Compiladores

Documentação da Linguagem #C0DEFF

Análise Léxica

UNIVERSIDADE FEDERAL DA BAHIA
CIÊNCIA DA COMPUTAÇÃO

DAVID SAMPAIO MOURA
JOÃO VICTOR SANTANA COELHO
MATHEUS OLIVEIRA MOLINARI RAMOS

DOCUMENTAÇÃO DA LINGUAGEM #CODEFF: ANÁLISE LÉXICA

Salvador - BA
20 de Outubro de 2025

Sumário

1. Introdução

Este trabalho possui como escopo apresentar a documentação da linguagem #C0DEFF, uma linguagem baseada nas cores e nos seus códigos hexadecimais para definição de suas estruturas.

Para explicitação da linguagem, descreve-se o seu funcionamento a nível de construção de código, detalhando como caracteres alfanuméricos e símbolos são mapeados para cores específicas, e como essas representações cromáticas constituem a base para a formação de programas na linguagem.

Além disso, é apresentado o analisador léxico da linguagem, que detalha minuciosamente as regras empregadas para identificar e classificar os elementos do código, incluindo caracteres ASCII, letras, dígitos e símbolos de pontuação.

Por fim, são expostos exemplos de funcionamento e os diagramas de transição com o escopo de completar o entendimento sobre a #C0DEFF e permitir que o leitor possa programar com completude usando a mesma.

2. Como funciona a linguagem

A linguagem proposta, denominada #C0DEFF, apresenta-se como uma formulação estética e estrutural que integra conceitos cromáticos à representação textual, usando códigos hexadecimais das cores como base de sua sintaxe. Cada elemento da tabela ASCII (reduzida, por convenção da linguagem) ou palavra reservada adquire um valor cromático específico, resultando numa linguagem expressa por meio de cores.

Para compreender a linguagem, é preciso perscrutar sobre o funcionamento das estruturas vigentes na própria. Primeiramente, se faz necessário estudar sobre as palavras chave que possuem uma lista predefinida de códigos hexadecimais específicos que são compostos por letras minúsculas e dígitos na formação. Segue abaixo a tabela 1 sobre as definições.

Cor Hexadecimal	Função
#03fcd7	Palavra reservada para variáveis do tipo inteiro
#ff00c3	Palavra reservada para variáveis do tipo real
#1900ff	Palavra reservada para variáveis do tipo booleano
#00ff04	Palavra reservada para variáveis do tipo cadeia de caracteres
#eaff00	Palavra reservada para inicialização de laços condicionais
#ff0000	Palavra reservada para inicialização de trechos de recusa de condição
#006eff	Palavra reservada para inicialização de trechos condicionais
#ff9a00	Palavra reservada para inicialização de laços sequenciais
#0f5a00	Palavra reservada para retornos vazios de funções
#3f2c1e	Palavra reservada para retornos de funções
#ff00ff	Palavra reservada para paradas forçadas de laços
#84ffce	Palavra reservada para representação de valor-verdade V
#520000	Palavra reservada para representação de valor-verdade F
#ff006e	Palavra reservada para variáveis do tipo caractere-literal
#d9d2d2	Palavra reservada para valor nulo
#014100	Palavra reservada para inicialização de comentário de linha
#f0b	Palavra reservada para delimitação de comentário de bloco

Tabela 1 - Definição das palavras reservadas

Superada essa etapa, adentra-se na lógica que rege os caracteres pertencentes à tabela ASCII, considera–se, para tanto, o intervalo compreendido entre 32 e 126

correspondente aos caracteres “printáveis”. A partir desse conjunto, realiza-se a divisão de 16.777.216 (número total de combinações possíveis em um código hexadecimal de seis dígitos) pelo tamanho do intervalo referido e é multiplicado o valor obtido pela posição do caractere entre 32 e 126, obtendo-se assim o número decimal o qual deve ser convertido para um código hexadecimal de 6 dígitos que é referente ao símbolo da tabela ASCII na linguagem, à título de exemplo: o caracter espaço da tabela localizado na posição 32 na mesma, é o símbolo 0 do intervalo, logo ao ser feito (16.777.216 / 94) * 0 = 0, após conversão, define-se o espaço como #000000 em #C0DEFF.

Além disso, para distinguir visual e lexicalmente os diferentes caracteres alfanuméricos, a linguagem define um padrão cromático específico. Para os dígitos (0-9), definiu-se que eles ocupariam metade do espectro do canal verde, o que corresponde a 128 no intervalo total de 256 possíveis valores (0 a 255). Dessa forma, ao dividir 128 pela quantidade de dígitos, obtém-se um incremento médio de 12,8 unidades por dígito. Como o canal de cor trabalha com valores inteiros, utiliza-se a parte inteira desse valor (12) como variação dos tons de verde. Por fim, para que o último dígito (9) alcance o limite superior do intervalo verde, adiciona-se um deslocamento inicial de 147 e esse valor de compensação garante que, após as variações sucessivas de 12 unidades, o maior dígito atinja aproximadamente 255.

Para as letras maiúsculas e minúsculas, utilizou-se o intervalo do espectro vermelho no qual a primeira parte (0 a 127) do mesmo foi direcionado para as letras maiúsculas e a segunda parte (128 a 255) para as minúsculas, a lógica para seleção das cores foi a mesma utilizada para os dígitos, com as seguintes mudanças com o escopo de adaptação: a divisão para 26 valores (são 26 letras) e o incremento de compensação sendo 128, posto que foi usado o resultado racional da divisão de 128 pela quantia de letras. Segue a tabela 2 que descreve esse funcionamento.

Caracteres	Fórmula
Dígitos	$(\lambda - 48) \times 12 + 147$
Letras maiúsculas	$\lfloor(\lambda - 65) \times 2,4519 + 128\rfloor$
Letras minúsculas	$\lfloor(\lambda - 97) \times 2,4519 + 128 + 63.75\rfloor$
Outros caracteres	$(\lambda - 32) \times 178481$

Tabela 2 - Fórmulas para converter caractere da tabela ASCII em códigos hexadecimais aceitos

Com a compreensão desses conceitos, pode-se avançar para o estudo do analisador léxico, responsável por definir a sequência de regras que estruturam o funcionamento do #C0DEFF como um sistema de programação completo. Esse analisador detalha a forma como

cada elemento do código é identificado, classificado e transformado em tokens, permitindo que o compilador ou interpretador compreenda e processe corretamente as instruções.

3. Código Flex

3.1. Seção de Definições em C

Nesta seção, serão feitas algumas definições de variáveis no escopo da linguagem C que serão úteis para o analisador. Primeiramente, é definido a variável ***yylval*** que vai receber o valor concreto de cada token lido pelo analisador.

Logo após, é inicializada a tabela de símbolos como um *array* de *strings* e também é inicializada uma variável contadora de símbolos. Abaixo existe uma função para buscar a posição de um símbolo na tabela ou então adicionar o símbolo na tabela, caso ele ainda não exista.

3.2. Enumeração de Tokens Refinada

Nesta seção, é definida a enumeração *TokenType* que atribui um valor inteiro numérico único a cada tipo de token da linguagem #C0DEFF. São eles:

3.2.1. Fim de arquivo

- *T_EOF;*

3.2.2. Palavras-chave

- *T_IF;*
- *T_ELSE;*
- *T_WHILE;*
- *T_INT_KEYWORD;*
- *T_FLOAT_KEYWORD;*
- *T_RETURN;*
- *T_CHAR_KEYWORD;*
- *T_BOOL;*
- *T_STR;*
- *T_FOR;*
- *T_VOID;*
- *T_BREAK;*
- *T_TRUE;*
- *T_FALSE;*
- *T_NULL;*

3.2.3. Identificadores e Literais

- *T_ID;*
- *T_INTEGER;*
- *T_FLOAT;*
- *T_STRING;*
- *T_LITERAL_CHAR;*

3.2.4. Operadores

- *T_OP_IGUALDADE;*
- *T_OP_DIFERENTE;*
- *T_OP_MENOR;*
- *T_OP_MAIOR;*
- *T_OP_MENOR_IGUAL;*
- *T_OP_MAIOR_IGUAL;*
- *T_OP_SOMA;*
- *T_OP_SUB;*
- *T_OP_MULT;*
- *T_OP_DIV;*
- *T_OP_ATRIBUICAO;*
- *T_OP_TERNARY_IF;*
- *T_OP_TERNARY_ELSE;*

3.2.5. Separadores

- *T_PONTO_VIRGULA;*
- *T_VIRGULA;*
- *T_PARENTESES_ESQ;*
- *T_PARENTESES_DIR;*
- *T_CHAVES_ESQ;*
- *T_CHAVES_DIR;*
- *T_COLCHETES_ESQ;*
- *T_COLCHETES_DIR;*

3.2.6. Erro

- *T_UNKNOWN.*

3.3. Definições de Expressões Regulares

Nesta seção, são atribuídos alguns apelidos à expressões regulares que serão usadas várias vezes durante a seção de regras. Os apelidos são:

- ASCII:

```
(#000000)|(#02B931)|(#057262)|(#082B93)|(#0AE4C4)|(#0D9DF5)|
(#105726)|(#131057)|(#15C988)|(#1882B9)|(#1B3BEA)|(#1DF51B)|
(#20AE4C)|(#23677D)|(#2620AE)|(#28D9DF)|(#009300)|(#009F00)|
(#00AB00)|(#00B700)|(#00C300)|(#00CF00)|(#00DB00)|(#00E700)|
(#00F300)|(#00FF00)|(#46CEFA)|(#49882B)|(#4C415C)|(#4EFA8D)|
(#51B3BE)|(#546CEF)|(#572620)|(#800000)|(#820000)|(#840000)|
(#870000)|(#890000)|(#8C0000)|(#8E0000)|(#910000)|(#930000)|
(#960000)|(#980000)|(#9A0000)|(#9D0000)|(#9F0000)|(#A20000)|
(#A40000)|(#A70000)|(#A90000)|(#AC0000)|(#AE0000)|(#B10000)|
(#B30000)|(#B50000)|(#B80000)|(#BA0000)|(#BD0000)|(#A0AE4B)|
(#A3677C)|(#A620AD)|(#A8D9DE)|(#AB930F)|(#AE4C40)|(#BF0000)|
(#C20000)|(#C40000)|(#C70000)|(#C90000)|(#CC0000)|(#CE0000)|
(#D00000)|(#D30000)|(#D50000)|(#D80000)|(#DA0000)|(#DD0000)|
(#DF0000)|(#E20000)|(#E40000)|(#E60000)|(#E90000)|(#EB0000)|
(#EE0000)|(#F00000)|(#F30000)|(#F50000)|(#F80000)|(#FA0000)|
(#FD0000)|(#F7D46B)|(#FA8D9C)|(#FD46CD)|(#FFFFFE)
```

- ASCII_SEM_ASPAS:

```
(#000000)|(#02B931)|(#082B93)|(#0AE4C4)|(#0D9DF5)|(#105726)|
(#131057)|(#15C988)|(#1882B9)|(#1B3BEA)|(#1DF51B)|(#20AE4C)|
(#23677D)|(#2620AE)|(#28D9DF)|(#009300)|(#009F00)|(#00AB00)|
(#00B700)|(#00C300)|(#00CF00)|(#00DB00)|(#00E700)|(#00F300)|
(#00FF00)|(#46CEFA)|(#49882B)|(#4C415C)|(#4EFA8D)|(#51B3BE)|
(#546CEF)|(#572620)|(#800000)|(#820000)|(#840000)|(#870000)|
(#890000)|(#8C0000)|(#8E0000)|(#910000)|(#930000)|(#960000)|
(#980000)|(#9A0000)|(#9D0000)|(#9F0000)|(#A20000)|(#A40000)|
(#A70000)|(#A90000)|(#AC0000)|(#AE0000)|(#B10000)|(#B30000)|
(#B50000)|(#B80000)|(#BA0000)|(#BD0000)|(#A0AE4B)|(#A3677C)|
(#A620AD)|(#A8D9DE)|(#AB930F)|(#AE4C40)|(#BF0000)|(#C20000)|
(#C40000)|(#C70000)|(#C90000)|(#CC0000)|(#CE0000)|(#D00000)|
(#D30000)|(#D50000)|(#D80000)|(#DA0000)|(#DD0000)|(#DF0000)|
(#E20000)|(#E40000)|(#E60000)|(#E90000)|(#EB0000)|(#EE0000)|
(#F00000)|(#F30000)|(#F50000)|(#F80000)|(#FA0000)|(#FD0000)|
(#F7D46B)|(#FA8D9C)|(#FD46CD)|(#FFFFFE)
```

- ASCII_SEM_ASPA:
  `(#000000)|(#02B931)|(#057262)|(#082B93)|(#0AE4C4)|(#0D9DF5)|(#105726)|(#15C988)|(#1882B9)|(#1B3BEA)|(#1DF51B)|(#20AE4C)|(#23677D)|(#2620AE)|(#28D9DF)|(#009300)|(#009F00)|(#00AB00)|(#00B700)|(#00C300)|(#00CF00)|(#00DB00)|(#00E700)|(#00F300)|(#00FF00)|(#46CEFA)|(#49882B)|(#4C415C)|(#4EFA8D)|(#51B3BE)|(#546CEF)|(#572620)|(#800000)|(#820000)|(#840000)|(#870000)|(#890000)|(#8C0000)|(#8E0000)|(#910000)|(#930000)|(#960000)|(#980000)|(#9A0000)|(#9D0000)|(#9F0000)|(#A20000)|(#A40000)|(#A70000)|(#A90000)|(#AC0000)|(#AE0000)|(#B10000)|(#B30000)|(#B50000)|(#B80000)|(#BA0000)|(#BD0000)|(#A0AE4B)|(#A3677C)|(#A620AD)|(#A8D9DE)|(#AB930F)|(#AE4C40)|(#BF0000)|(#C20000)|(#C40000)|(#C70000)|(#C90000)|(#CC0000)|(#CE0000)|(#D00000)|(#D30000)|(#D50000)|(#D80000)|(#DA0000)|(#DD0000)|(#DF0000)|(#E20000)|(#E40000)|(#E60000)|(#E90000)|(#EB0000)|(#EE0000)|(#F00000)|(#F30000)|(#F50000)|(#F80000)|(#FA0000)|(#FD0000)|(#F7D46B)|(#FA8D9C)|(#FD46CD)|(#FFFFFE)`

