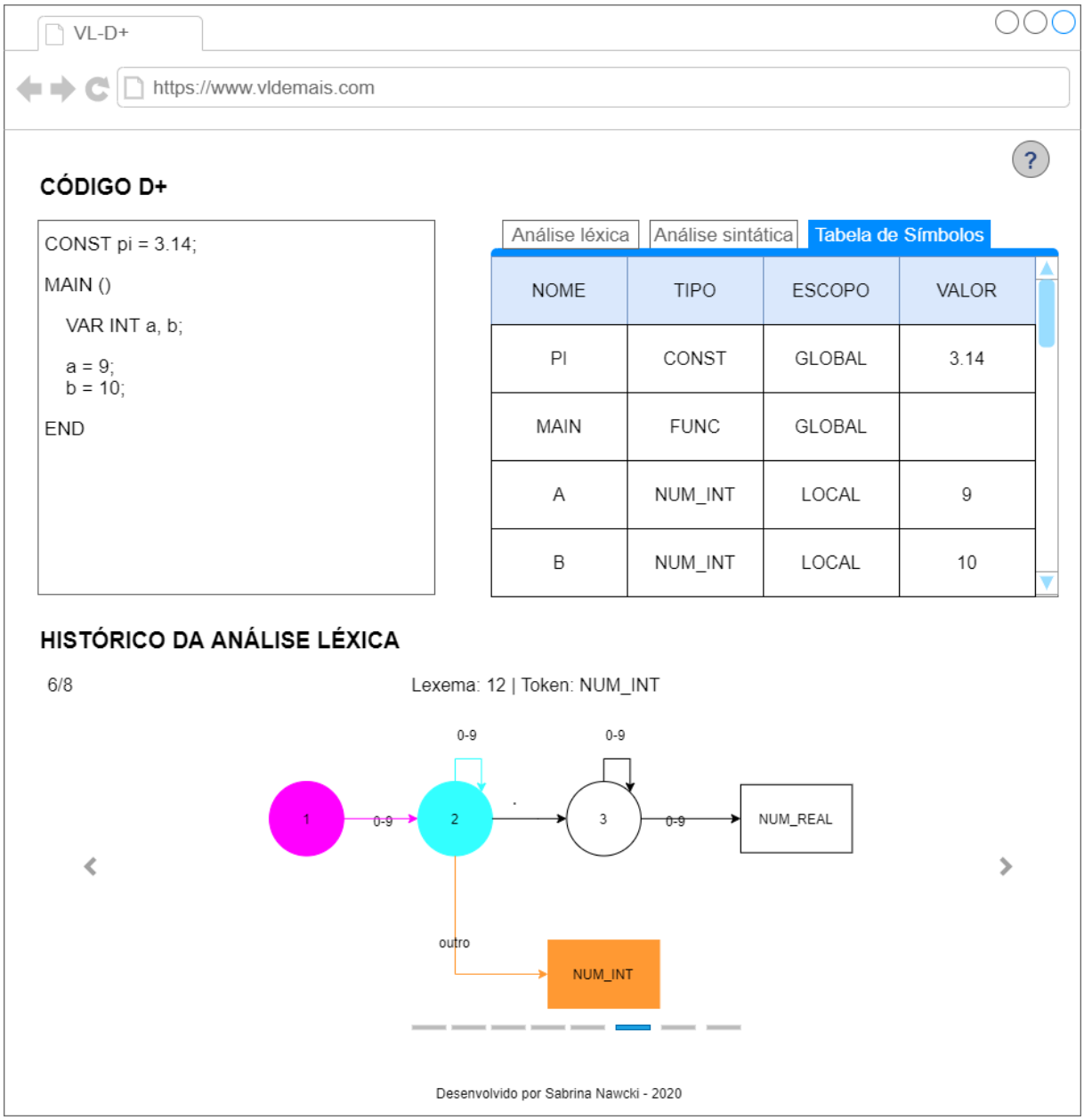


FONTE: a própria autora

A figura 50 apresenta o *mockup* da tela inicial após uma interação fictícia do usuário na aba da análise sintática. No exemplo o usuário digita na caixa de texto, na parte superior à esquerda da tela, um código válido na linguagem D+ e como resultado é gerada a árvore sintática correspondente ao código, que pode ser visualizada

completamente ao mexer na barra de *scroll* situada à direita. Mesmo com o usuário em outra aba, a análise léxica funciona normalmente e gera o seu histórico.