- DIGITO:
  `(#009300)|(#009F00)|(#00AB00)|(#00B700)|(#00C300)|(#00CF00)|(#00DB00)|(#00E700)|(#00F300)|(#00FF00)`

- MINUSCULO:
  `(#BF0000)|(#C20000)|(#C40000)|(#C70000)|(#C90000)|(#CC0000)|(#CE0000)|(#D00000)|(#D30000)|(#D50000)|(#D80000)|(#DA0000)|(#DD0000)|(#DF0000)|(#E20000)|(#E40000)|(#E60000)|(#E90000)|(#EB0000)|(#EE0000)|(#F00000)|(#F30000)|(#F50000)|(#F80000)|(#FA0000)|(#FD0000)`

- MAIUSCULO:
  `(#800000)|(#820000)|(#840000)|(#870000)|(#890000)|(#8C0000)|(#8E0000)|(#910000)|(#930000)|(#960000)|(#980000)|(#9A0000)|(#9D0000)|(#9F0000)|(#A20000)|(#A40000)|(#A70000)|(#A90000)|(#AC0000)|(#AE0000)|(#B10000)|(#B30000)|(#B50000)|(#B80000)|(#BA0000)|(#BD0000)`

- LETRA: `{MINUSCULO}|{MAIUSCULO}`
- ALPHANUMERICO: `{LETRA}|{DIGITO}`
- UNDERLINE: `#AB930F`
- ESPACO: `#000000`

- COMMENT_LINE: #014100
- COMMENT_BLOCK: #f0b
- EXCLAMACAO: #02B931
- ASPAS: #057262
- ASPA: #131057
- HASHTAG: #082B93
- PERCENT: #0D9DF5
- E_COMERCIAL: #105726
- PARENTESIS_L: #15C988
- PARENTESIS_R: #1882B9
- ASTERISCO: #1B3BEA
- MAIS: #1DF51B
- VIRGULA: #20AE4C
- MENOS: #23677D
- PONTO: #2620AE
- BARRA: #28D9DF
- DOIS_PONTOS: #46CEFA
- PONTO_E_VIRGULA: #49882B
- MENOR_QUE: #4C415C
- IGUAL: #4EFA8D
- MAIOR_QUE: #51B3BE
- INTERROGACAO: #546CEF
- ARROBA: #572620
- COLCHETE_L: #A0AE4B
- CONTRA_BARRA: #A3677C
- COLCHETE_R: #A620AD
- CIRCUNFLEXO: #A8D9DE
- CHAVE_L: #F7D46B
- PIPE: #FA8D9C
- CHAVE_R: #FD46CD

OBS: Notações dentro de chaves como “`{MINUSCULO}`” ou “`{DIGITO}`” são referências à outros apelidos já criados anteriormente.

3.4. Seção de Regras

Na seção de regras, é onde ocorre a significação das expressões regulares que serão lidas pela linguagem e como o analisador deve interpretar cada uma delas. As regras são as seguintes:

- `[\t]+`: Nada é feito ao encontrar uma ou mais tabulações, esse caractere é ignorado pelo analisador.
- `({ESPACO})+`: Ao encontrar uma cadeia de caracteres que representa o apelido “ESPACO” por uma ou mais vezes, o analisador ignora-os também.
- `\n`: Ao encontrar uma quebra de linha, o contador de linhas: *yylineno* é incrementado automaticamente, mas não é feita a tokenização deste caractere.
- `{COMMENT_LINE}[^\n]*`: Ao identificar um operador de comentário em linha seguido de quaisquer sequências de caracteres até o fim da linha, o analisador entende que é um comentário e ignora a linha.
- `{COMMENT_BLOCK}([^#]|(#+[^f])|(#+f[^0])|(#+f0[^b]))*{COMMENT_BLOCK}`: Ao identificar um operador de comentário em bloco, o analisador consome todos os caracteres que não formem o operador de comentário em bloco até o próximo operador de comentário em bloco e ignora toda essa seção pois é identificada como um bloco de comentário.
- `#006eff`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_IF*.
- `#ff0000`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_ELSE*.
- `#eaff00`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_WHILE*.
- `#03fcd7`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_INT_KEYWORD*.
- `#ff00c3`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_FLOAT_KEYWORD*.

- `#3f2c1e`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_RETURN*.
- `#ff006e`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_CHAR_KEYWORD*.
- `#1900ff`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_BOOL*.
- `#00ff04`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_STR*.
- `#ff9a00`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_FOR*.
- `#0f5a00`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_VOID*.
- `#ff00ff`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_BREAK*.
- `#84ffce`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_TRUE*.
- `#520000`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_FALSE*.
- `#d9d2d2`: Ao identificar essa sequência específica de caracteres, o analisador identifica como uma palavra reservada e retorna o identificador único *T_NULL*.
- `({DIGITO})+`: Ao identificar qualquer sequência que corresponda a um *DIGITO* na nossa linguagem por uma ou mais vezes, o analisador interpreta-o como um inteiro e retorna o identificador *T_INTEGER.*
- `({DIGITO})+({PONTO})({DIGITO})+`: Ao identificar qualquer sequência que corresponda a um *DIGITO* na nossa linguagem por uma ou mais vezes, seguido de uma sequência que representa um *PONTO*, seguido de um ou mais representações de *DIGITO* novamente, o analisador interpreta-o como um número de ponto flutuante e retorna o identificador *T_FLOAT.*
- `({ASPAS})({ASCII_SEM_ASPAS})*({ASPAS})`: Ao identificar qualquer sequência que corresponda a *ASPAS* na nossa linguagem, seguido por qualquer sequência ASCII (exceto *ASPAS*) aceita pela linguagem e finalizando com a

representação de *ASPAS* novamente, o analisador interpreta-o como uma cadeia de caracteres e retorna o identificador *T_STRING.*

- `({ASPA})(({ASCII_SEM_ASPA})|({CONTRA_BARRA})(({MINUSCULO})|({ASPA})|({CONTRA_BARRA})))({ASPA})`: Ao identificar qualquer sequência que corresponda a *ASPA* na linguagem, seguido por qualquer representação ASCII (exceto *ASPA*) ou uma sequência iniciada pela representação de *CONTRA_BARRA* seguida por alguma representação de letra minúscula, por outra representação de *CONTRA_BARRA* ou até mesmo por uma representação de *ASPA*, e finalizada por outra representação de ASPA que é aceita pela linguagem, é interpretada como um caractere literal e retorna o identificador *T_LITERAL_CHAR.*
- `(({LETRA})|({UNDERLINE}))((({LETRA})|({UNDERLINE}))|({DIGITO}))*`: Ao identificar qualquer sequência que comece com alguma representação de *LETRA* ou *UNDERLINE* da linguagem seguido por zero ou mais representações de *LETRA*, *UNDERLINE*, ou *DIGITO* da linguagem, o analisador interpreta-o como uma variável e atribui à *yylval* o valor da posição daquela variável na tabela de símbolos e retorna o identificador *T_ID.*
- `({IGUAL})({IGUAL})`: Ao identificar uma correspondência a uma sequência de duas representações do *IGUAL* na linguagem, o analisador interpreta-o como um operador de igualdade e retorna o identificador *T_OP_IGUALDADE.*
- `({EXCLAMACAO})({IGUAL})`: Ao identificar a sequência da representação da *EXCLAMACAO* seguido da representação do *IGUAL*, o analisador interpreta-o como um operador de desigualdade e retorna o identificador *T_OP_DIFERENTE.*
- `({MENOR_QUE})({IGUAL})`: Ao identificar a sequência da representação do *MENOR_QUE* seguido da representação do *IGUAL*, o analisador interpreta-o como um operador de menor ou igual e retorna o identificador *T_OP_MENOR_IGUAL.*
- `({MAIOR_QUE})({IGUAL})`: Ao identificar a sequência da representação do *MAIOR_QUE* seguido da representação do *IGUAL*, o analisador interpreta-o como um operador de maior ou igual e retorna o identificador *T_OP_MAIOR_IGUAL.*
- `({MENOR_QUE})`: Ao identificar a representação do *MENOR_QUE,* o analisador interpreta-o como um operador de menor e retorna o identificador *T_OP_MENOR.*

- ({MAIOR_QUE}): Ao identificar a representação do *MAIOR_QUE,* o analisador interpreta-o como um operador de maior e retorna o identificador *T_OP_MAIOR.*
- ({MAIS}): Ao identificar a representação do *MAIS,* o analisador interpreta-o como um operador de soma e retorna o identificador *T_OP_SOMA.*
- ({MENOS}): Ao identificar a representação do *MENOS,* o analisador interpreta-o como um operador de subtração e retorna o identificador *T_OP_SUB.*
- ({ASTERISCO}): Ao identificar a representação do *ASTERISCO,* o analisador interpreta-o como um operador de multiplicacao e retorna o identificador *T_OP_MULT.*
- ({BARRA}): Ao identificar a representação do *BARRA,* o analisador interpreta-o como um operador de divisão e retorna o identificador *T_OP_DIV.*
- ({IGUAL}): Ao identificar a representação do *IGUAL,* o analisador interpreta-o como um operador de atribuição e retorna o identificador *T_OP_ATRIBUICAO.*
- ({INTERROGACAO}): Ao identificar a representação do *INTERROGACAO,* o analisador interpreta-o como um operador ternário de condição verdadeira e retorna o identificador *T_OP_TERNARY_IF.*
- ({DOIS_PONTOS}): Ao identificar a representação do *DOIS_PONTOS,* o analisador interpreta-o como um operador ternário de condição falsa e retorna o identificador *T_OP_TERNARY_ELSE.*
- ({PONTO_E_VIRGULA}): Ao identificar a representação do *PONTO_E_VIRGULA,* o analisador interpreta-o como um separador de instrução e retorna o identificador *T_PONTO_E_VIRGULA.*
- ({VIRGULA}): Ao identificar a representação do *VIRGULA,* o analisador interpreta-o como um separador e retorna o identificador *T_VIRGULA.*
- ({PARENTESIS_L}): Ao identificar a representação do *PARENTESIS_L,* o analisador interpreta-o como uma abertura de parênteses e retorna o identificador *T_PARENTESIS_ESQ.*
- ({PARENTESIS_R}): Ao identificar a representação do *PARENTESIS_R,* o analisador interpreta-o como um fechamento de parênteses e retorna o identificador *T_PARENTESIS_DIR.*
- ({CHAVE_L}): Ao identificar a representação do *CHAVE_L,* o analisador interpreta-o como uma abertura de chaves e retorna o identificador *T_CHAVES_ESQ.*

- `({CHAVE_R})`: Ao identificar a representação do *CHAVE_R,* o analisador interpreta-o como um fechamento de chaves e retorna o identificador *T_CHAVES_DIR.*
- `({COLCHETE_L})`: Ao identificar a representação do *COLCHETE_L,* o analisador interpreta-o como uma abertura de colchetes e retorna o identificador *T_COLCHETES_ESQ.*
- `({COLCHETE_R})`: Ao identificar a representação do *COLCHETE_R,* o analisador interpreta-o como umfechamento de colchetes e retorna o identificador *T_COLCHETES_DIR.*
- `.`: Ao identificar qualquer caractere que não tenha sido validado por alguma das regras anteriores, o analisador interpreta esse caractere como um caractere desconhecido e retorna o identificador *T_UNKNOWN*.

3.5. Seção de Código do Usuário

Nesta seção, tem-se o código em C que será executado. Inicialmente é definido a função principal *`main`* que faz uma verificação inicial para saber se deve-se analisar um arquivo ou o texto inserido no terminal.

Em seguida é iniciado um laço de análise léxica onde o analisador *yylex()* usa as regras definidas anteriormente para gerar os identificadores únicos em sequência e para cada novo identificador encontrado, uma sequência de impressões divididas por identificador é feita dando a estrutura de cada token:

- `<{lexema}>`: para tokens que representam palavras reservadas ou operadores ou separadores;
- `<id,{posição na tabela de símbolos}>`: para tokens que representam variáveis;
- `<int,{lexema}>`: para tokens que representam números inteiros;
- `<float,{lexema}>`: para tokens que representam números reais;
- `<string,{lexema}>`: para tokens que representam cadeias de caracteres;
- `<char,{lexema}>`: para tokens que representam caracteres literais.

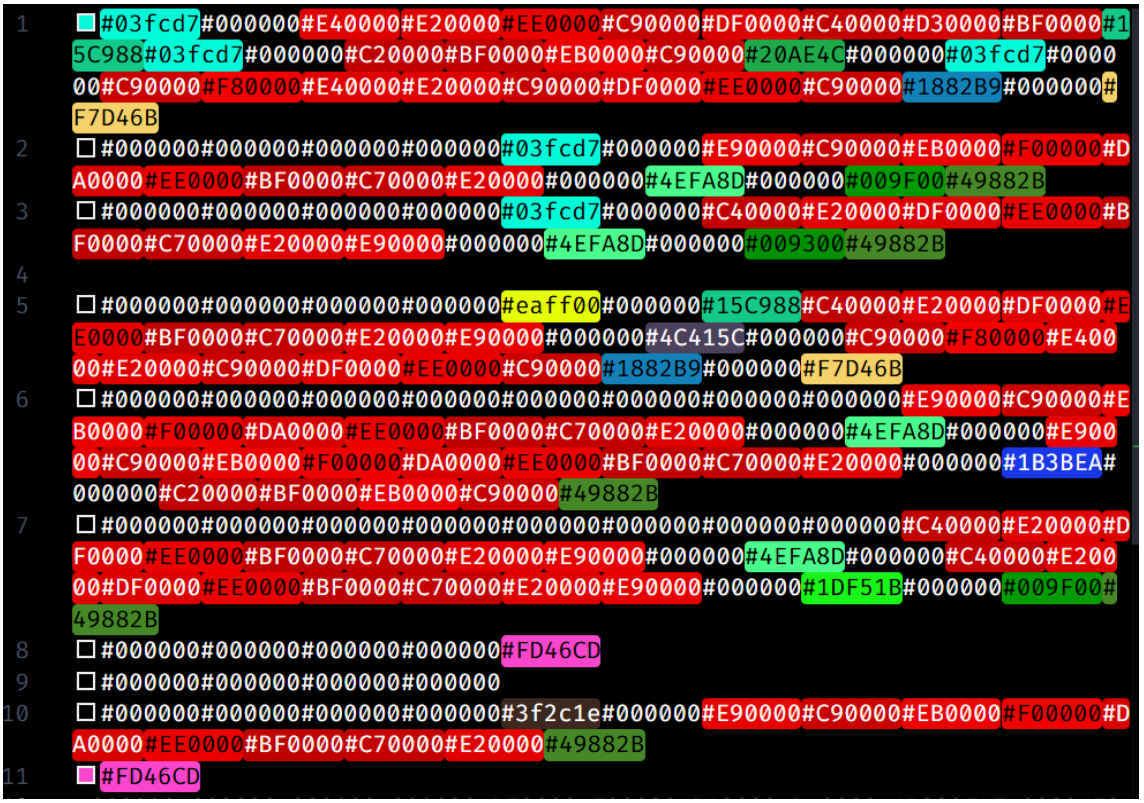
3.6. Tabela de Símbolos

O código flex imprime, em formato de tabela padronizada da linguagem MARKDOWN, a tabela de símbolos com duas colunas: “Posição” e “Identificador”, onde cada linha representa uma variável com a posição sendo um inteiro de 0 a n, onde n - 1 é a quantidade de símbolos e o identificador é o nome dado à variável no código original.