FIGURA 51 - *Mockup* da tabela de símbolos



FONTE: a própria autora

A figura 51 apresenta o *mockup* da tela inicial após uma interação fictícia do usuário na aba da tabela de símbolos. No exemplo o usuário digita na caixa de texto, na parte superior à esquerda da tela, um código válido na linguagem D+ e como resultado é gerada a tabela de símbolos correspondente ao código, que pode ser visualizada

completamente ao mexer na barra de *scroll* situada à direita. Mesmo com o usuário em outra aba, a análise léxica funciona normalmente e gera o seu histórico.

5.3 FUNCIONALIDADES DA FERRAMENTA VL-D+

Nessa seção são apresentadas as funcionalidades da ferramenta VL-D+ a serem desenvolvidas, mostrando cada componente de tela e como a interação do usuário com esses componentes deve ocorrer.

5.3.1 Inserção de código

A inserção de código deve ser feita numa caixa de texto situada a esquerda da interface e deve possibilitar que o usuário digite qualquer caractere, delete caracteres e utilize os atalhos de copiar (CTRL + V), colar (CTRL + C), recortar (CTRL + X), selecionar tudo (CTRL + A) e desfazer (CTRL + Z).

Quando houver inserção ou remoção de caracteres no final do código o processo do compilador deve continuar sem interrupções, caso contrário um botão de re-compilar código deve aparecer na tela e os processos do compilador devem parar até que o botão seja pressionado.

Caso haja erros de compilação no código inserido, o erro deve ser destacado dentro da caixa de texto em outra cor, facilitando o usuário a identificação de erros.

5.3.2 Aba de seleção

A aba de seleção deverá ficar no lado direito da interface e deve possibilitar que o usuário da aplicação consiga selecionar uma das seguintes opções: Análise Léxica, Análise Sintática, Tabela de Símbolos e caso haja erros de compilação deve haver a opção de Erros.

Cada aba deve realizar as suas funções independentemente se esteja selecionada, desta forma a aplicação irá estar com todas as abas atualizadas e demorará menos para apresentar seus resultados caso o usuário queira trocar de aba.

Quando uma aba estiver selecionada a sua cor deve mudar para demonstrar ao usuário qual aba está sendo apresentada no momento.

5.3.3 Análise léxica

A aba da Análise Léxica deve apresentar ao usuário o respectivo autômato finito do último lexema que está sendo construído na caixa de texto de inserção de código. As cores do diagrama devem mudar de acordo com a inserção de novos caracteres na caixa de texto mostrando ao usuário em qual passo do diagrama o compilador está.

Quando o compilador identificar um *token* ou um erro sintático o autômato finito deverá ser inserido no *slide show* do histórico da Análise Léxica. Itens que não são considerados como erro, mas fazem parte do código, como espaçadores, tabulações e quebras de linhas também são demonstrados no histórico, porém, não apresentam um *token.*

5.3.4 Análise sintática

A aba da Análise Sintática deve apresentar ao usuário a respectiva Árvore Sintática que o compilador está construindo a partir do código digitado na caixa de texto pelo usuário até o momento atual.

5.3.5 Tabela de símbolos

A aba da Tabela de Símbolos deve apresentar ao usuário a tabela de símbolos gerada pelo compilador até aquele momento com os atributos que são necessários para o processo de compilação.

5.3.6 Registro de erros

A aba de Erros tem o propósito de ajudar o usuário a visualizar quais foram os erros encontrados até o momento atual e deve aparecer apenas quando há algum erro de compilação. Se algum erro for corrigido o mesmo deve desaparecer da aba e caso todos os erros apresentados nessa aba sejam corrigidos, a aba deve desaparecer da tela.

6 **CONCLUSÃO**

Tendo em vista o exposto, é possível identificar que os discentes da disciplina de Compiladores possuem dificuldades no aprendizado, e fora dos centros de educação existem poucos recursos disponibilizados para o auxílio do discente na compreensão do conteúdo da disciplina.

Conforme apresentado no projeto, já existem ferramentas que reduzem essa problemática, porém todas trazem resultados estáticos das fases do compilador ao usuário e todas necessitam da instalação da ferramenta em um computador. Portanto, a visualização dos resultados se torna mais cansativa ao usuário e a instalação do produto às vezes pode ser algo inviável ou inatingível, tendo em vista que o instalador pode precisar de um sistema operacional específico.

Como alternativa, a ferramenta a ser criada no projeto deve abordar a tabela de símbolos e as fases da análise léxica e sintática dos compiladores numa interface *web* amigável e intuitiva, que traz os resultados da compilação do código digitado pelo usuário em tempo real, podendo assim ser mais interessante para o usuário e necessitando apenas de um dispositivo que consiga acessar a ferramenta através da internet.

As maiores dificuldades apresentadas para a realização desse projeto foram: apresentar a teoria dos compiladores, pois o seu conteúdo é muito vasto; apresentar as metodologias que serão utilizadas sem possuir um projeto; e, por fim, conseguir transmitir o propósito do projeto ao leitor do projeto, considerando que seu nível de conhecimento do assunto seja mínimo.

Os próximos passos para a continuação desta pesquisa serão: a criação da ferramenta VL-D+, o levantamento de testes necessários para a aprovação da ferramenta e a pesquisa de campo com discentes de compiladores.

REFERÊNCIAS

AHO, Alfred V.; SETHI Ravi; ULLMAN Jeffrey D. *Compiladores, Princípios, Técnicas e Ferramentas.* 1. ed. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora S.A., 1995.

AHO, Alfred V. *et al. Compiladores: princípios, técnicas e ferramentas.* 2. ed. São Paulo: Pearson Addison-Wesley, 2008.

BACKES, Jerônimo; DAHMER Alessandra. *C-gen – Ambiente Educacional para Geração de Compiladores*. Santa Cruz do Sul: Universidade de Santa Cruz do Sul, 2006.

BENYON, David. *Interação humano-computador.* 2. ed. São Paulo: Pearson Prentice Hall, 2011.

COSTA, Kelton A. P.; SILVA, Luis Alexandre; BRITO, Talita Pagani. *Auxílio no ensino em compiladores: software simulador como ferramenta de apoio na área de compiladores.* 2. ed. São Paulo: Simpósio Internacional de Educação, 2008.

FOLEISS, J. H. *et al*. *SCC: Um Compilador C como Ferramenta de Ensino de Compiladores*. São Paulo: Workshop sobre Educação em Arquitetura de Computadores – WEAC, 2009.

FURLAN, Diógenes Cogo. *Linguagem D+*. Curitiba, 2019.

GARRETT, Jesse James. *Os Elementos da Experiência do Usuário.* 2000. Disponível em: <http://www.jjg.net/elements/translations/elements_pt.pdf>. Acesso em: 08 mai. 2020.

MDN web docs, Mozilla. *Canvas.* 2020. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML/Canvas>. Acesso em: 22 mai. 2020.

MDN web docs, Mozilla. *Sobre JavaScript.* 2019. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/About_JavaScript>. Acesso em: 22 mai. 2020.

MICROSOFT. *Documentação do Visual Studio.* Entre 2019 e 2020. Disponível em: <https://docs.microsoft.com/pt-br/visualstudio/windows/?view=vs-2019>. Acesso em: 14 mai. 2020.

NETO, João José. *Introdução à compilação*. 2. ed. Rio de Janeiro: Elsevier, 2016.

NC STATE UNIVERSITY, The Center for Universal Design. *The Principles of Universal Design.* 2. ed. 1997. Disponível em <https://projects.ncsu.edu/ncsu/design/cud/about_ud/udprinciplestext.htm>. Acesso em: 07 mai. 2020.