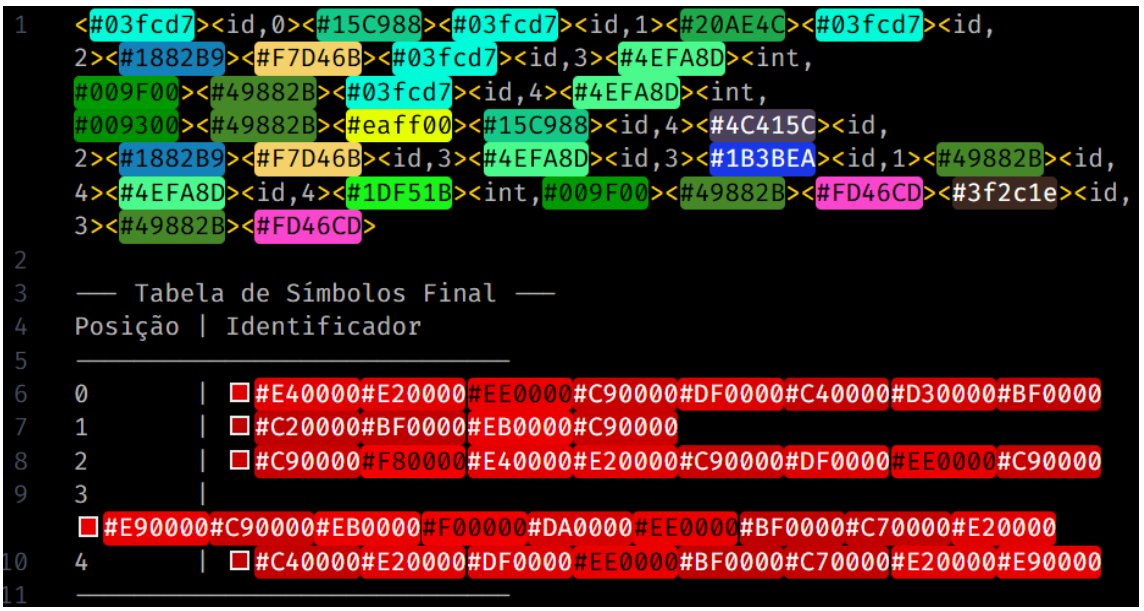
4. Exemplos

4.1. Calculadora de potência

O exemplo de código abaixo implementa uma função que recebe dois parâmetros inteiros $base$ e $expoente$ e retorna o valor da potência desses dois valores $base^{expoente}$.

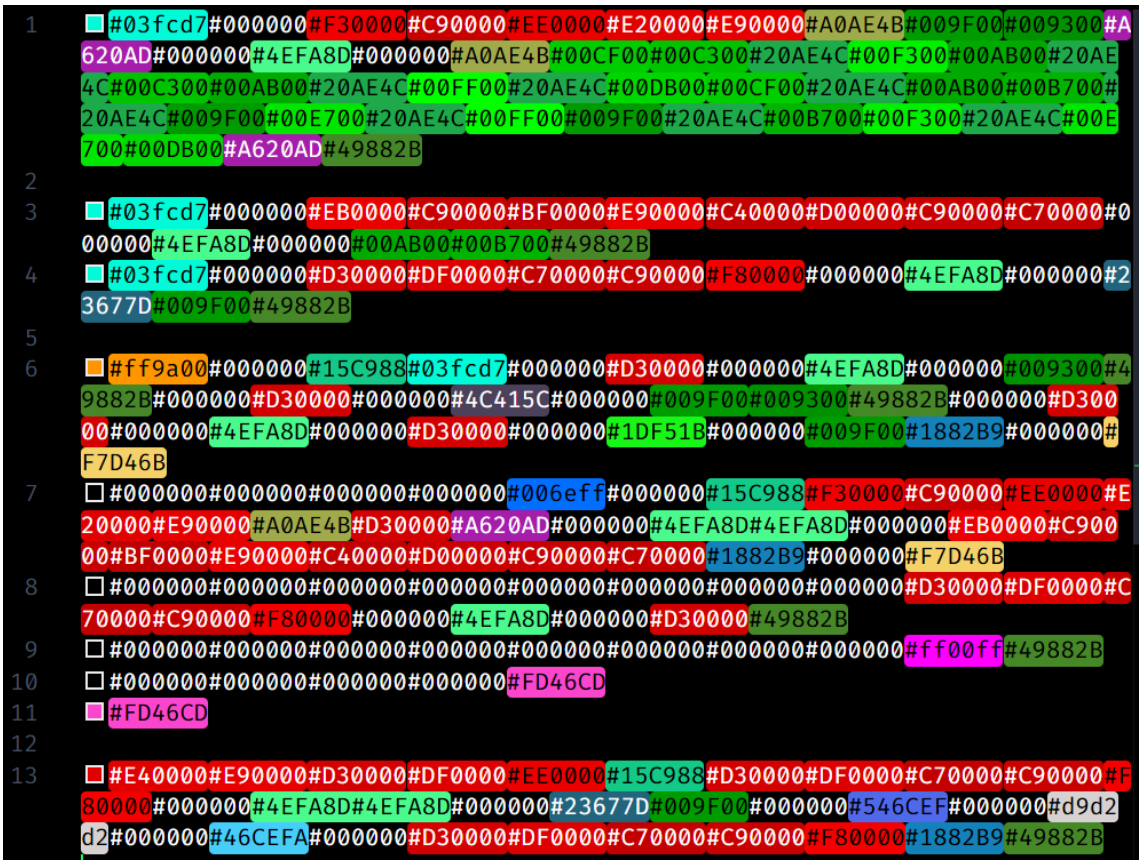


Esta é a saída tokenizada do analisador léxico:



4.2. Busca sequencial em um vetor (com comentários)

O exemplo de código abaixo realiza uma busca sequencial analisando cada registro de um vetor preenchido em tempo de compilação.



Esta é a saída tokenizada do analisador léxico:


<#03fcd7><id,0><#A0AE4B><int,
#009F00#009300><#A620AD><#4EFA8D><#A0AE4B><int,
#00CF00#00C300><#20AE4C><int,#00F300#00AB00><#20AE4C><int,
#00C300#00AB00><#20AE4C><int,#00FF00><#20AE4C><int,
#00DB00#00CF00><#20AE4C><int,#00AB00#00B700><#20AE4C><int,
#009F00#00E700><#20AE4C><int,#00FF00#009F00><#20AE4C><int,
#00B700#00F300><#20AE4C><int,
#00E700#00DB00><#A620AD><#49882B><#03fcd7><id,1><#4EFA8D><int,
#00AB00#00B700><#49882B><#03fcd7><id,2><#4EFA8D><#23677D><int,
#009F00><#49882B><#ff9a00><#15C988><#03fcd7><id,3><#4EFA8D><int,
#009300><#49882B><id,3><#4C415C><int,#009F00#009300><#49882B><id,
3><#4EFA8D><id,3><#1DF51B><int,
#009F00><#1882B9><#F7D46B><#006eff><#15C988><id,0><#A0AE4B><id,
3><#A620AD><#4EFA8D#4EFA8D><id,1><#1882B9><#F7D46B><id,2><#4EFA8D><id,
3><#49882B><#ff00ff><#49882B><#FD46CD><#FD46CD><id,4><#15C988><id,
2><#4EFA8D#4EFA8D><#23677D><int,#009F00><#546CEF><#d9d2d2><#46CEFA><id,
2><#1882B9><#49882B>
── Tabela de Símbolos Final ──
Posição | Identificador
0 | #F30000#C90000#EE0000#E20000#E90000
1 | #EB0000#C90000#BF0000#E90000#C40000#D00000#C90000#C70000
2 | #D30000#DF0000#C70000#C90000#F80000
3 | #D30000
4 | #E40000#E90000#D30000#DF0000#EE0000

4.3. Cálculo da média de Fibonacci

O exemplo abaixo possui duas funções onde uma delas calcula o valor da sequência de fibonacci numa posição fornecida como parâmetro e a outra calcula a média desse valor pela posição.

```
1  #03fcd7#000000#CC0000#D30000#C20000#15C988#03fcd7#000000#DF0000#1882B9#0
   00000#F7D46B
2  #000000#000000#000000#000000#006eff#000000#15C988#DF0000#000000#4EFA8D#4
   EFA8D#000000#009300#1882B9#000000#F7D46B
3  #000000#000000#000000#000000#000000#000000#000000#000000#3f2c1e#000000#0
   09300#49882B
4  #000000#000000#000000#000000#FD46CD#000000#ff0000#000000#006eff#000000#1
   5C988#DF0000#000000#4EFA8D#4EFA8D#000000#009F00#1882B9#000000#F7D46B
5  #000000#000000#000000#000000#000000#000000#000000#000000#3f2c1e#000000#0
   09F00#49882B
6  #000000#000000#000000#000000#FD46CD#000000#ff0000#000000#F7D46B
7  #000000#000000#000000#000000#000000#000000#000000#000000#3f2c1e#000000#C
   C0000#D30000#C20000#15C988#DF0000#000000#23677D#000000#009F00#1882B9#0000
   00#1DF51B#000000#CC0000#D30000#C20000#15C988#DF0000#000000#23677D#000000#
   00AB00#1882B9#49882B
8  #000000#000000#000000#000000#FD46CD
9  #FD46CD
10
11 #ff00c3#000000#CC0000#D30000#C20000#AB930F#DD0000#C90000#C70000#D30000#B
   F0000#15C988#03fcd7#000000#DF0000#1882B9#000000#F7D46B
12 #000000#000000#000000#000000#006eff#000000#15C988#DF0000#000000#4EFA8D#4
   EFA8D#000000#009300#1882B9#000000#F7D46B
13 #000000#000000#000000#000000#000000#000000#000000#000000#3f2c1e#000000#0
   09300#2620AE#009300#49882B
14 #000000#000000#000000#000000#FD46CD
15
16 #000000#000000#000000#000000#3f2c1e#000000#CC0000#D30000#C20000#15C988#D
   F0000#1882B9#000000#28D9DF#000000#DF0000#49882B
17 #FD46CD
18
```

Esta é a saída tokenizada do analisador léxico:

```
1  <#03fcd7><id,0><#15C988><#03fcd7><id,
   1><#1882B9><#F7D46B><#006eff><#15C988><id,1><#4EFA8D#4EFA8D><int,
   #009300><#1882B9><#F7D46B><#3f2c1e><int,
   #009300><#49882B><#FD46CD><#ff0000><#006eff><#15C988><id,
   1><#4EFA8D#4EFA8D><int,#009F00><#1882B9><#F7D46B><#3f2c1e><int,
   #009F00><#49882B><#FD46CD><#ff0000><#F7D46B><#3f2c1e><id,0><#15C988><id,
   1><#23677D><int,#009F00><#1882B9><#1DF51B><id,0><#15C988><id,
   1><#23677D><int,#00AB00><#1882B9><#49882B><#FD46CD><#FD46CD><#ff00c3><id,
   2><#15C988><#03fcd7><id,1><#1882B9><#F7D46B><#006eff><#15C988><id,
   1><#4EFA8D#4EFA8D><int,#009300><#1882B9><#F7D46B><#3f2c1e><float,
   #009300#2620AE#009300><#49882B><#FD46CD><#3f2c1e><id,0><#15C988><id,
   1><#1882B9><#28D9DF><id,1><#49882B><#FD46CD>
2
3  ── Tabela de Símbolos Final ──
4  Posição | Identificador
5  ──────────────────────────────
6  0       | #CC0000#D30000#C20000
7  1       | #DF0000
8  2       |
   #CC0000#D30000#C20000#AB930F#DD0000#C90000#C70000#D30000#BF0000
9  ──────────────────────────────
```

5. Diagrama de transição

Estudo dirigido 7: "*[...] as expressões regulares definidas em genericamente e em um arquivo Flex (.l) são modeladas por diagramas de transição, que são a base conceitual de qualquer analisador léxico.*"

5.1. Espaços em branco e comentários

A regra [\t]+ reconhece uma ou mais tabulações consecutivas no código-fonte. Como descrito no diagrama 1, o autômato parte do estado inicial A(0) e, ao ler \t, transita para o próximo estado, permanecendo nele enquanto houver tabulações. Quando a sequência termina, vai para o estado *T_IGNORE*, descartando a entrada sem gerar token. Isso garante que tabulações sejam apenas consumidas e ignoradas durante a análise léxica.

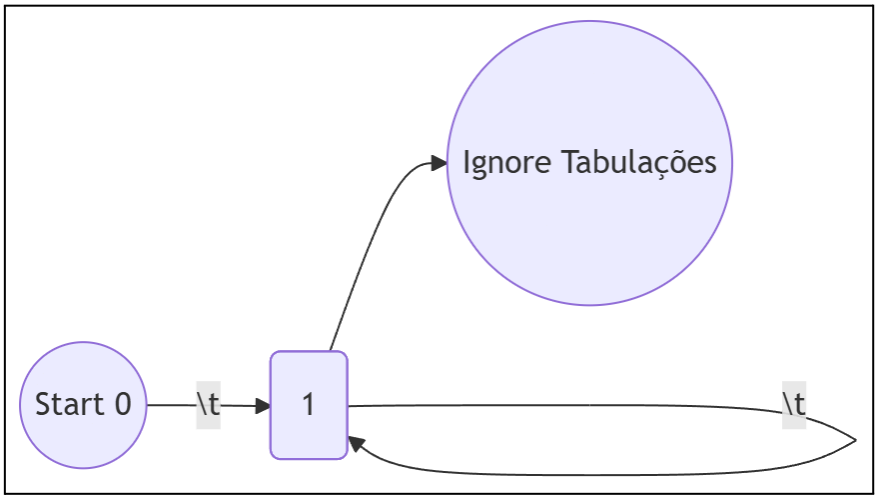


Diagrama 1. Tabulações

A regra `({ESPACO})+` reconhece uma ou mais ocorrências de espaços no código-fonte. Como mostrado no diagrama 2, o autômato inicia em `A(0)`, lê a sequência `#000000` que representa o caractere espaço e percorre os estados 2 a 8 consumindo cada parte da cor. Ao final, retorna ao segundo caso haja mais espaços ou transita para `Ignore Espaços`, descartando a entrada sem gerar token. Assim como as tabulações, espaços são apenas consumidos e ignorados na análise léxica.

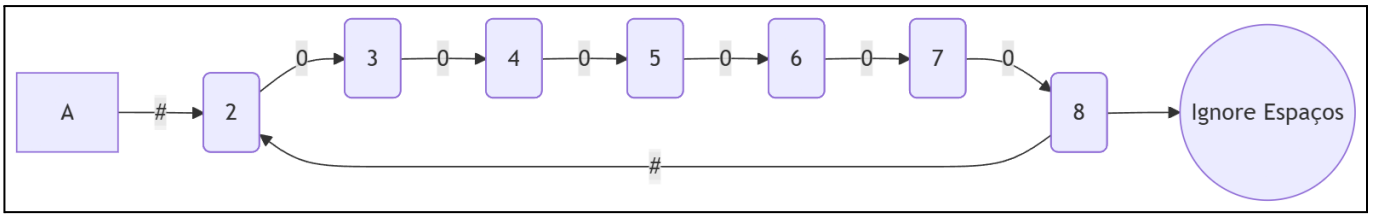


Diagrama 2. Espaços

A regra `\n` reconhece o caractere de nova linha. Assim como no diagrama 3, o autômato parte do estado inicial `A(0)` e, ao ler `\n`, transita diretamente para o próximo estado, onde a entrada é consumida e ignorada. Nenhum token é gerado, servindo apenas para avançar a leitura e atualizar o número da linha automaticamente.

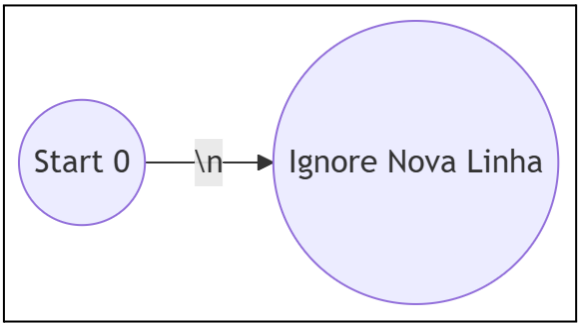


Diagrama 3. Nova Linha

A regra `{COMMENT_LINE}[^\n]*` reconhece comentários de linha iniciados pela cor `#014100` e seguidos por qualquer sequência de caracteres até a quebra de linha. Como no diagrama 4, o autômato começa em `A(0)` e percorre os estados 1 a 7 ao consumir o padrão da cor. Em `B`, ele lê qualquer caractere diferente de `\n`, permanecendo nesse estado até encontrar uma nova linha. Ao ler `\n`, transita para o estado de `Ignore Comentário de Linha`, onde a entrada é descartada sem gerar token, ignorando completamente o comentário.

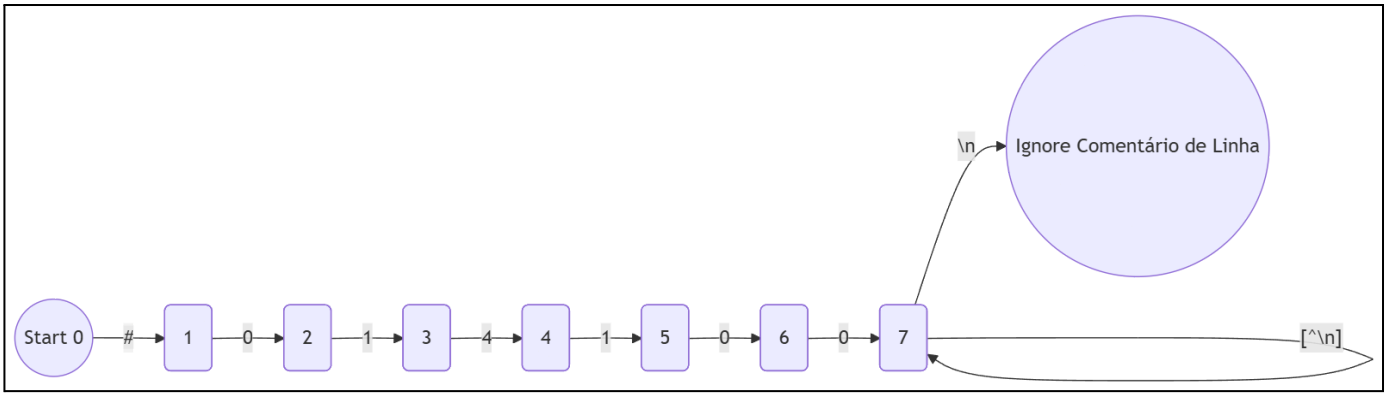


Diagrama 4. Comentário Linha

A regra `{COMMENT_BLOCK}([^#]|(#+[^f])|(#+f[^0])|(#+f0[^b]))*{COMMENT_BLOCK}` reconhece comentários de bloco delimitados pela sequência `#f0b`. Como no diagrama 5, o autômato parte de `Start 0` e percorre os estados 1 a 4 ao consumir a sequência inicial. No próximo estado, ele aceita qualquer conteúdo interno que não corresponda ao padrão de fechamento, mantendo-se nesse estado. Quando encontra novamente a sequência `#f0b`,

transita pelos estados 5 a 7 e finaliza, descartando o trecho sem gerar token. Assim, todo o comentário de bloco é ignorado na análise léxica.

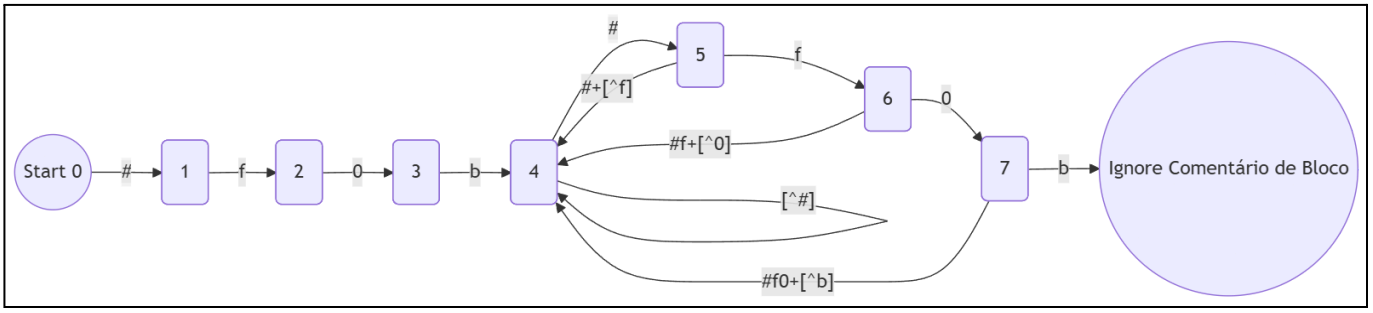


Diagrama 5. Comentário Bloco

5.2. Palavras-Chave

As palavras-chave dessa linguagem são reconhecidas por meio de sequências específicas hexadecimais, cada uma associada a um token reservado. Nos diagramas 6 até 12, todas seguem o mesmo padrão: o autômato inicia no estado `Start 0`, consome o caractere `#` e percorre uma sequência de transições que correspondem aos dígitos hexadecimais da cor definida para a palavra-chave. Ao final da leitura, chega a um estado de aceitação que retorna o token correspondente (`T_IF`, `T_ELSE`, `T_WHILE`, `T_RETURN`, `T_FOR`, `T_VOID` ou `T_BREAK`). Assim, cada palavra-chave é identificada de forma direta e única a partir de seu código de cor, como apresentado na tabela 1.

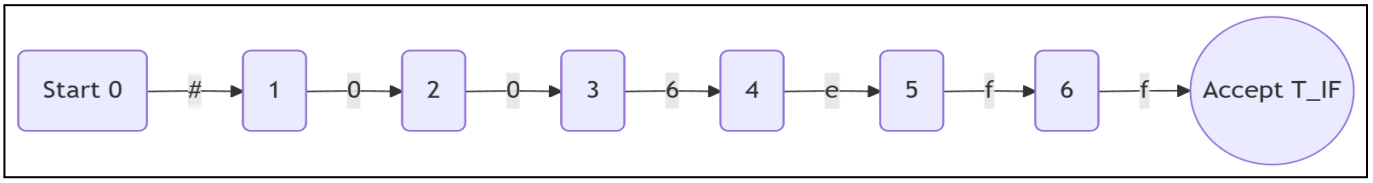


Diagrama 6. If

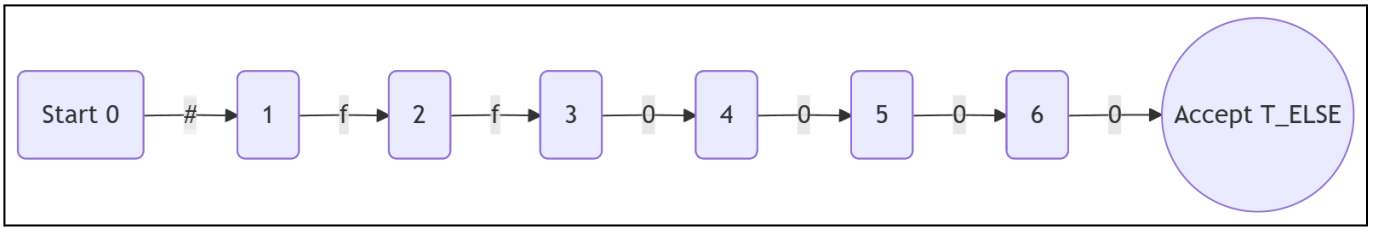


Diagrama 7. Else

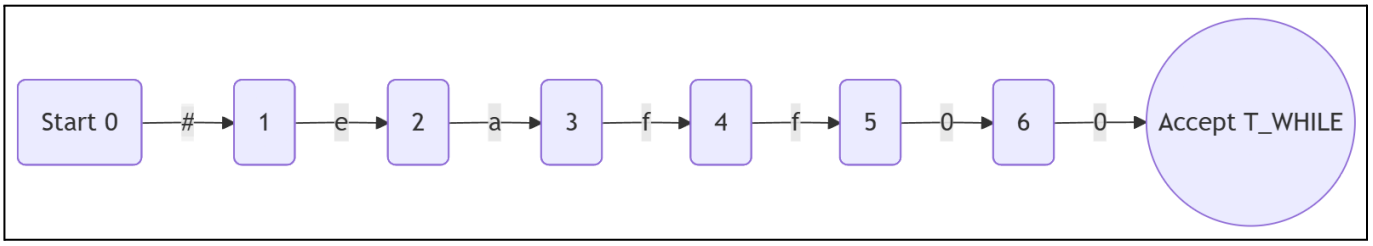


Diagrama 8. While

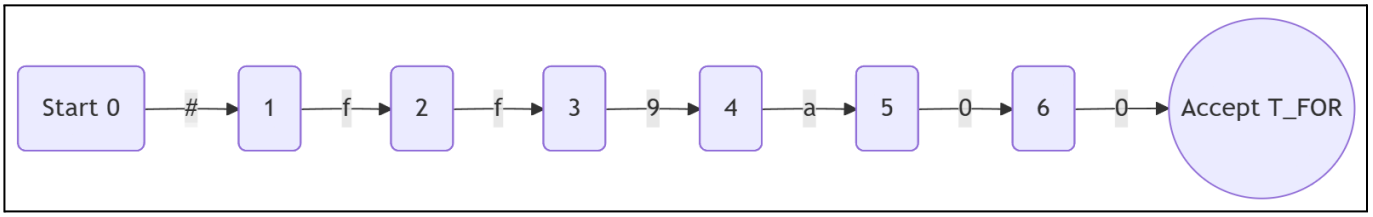


Diagrama 9. For

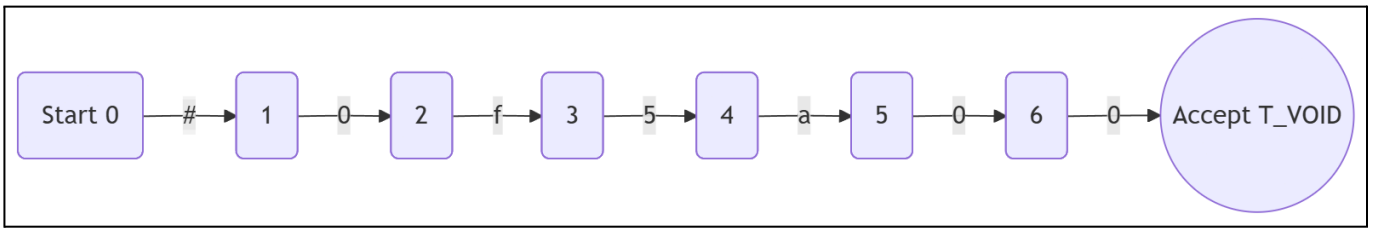


Diagrama 10. Void

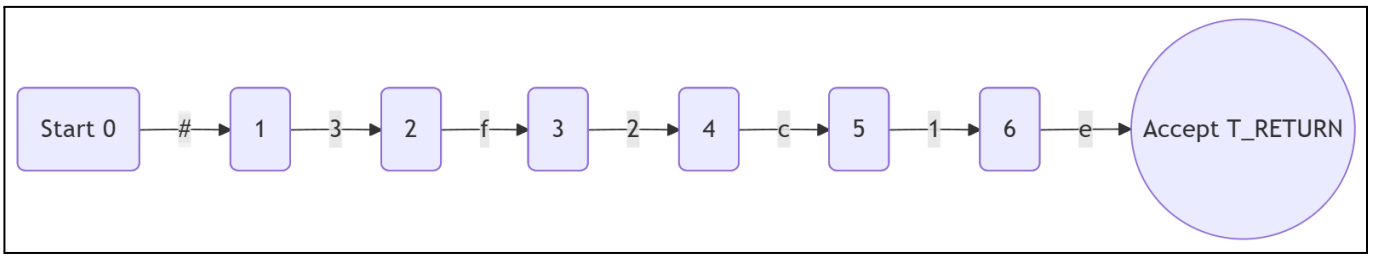


Diagrama 11. Return

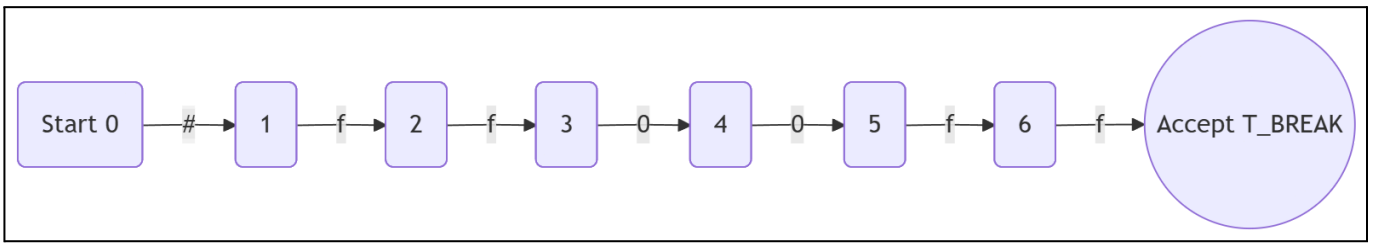


Diagrama 12. Break

5.3. Tipos de dados

Os tipos de dados também são reconhecidos por sequências fixas de cores hexadecimais, cada uma vinculada a um token específico. O funcionamento do autômato é idêntico ao das palavras-chave: parte do estado inicial `Start 0`, consome o caractere # e percorre os estados correspondentes aos dígitos da cor. Ao final da sequência, alcança o estado de aceitação que retorna o token apropriado: `T_INT_KEYWORD`, `T_FLOAT_KEYWORD`, `T_CHAR_KEYWORD`, `T_BOOL` ou `T_STR`. Dessa forma, cada tipo de dado é identificado de forma precisa e única durante a análise léxica.