RIBEIRO, Gabriel Pinto. *Interpretador da linguagem D+*. Trabalho de conclusão de curso (Bacharelado de Ciência da Computação) – Faculdade de Ciências Exatas, Universidade Tuiuti do Paraná, Curitiba, 2019.

SANTOS, Pedro Reis. LANGLOIS Thibault. *Compiladores: da teoria à prática.* 1. ed. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora Ltda., 2018.

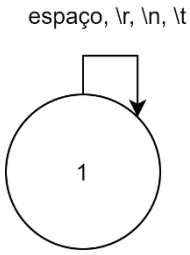
SCHNEIDER, Carlos Sérgio; PASSERINO, Liliana Maria; OLIVEIRA, Ricardo Ferreira. *Compilador Educativo VERTO: ambiente para aprendizagem de compiladores.* 3. ed. Rio Grande do Sul: Centro Interdisciplinar de Novas Tecnologias na Educação (CINTED) - Universidade Federal do Rio Grande do Sul (UFRGS), 2005.

SOBRAL, Wilma Sirlange. *Design de interfaces: introdução.* 1. ed. São Paulo: Érica, 2019.

W3SCHOOLS. *HTML Canvas Graphics.* Disponível em: <https://www.w3schools.com/html/html5_canvas.asp>. Acesso em: 25 mai. 2020.

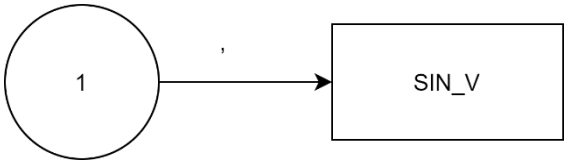
APÊNDICE A – AUTÔMATOS DA LINGUAGEM D+

FIGURA 52 - Autômato para separadores (estado 1)



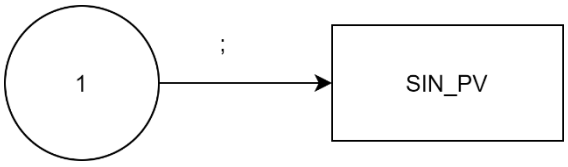
FONTE: a própria autora

FIGURA 53 - Autômato para a vírgula (estado 1)



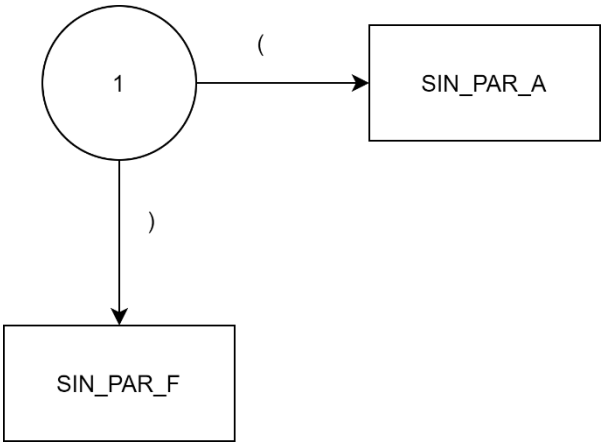
FONTE: a própria autora

FIGURA 54 - Autômato para o ponto e vírgula (estado 1)



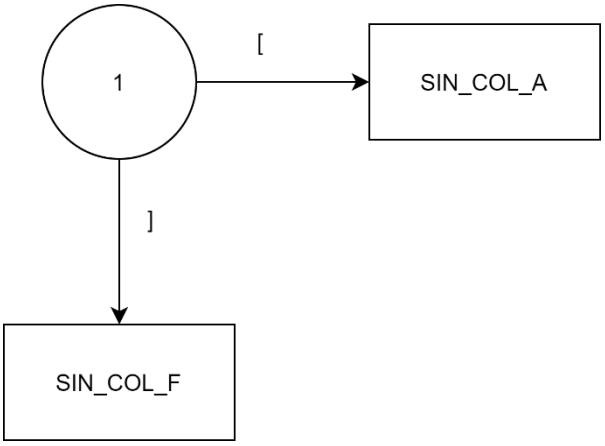
FONTE: a própria autora

FIGURA 55 - Autômato para os parênteses (estado 1)