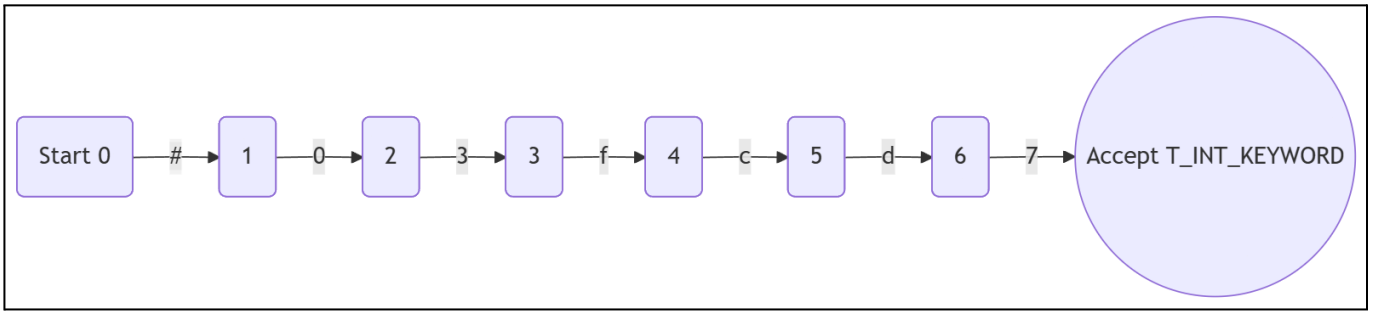


Diagrama 13. Inteiro

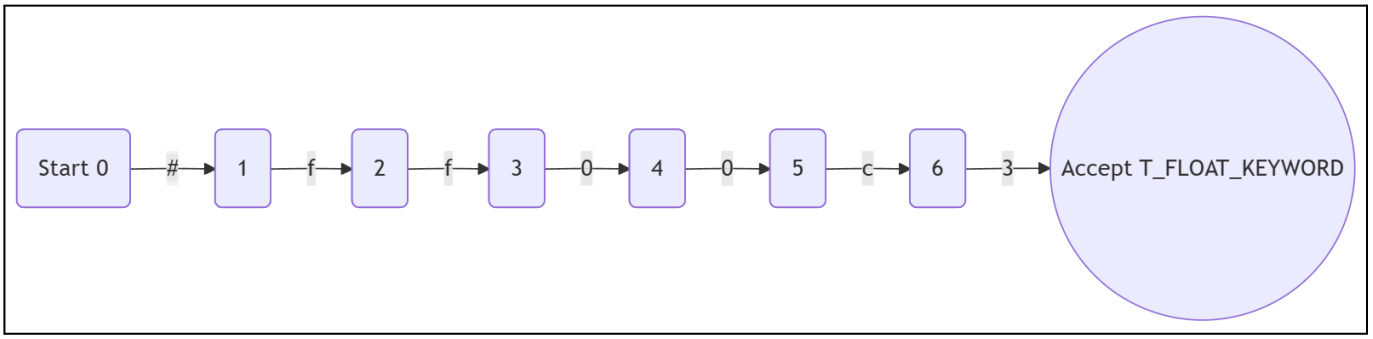


Diagrama 14. Float

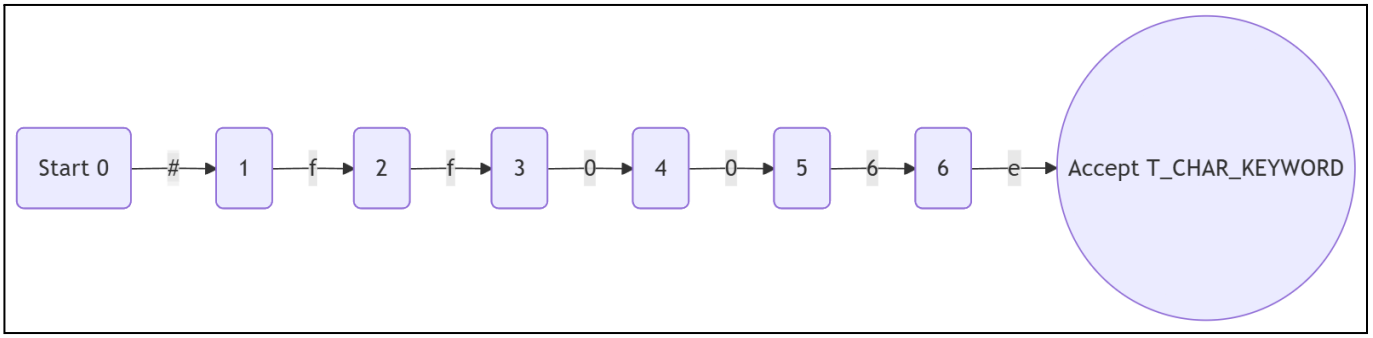


Diagrama 15. Char

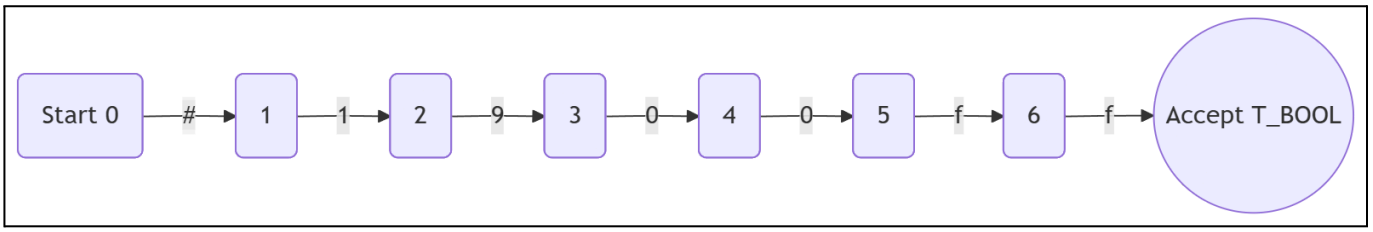


Diagrama 16. Bool

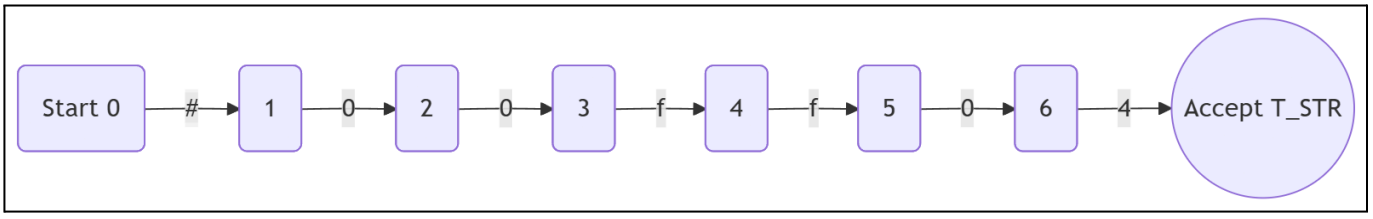


Diagrama 17. String

5.4. Literais especiais

Os literais especiais `true`, `false` e `null` são reconhecidos por cores hexadecimais fixas da mesma forma que palavras-chave e tipos de dados. O autômato inicia no estado `Start 0`,

consome o caractere # e avança por estados que correspondem aos dígitos hexadecimais da cor específica. Ao final, alcança um estado de aceitação que retorna o token correspondente: `T_TRUE`, `T_FALSE` ou `T_NULL`. Essa abordagem garante o reconhecimento direto e sem ambiguidade desses valores constantes durante a análise léxica.

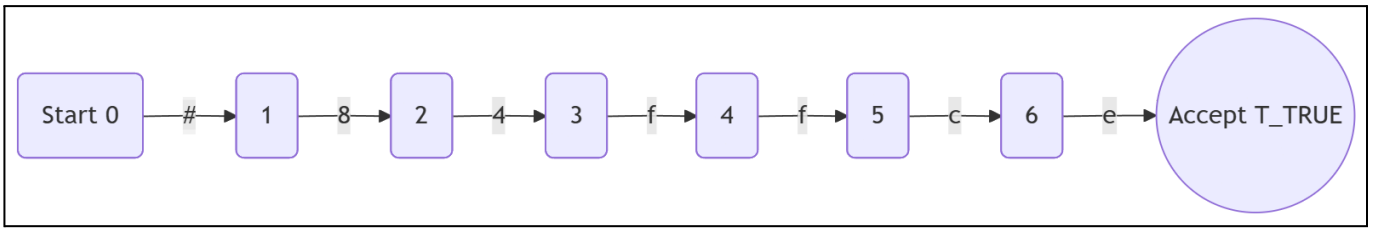


Diagrama 18. True

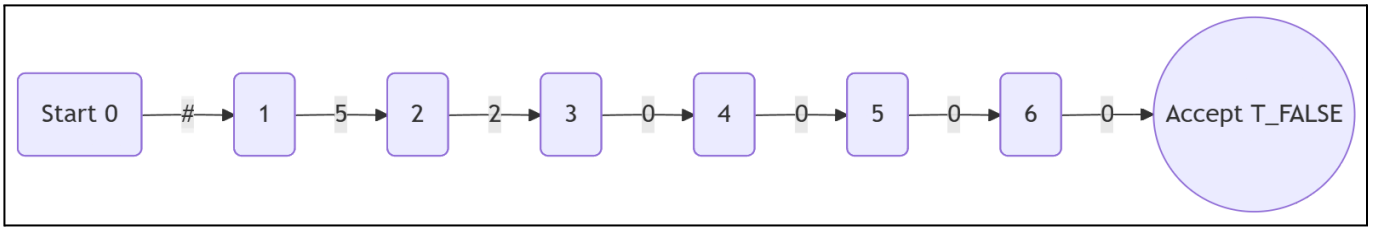


Diagrama 19. False

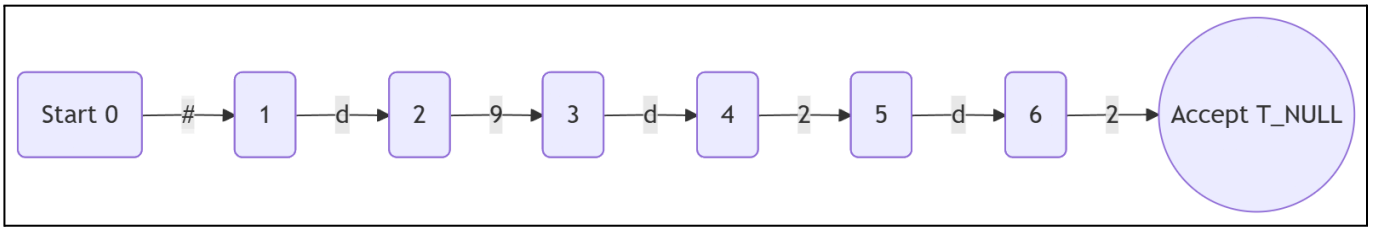


Diagrama 20. Null

5.5. Literais numéricos e strings

Nesse caso, o reconhecimento de números segue uma lógica diferente das palavras-chave e símbolos fixos, pois envolve **padrões variáveis**. O autômato parte do estado inicial `Start 0` e consome um ou mais dígitos, indo para `B(1)`. A partir daí, se a entrada terminar, o token `T_INTEGER` é aceito. Caso contrário, se for encontrado um ponto decimal (`#2620AE`), o autômato percorre os estados 1 a 8 correspondentes à cor do ponto e, em seguida, lê mais dígitos. Ao consumir pelo menos um dígito após o ponto, chega ao estado `D`, aceitando o token `T_FLOAT`. Assim, a distinção entre inteiros e números de ponto flutuante depende da presença ou não do ponto decimal após a sequência inicial de dígitos.

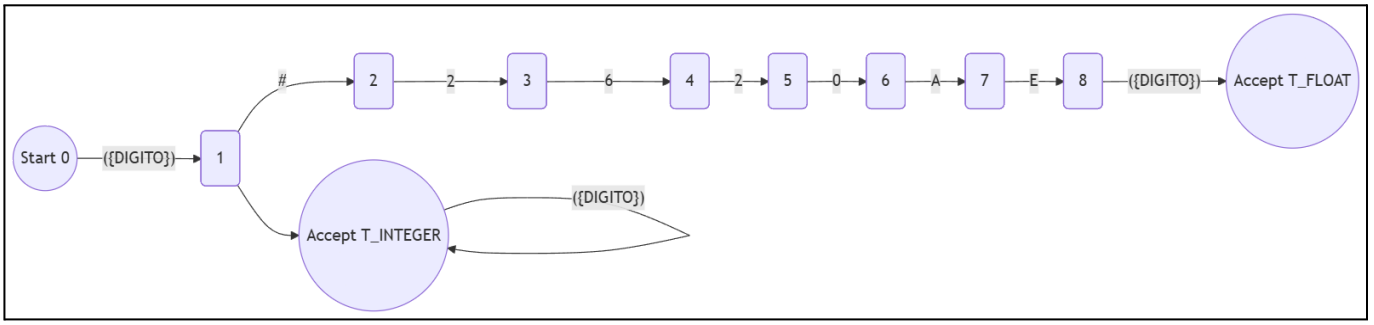


Diagrama 21. Literal numérico

O reconhecimento de strings utiliza um padrão delimitado por aspas. O autômato inicia no estado `Start 0` e, ao ler o caractere **aspas de abertura**, transita para o `próximo estado`. Nesse estado, consome livremente qualquer sequência de caracteres permitidos (`ASCII_SEM_ASPAS`), permanecendo em `B`. Quando encontra a **aspas de fechamento**, transita para o `último estado`, aceitando o token `T_STRING`. Assim, todo o conteúdo entre as aspas é reconhecido como uma única string literal.