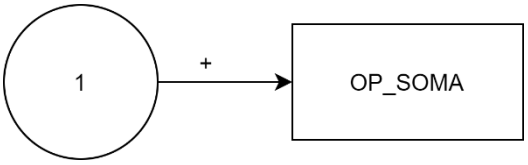
FONTE: a própria autora

FIGURA 56 - Autômato para os colchetes (estado 1)



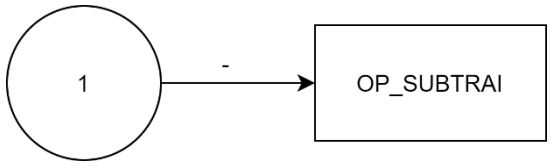
FONTE: a própria autora

FIGURA 57 - Autômato para o operador de adição (estado 1)



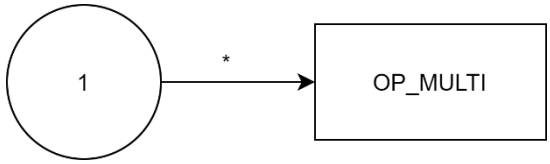
FONTE: a própria autora

FIGURA 58 - Autômato para o operador de subtração (estado 1)



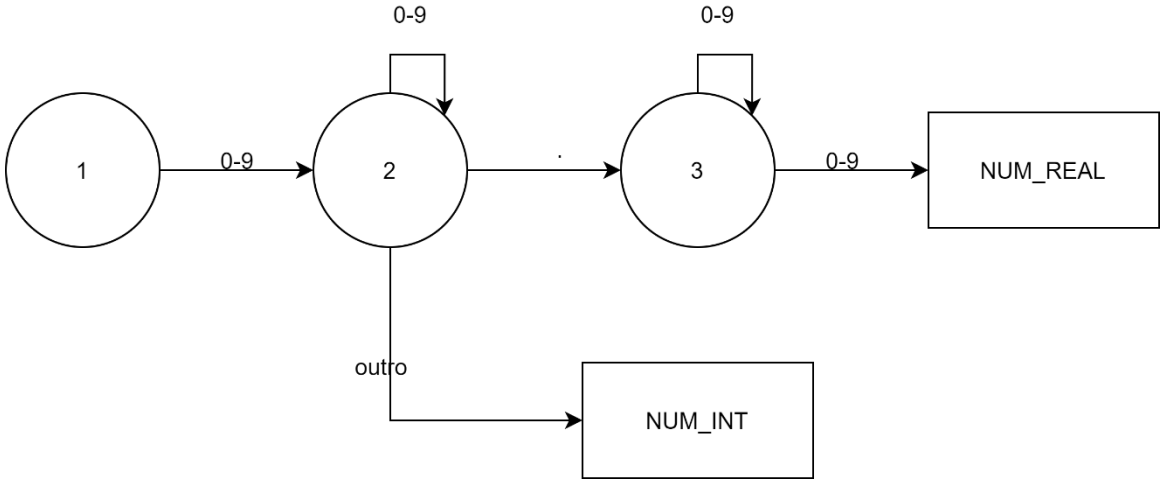
FONTE: a própria autora

FIGURA 59 - Autômato para o operador de multiplicação (estado 1)