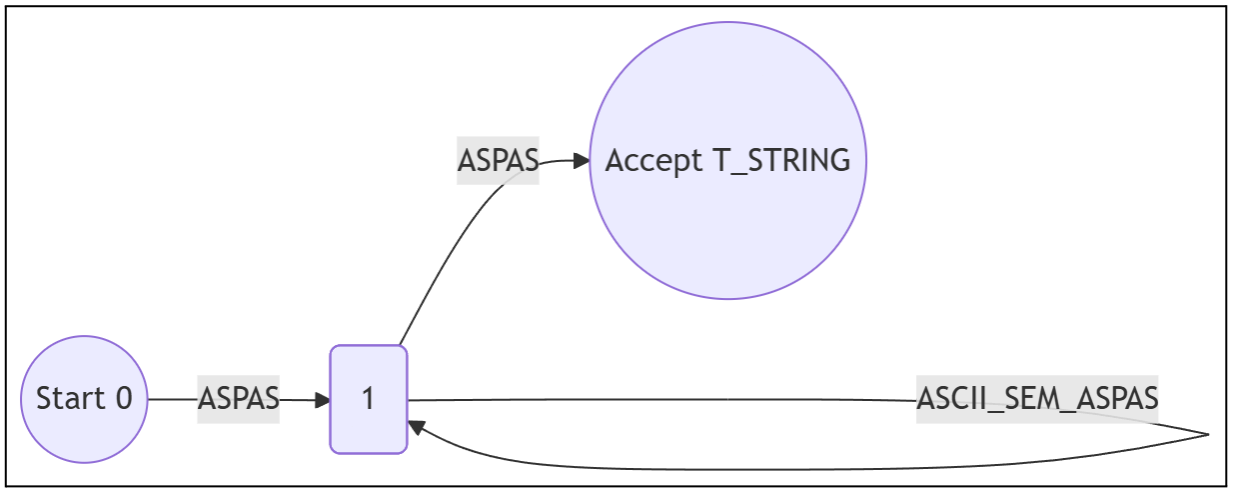


Diagrama 22. Literal string

O reconhecimento de literais de caractere segue uma lógica delimitada por aspas simples. O autômato parte do estado `Start 0` e, ao ler a **aspa de abertura**, vai para `B(1)`. Em seguida, pode consumir diretamente um caractere permitido (`ASCII_SEM_ASPA`), indo para `C(2)`, ou uma **barra invertida**, indo para `D(3)`, que permite o reconhecimento de sequências de escape (`MINUSCULO`, `ASPA` ou `CONTRA_BARRA`). Após reconhecer o caractere ou a sequência de escape, transita para `C`. Por fim, ao ler a **aspa de fechamento**, o autômato vai para o `último estado`, aceitando o token `T_LITERAL_CHAR`. Desse modo, reconhece tanto caracteres simples quanto caracteres com escape.

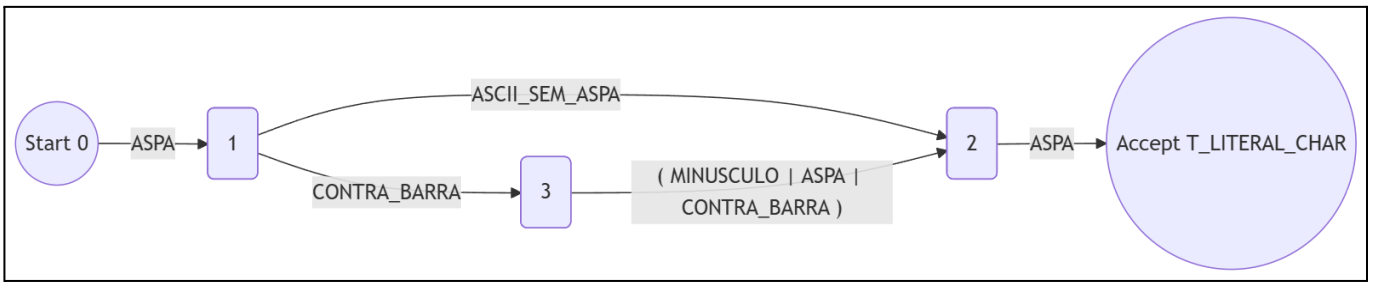


Diagrama 23. Literal Char

5.6. Operadores

5.6.1. Relacionais e atribuição

- `T_OP_MENOR` e `T_OP_MENOR_IGUAL`

A análise do operador “menor ou igual” (<=) e do operador “menor que” (<) começa no estado inicial, avançando ao reconhecer o caractere associado ao símbolo de menor. O autômato percorre uma sequência de transições para confirmar a presença do caractere de igual (=), aceitando o token `T_OP_MENOR_IGUAL` caso essa sequência seja completada. Se o caractere de igual não for encontrado após o símbolo de menor, o autômato aceita imediatamente o token `T_OP_MENOR`. Essa lógica garante que o analisador reconheça corretamente os dois operadores distintos, priorizando a forma composta (<=) sempre que possível.

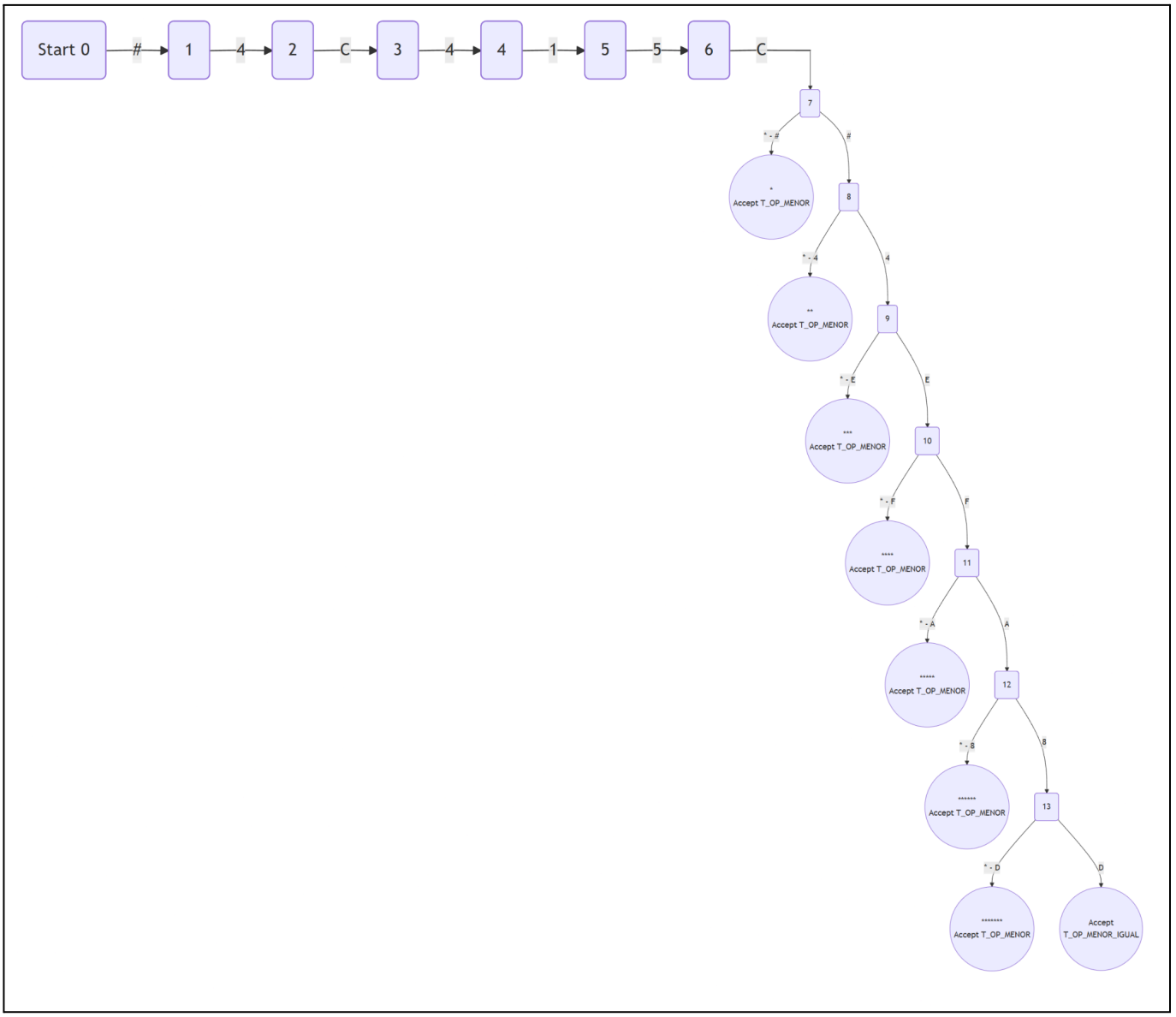


Diagrama 24. Menor e Menor Igual

- `T_OP_MAIOR` e `T_OP_MAIOR_IGUAL`

A análise do operador “maior ou igual” (>=) e do operador “maior que” (>) inicia no estado inicial ao reconhecer o caractere associado ao símbolo de maior. O autômato segue uma série de transições para identificar a presença do caractere de igual (=). Caso a sequência completa seja detectada, o token `T_OP_MAIOR_IGUAL` é aceito. Se o caractere de igual não

ocorrer após o símbolo de maior, o autômato aceita imediatamente o token `T_OP_MAIOR`. Essa abordagem permite que o analisador diferencie corretamente entre os dois operadores, priorizando o reconhecimento do operador composto (>=) sempre que aplicável.

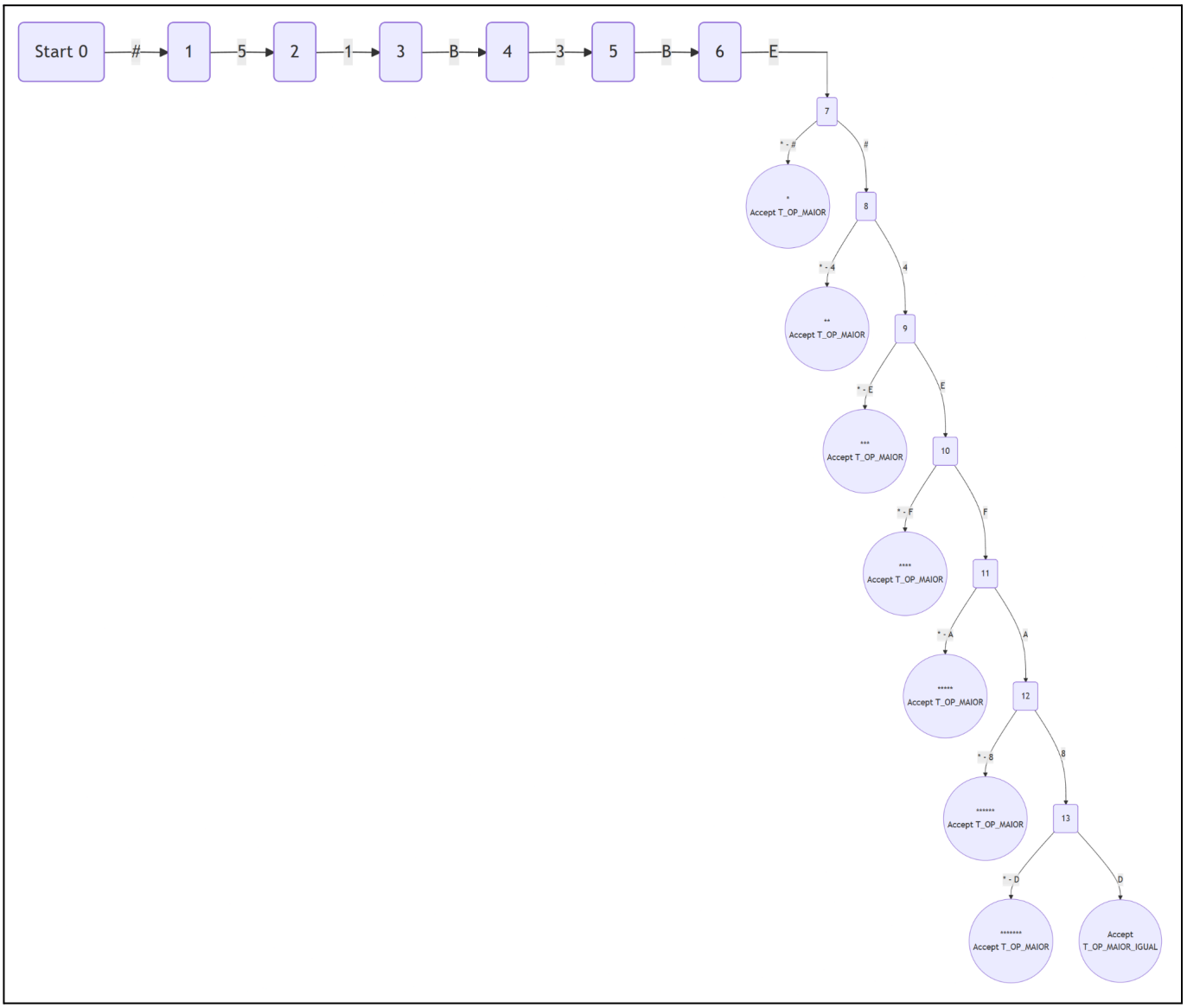


Diagrama 25. Maior e Maior Igual

- `T_OP_ATRIBUICAO e T_OP_IGUALDADE`

A identificação dos operadores de igualdade (==) e de atribuição (=) começa no estado inicial ao reconhecer o caractere associado ao símbolo de igual. O autômato verifica se há um segundo caractere igual em sequência. Se encontrado, o token `T_OP_IGUALDADE` é aceito, representando o operador de comparação. Caso contrário, se o caractere igual aparecer isoladamente, o token `T_OP_ATRIBUICAO` é aceito, indicando o operador de atribuição. Essa

estrutura garante o reconhecimento preciso dos dois operadores, priorizando o operador composto (==) quando presente.