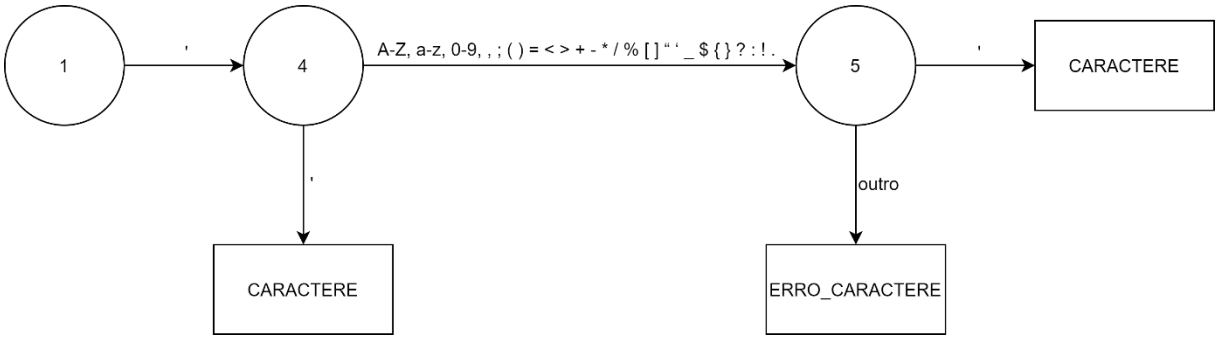
FONTE: a própria autora

FIGURA 60 - Autômato para números (estado 2)



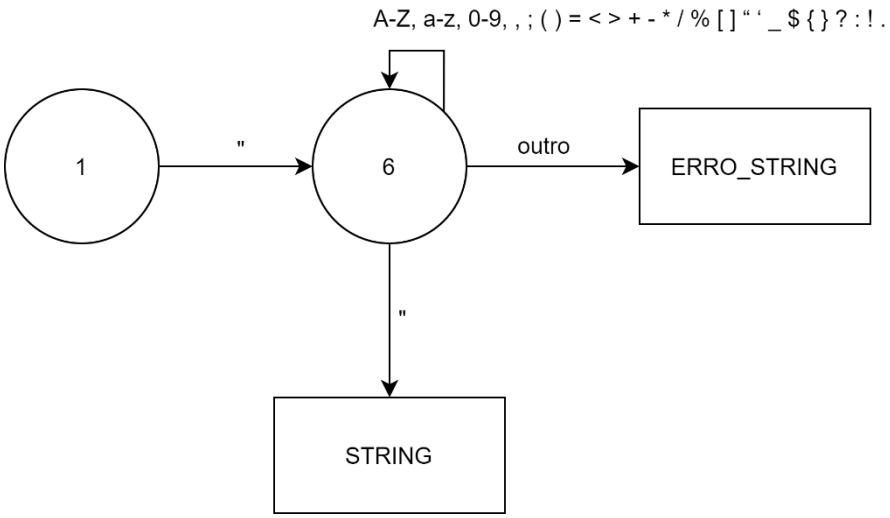
FONTE: a própria autora

FIGURA 61 - Autômato para caractere (estado 4)



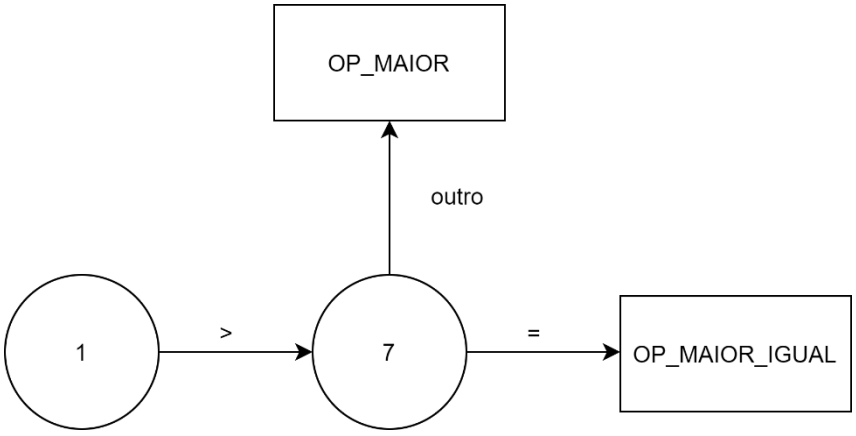
FONTE: a própria autora

FIGURA 62 - Autômato para *string* (estado 6)