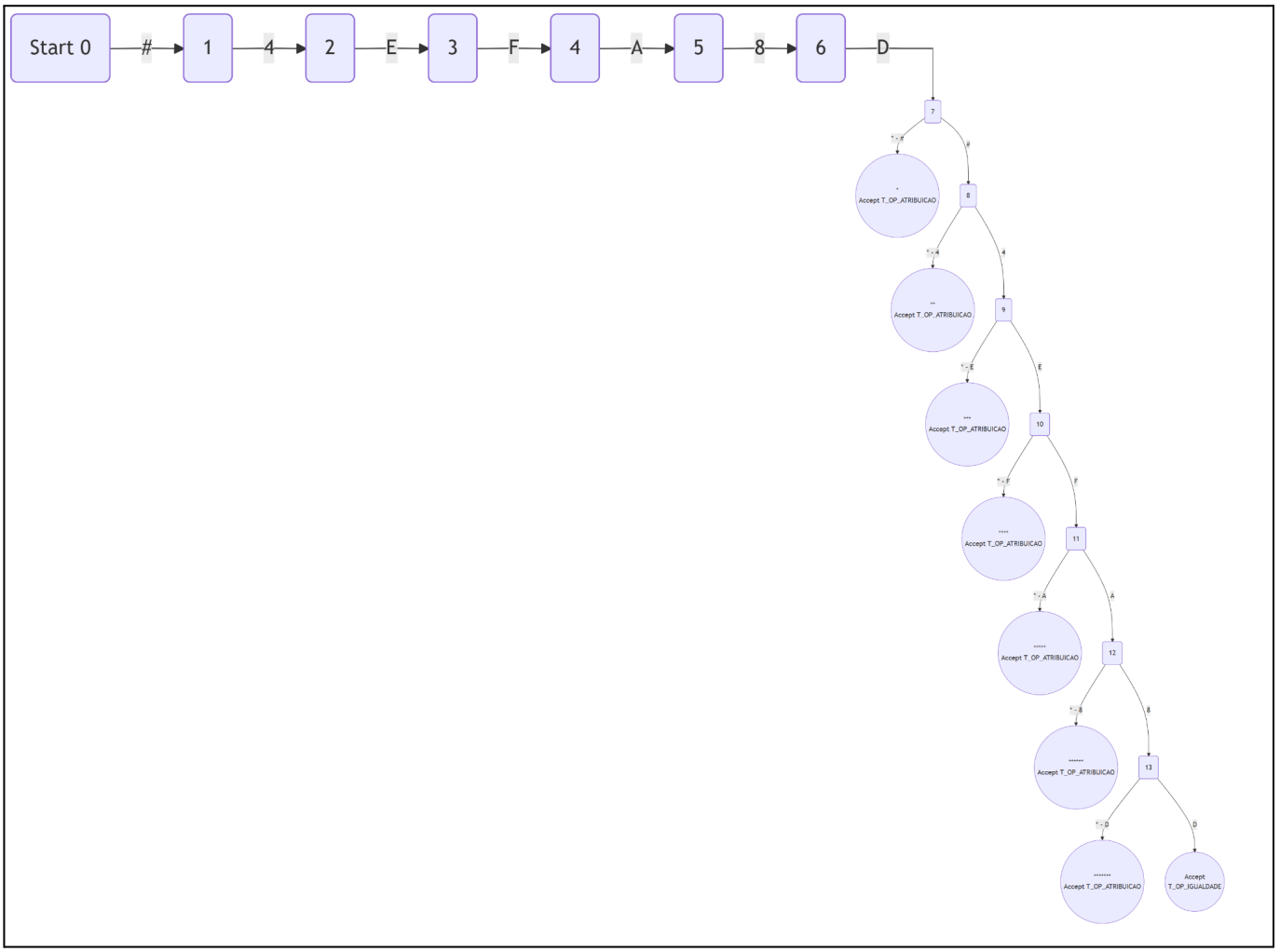


Diagrama 26. Atribuição e igualdade

5.6.2. Aritméticos

Os operadores aritméticos seguem o mesmo padrão de reconhecimento baseado em cores hexadecimais fixas. O autômato começa no estado `Start 0`, consome # e percorre uma sequência de estados correspondentes aos dígitos da cor de cada operador. Ao final, alcança um estado de aceitação que retorna o token apropriado: `T_OP_SOMA`, `T_OP_SUB`, `T_OP_MULT` ou `T_OP_DIV`. Esse mecanismo permite identificar de forma direta e eficiente os operadores aritméticos na análise léxica.

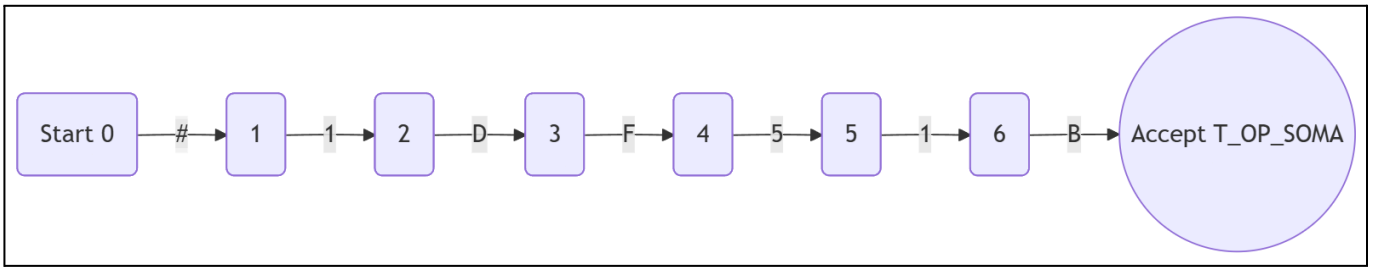


Diagrama 27. Soma

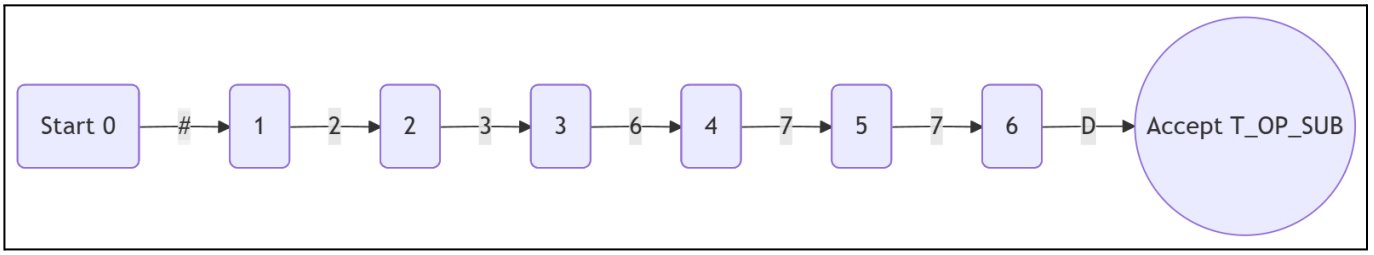


Diagrama 28. Subtração

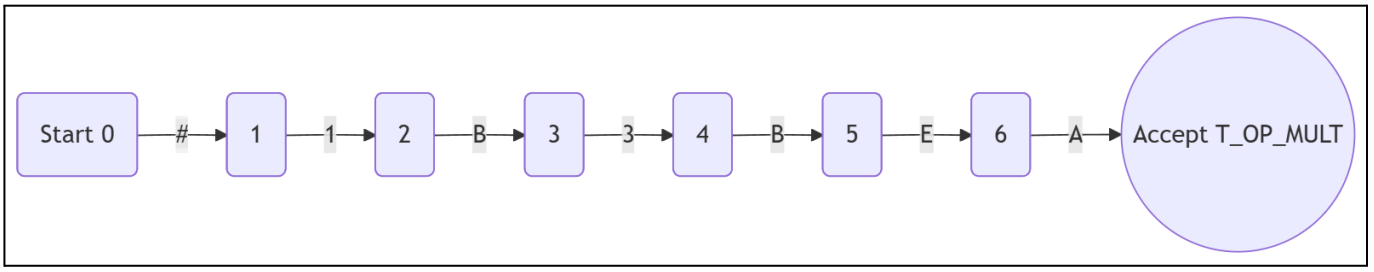


Diagrama 29. Multiplicação

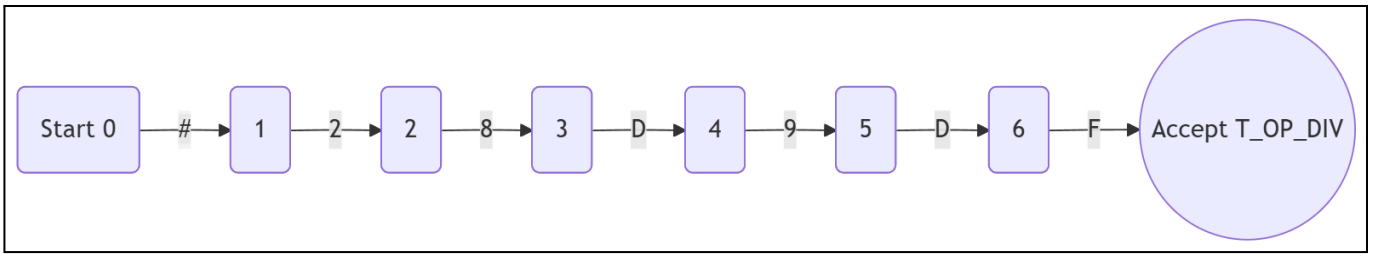


Diagrama 30. Divisão

5.6.3. Ternário

Os operadores ternários seguem o mesmo padrão de reconhecimento dos demais símbolos. O autômato parte do estado `Start 0`, consome o caractere `#` e percorre os estados correspondentes aos dígitos hexadecimais da cor associada a cada operador. Ao final, atinge um estado de aceitação que retorna o token `T_OP_TERNARY_IF` para o `?` ou `T_OP_TERNARY_ELSE` para o `:`. Assim, o analisador identifica de forma direta e única os operadores do ternário durante a análise léxica.

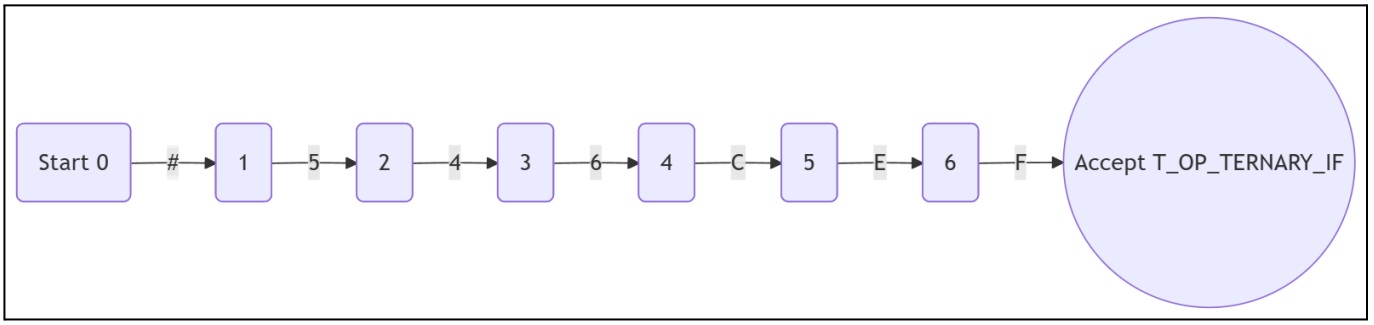


Diagrama 31. Ternário If

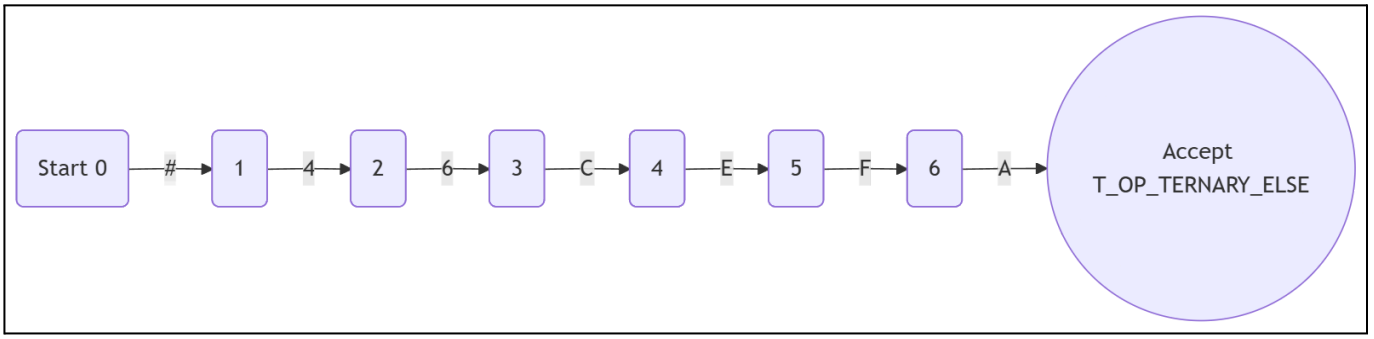


Diagrama 32. Ternário Else

5.7. Pontuação e agrupamento

Os símbolos de pontuação e agrupamento também são reconhecidos a partir de cores hexadecimais fixas. O autômato inicia no estado `Start 0`, consome # e avança pelos estados que representam os dígitos hexadecimais de cada símbolo. Ao final, alcança um estado de aceitação que retorna o token correspondente, como `T_PONTO_VIRGULA`, `T_VIRGULA`, `T_PARENTESES_ESQ`, `T_PARENTESES_DIR`, `T_COLCHETES_ESQ`, `T_COLCHETES_DIR`, `T_CHAVES_ESQ` ou `T_CHAVES_DIR`. Este padrão garante o reconhecimento simples e direto dos delimitadores e sinais de pontuação na análise léxica.