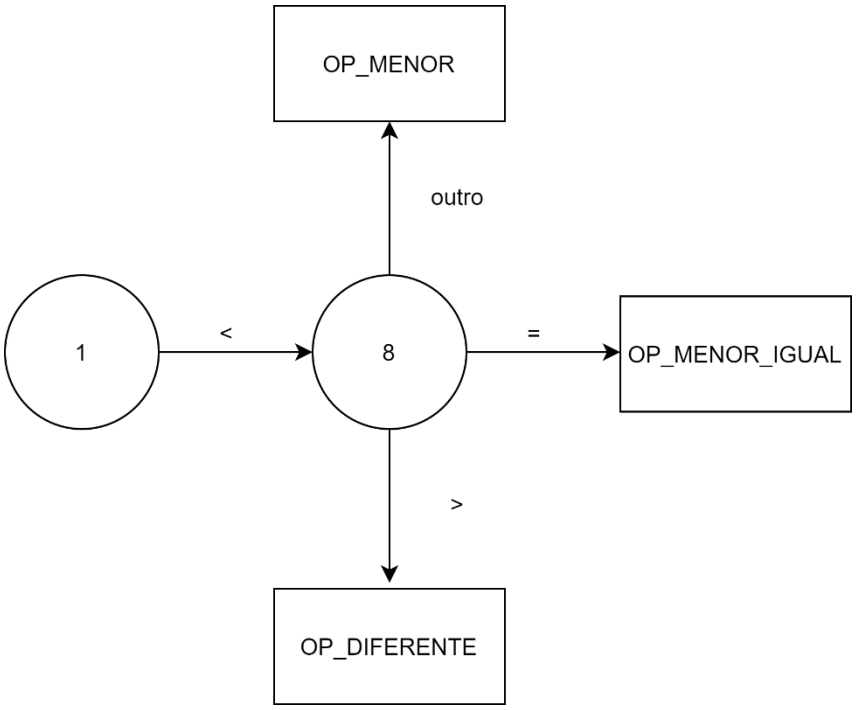
FONTE: a própria autora

FIGURA 63 - Autômato para o operador maior '>' (estado 7)



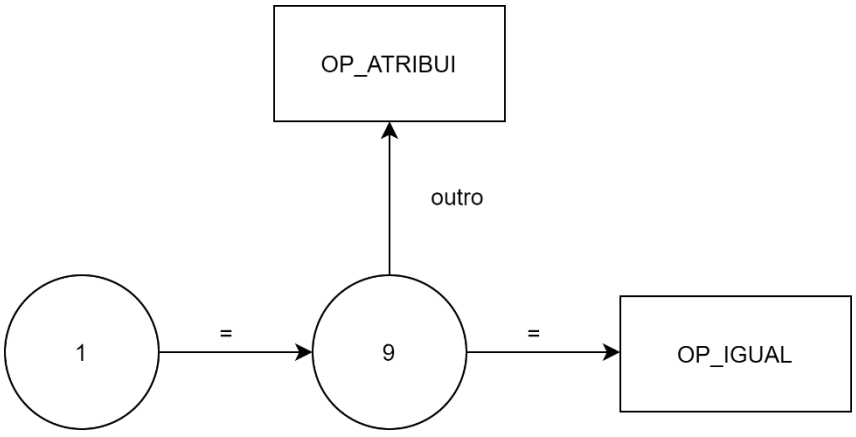
FONTE: a própria autora

FIGURA 64 - Autômato para o operador menor '<' (estado 8)



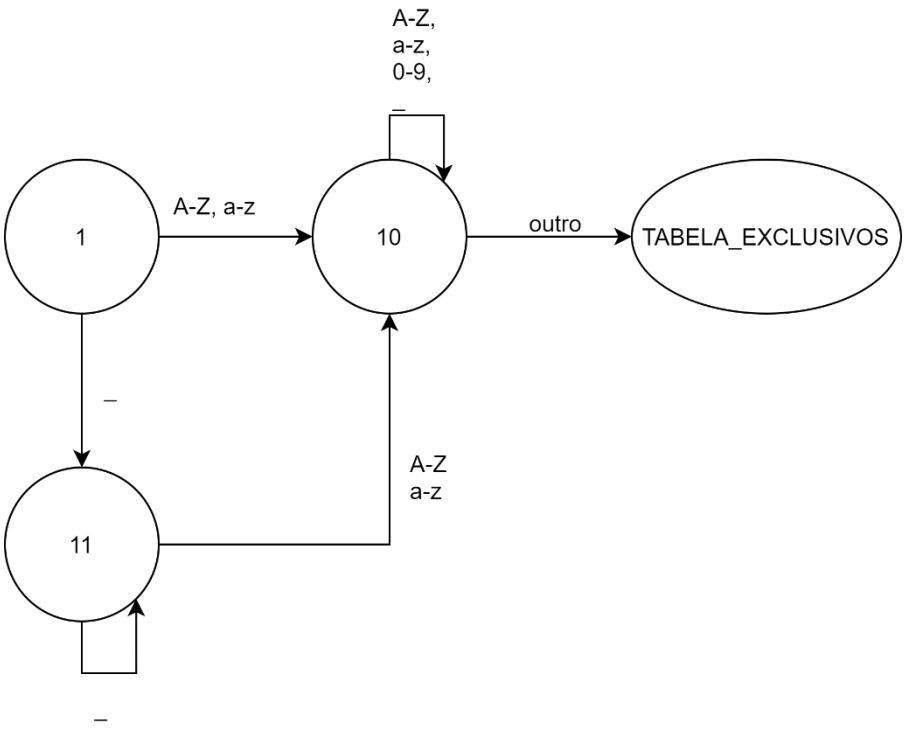
FONTE: a própria autora

FIGURA 65 - Autômato para o operador igual '=' (estado 9)



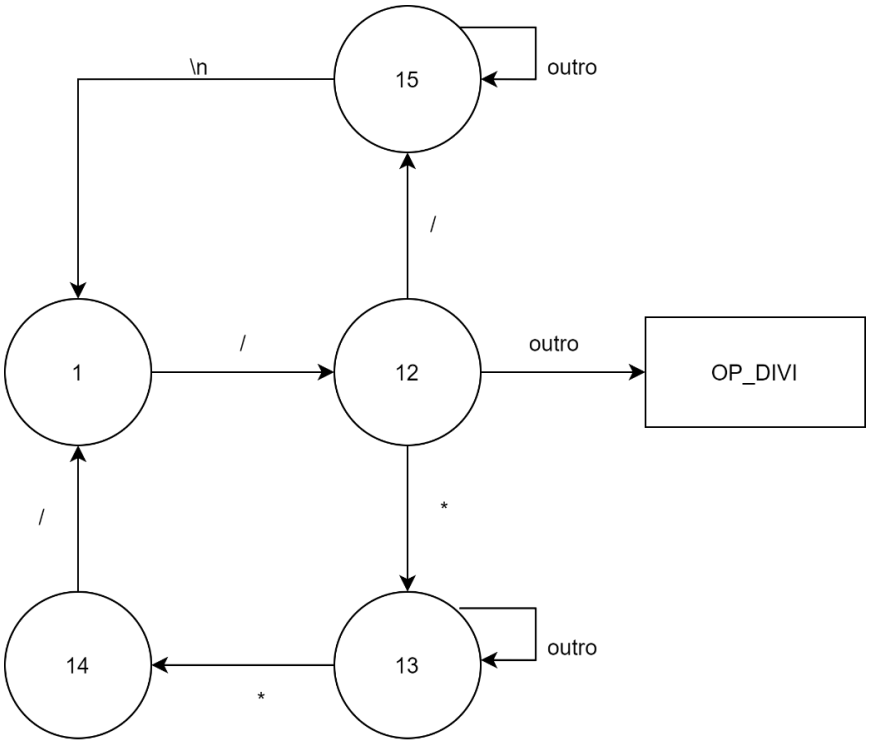
FONTE: a própria autora

FIGURA 66 - Autômato para valores exclusivos (estado 10)



FONTE: a própria autora

FIGURA 67 - Autômato para divisão e comentários (estado 12)



FONTE: a própria autora