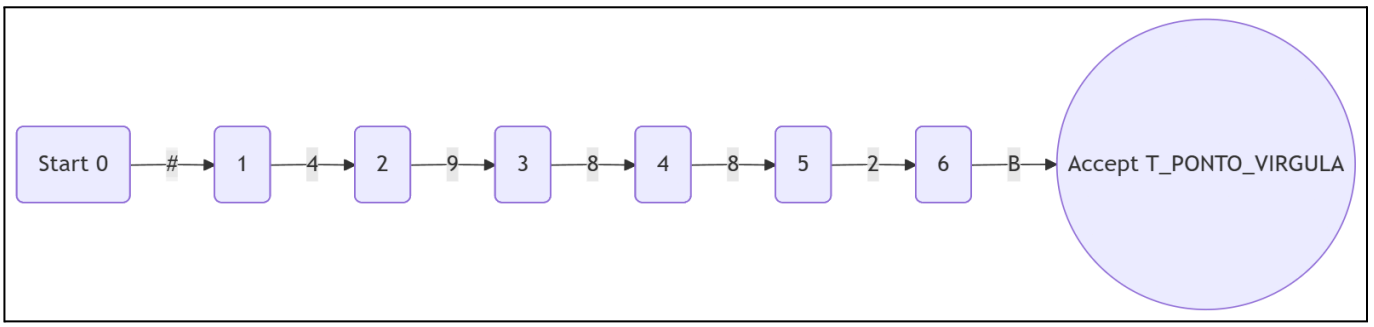


Diagrama 33. Ponto e vírgula

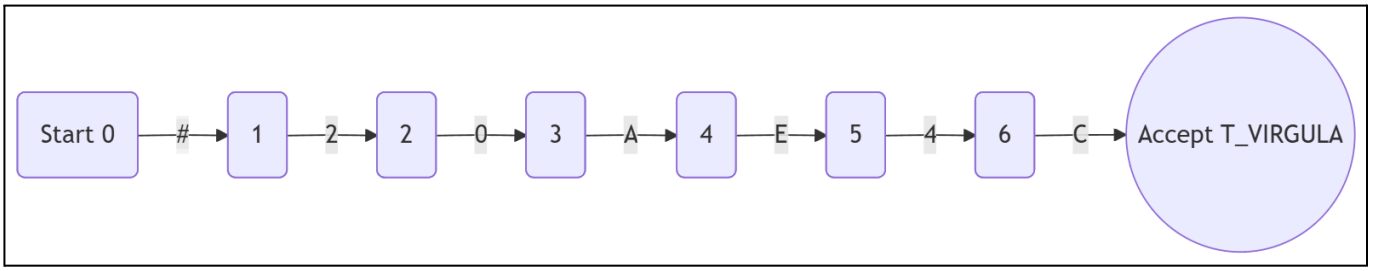


Diagrama 34. Vírgula

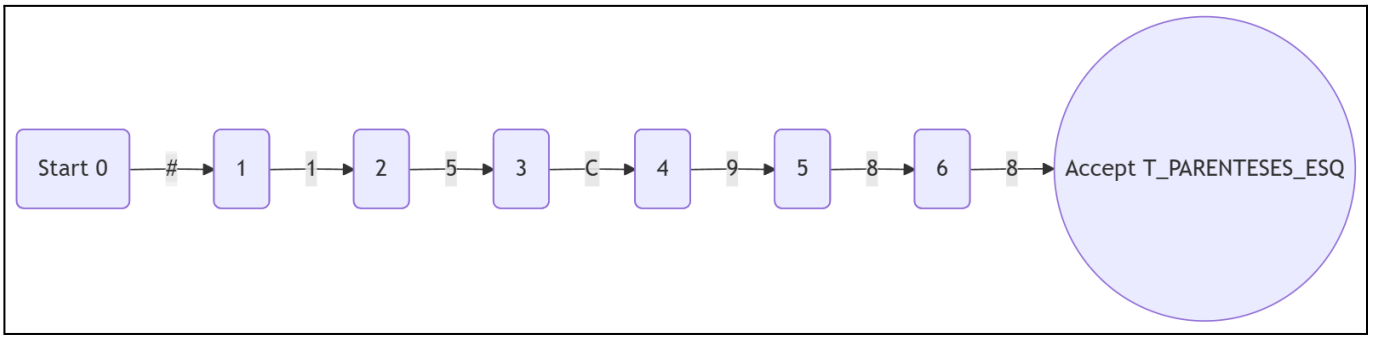


Diagrama 35. Parêntese Esquerdo

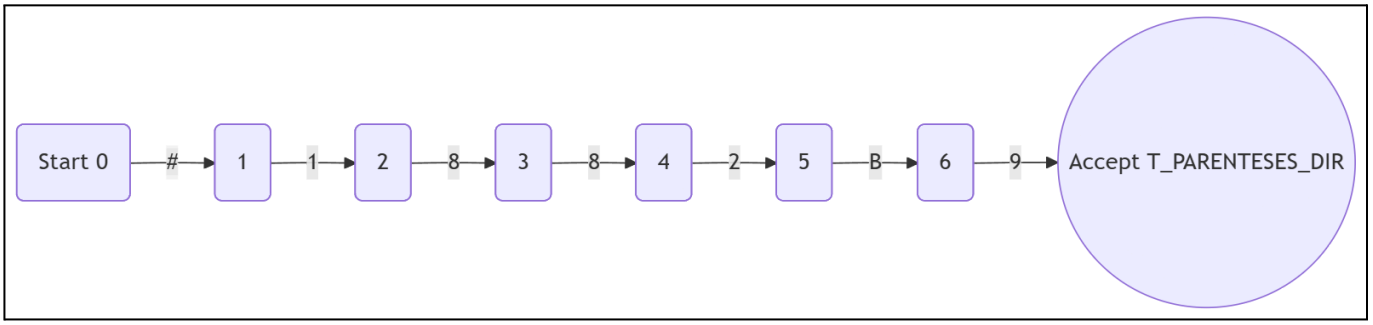


Diagrama 36. Parêntese Direito

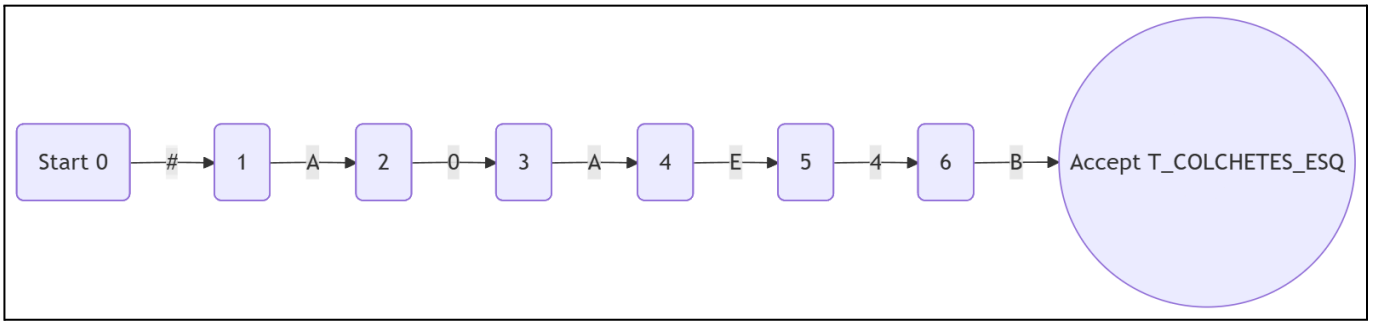


Diagrama 37. Colchete Esquerdo

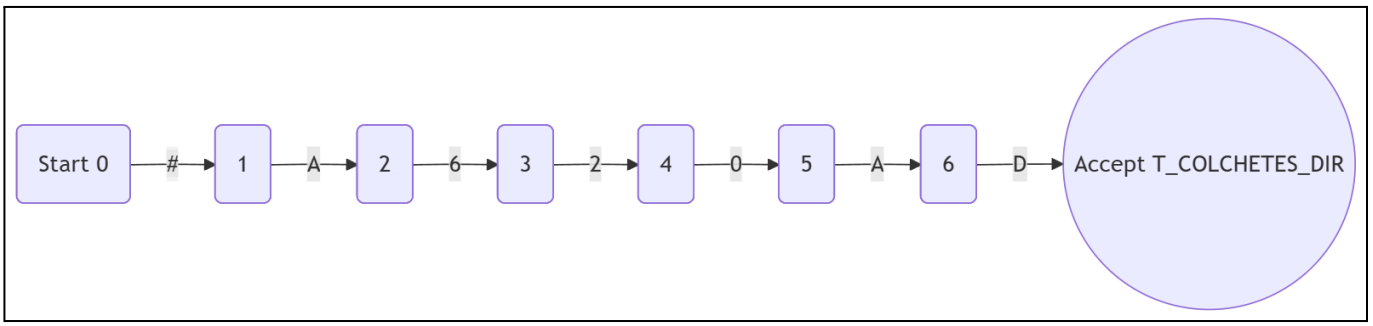


Diagrama 38. Colchete Direito

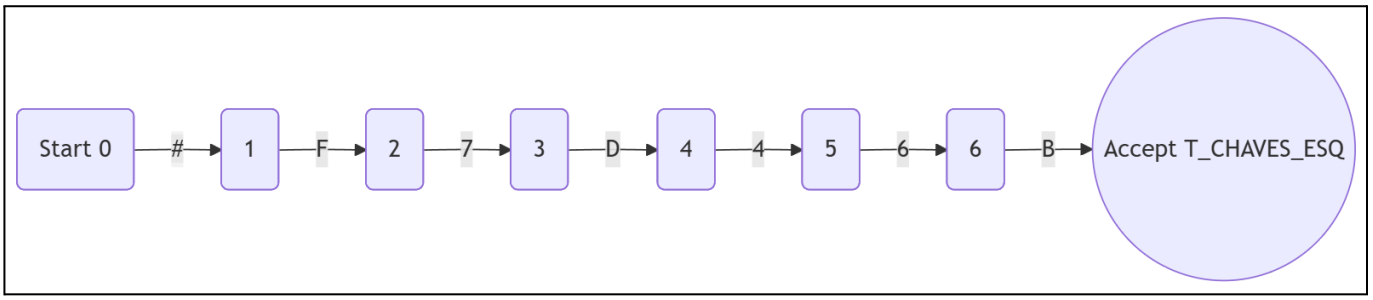


Diagrama 39. Chave Esquerda

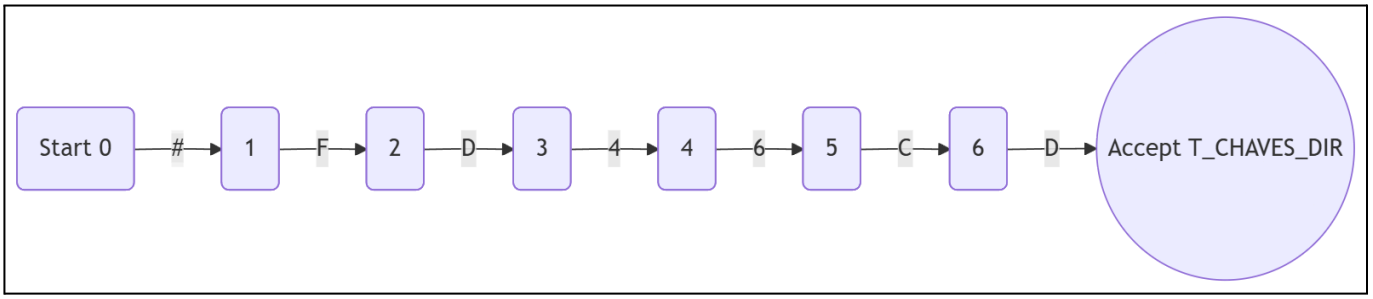


Diagrama 40. Chave Direita

5.8. Identificadores

O reconhecimento de identificadores inicia no estado `Start 0` e avança para o próximo estado ao ler uma letra ou underline. Nesse estado, o autômato aceita uma sequência contínua de letras, underlines ou dígitos, permanecendo nele. Quando a sequência termina, o autômato aceita o token `T_ID`. Esse padrão garante que os identificadores comecem com uma letra ou underline, seguidos por qualquer combinação desses caracteres ou dígitos.

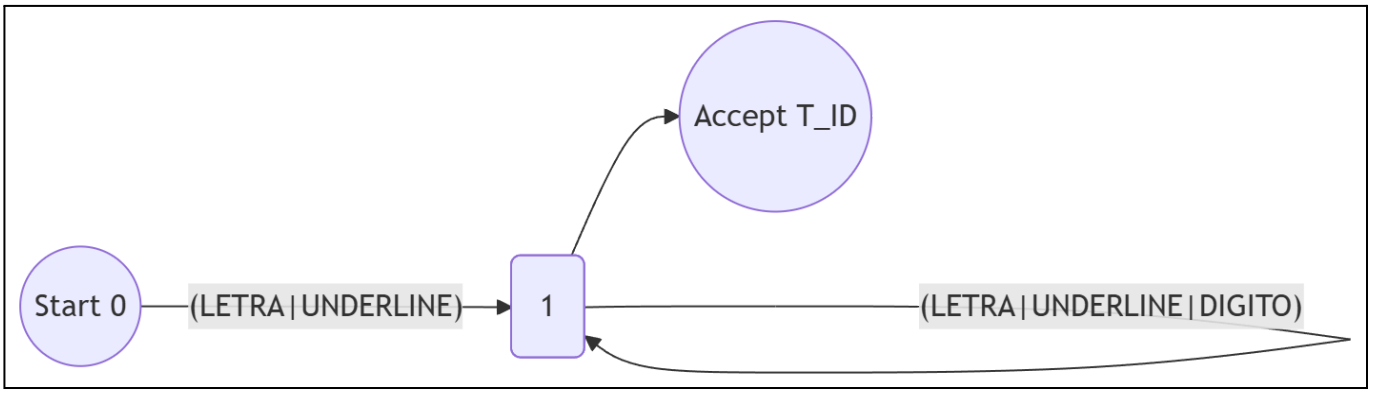


Diagrama 41. Identificador

6. Conclusão

É factível inferir que o #C0DEFF representa um marco conceitual na forma de estruturar linguagens de programação. Sua construção fundamenta-se na utilização de códigos hexadecimais como elemento central para representação de palavras-chave, caracteres e dígitos, atribuindo a cada componente uma identidade cromática própria. Essa abordagem amplia a noção tradicional de sintaxe, permitindo que a cor assuma um papel dentro do código.

Os exemplos e diagramas apresentados ao longo do trabalho desempenham papel fundamental na compreensão aprofundada do funcionamento do #C0DEFF. Por meio deles, é possível visualizar de forma clara como os princípios cromáticos e as regras léxicas se aplicam na prática, permitindo observar a correspondência direta entre cada símbolo e sua representação hexadecimal. Os diagramas de transição, em especial, evidenciam o fluxo de reconhecimento dos tokens e a interação entre os estados do analisador léxico, facilitando a assimilação das estruturas internas da linguagem.

Por fim, com o apoio do analisador léxico, o #C0DEFF estabelece as bases para um processamento consistente e determinístico, no qual cada regra de identificação e conversão cromática. Esse componente garante a operacionalidade da linguagem, permitindo que seus programas sejam construídos e interpretados de maneira estável. Assim, o #C0DEFF se consolida como um modelo conceitual que transcende a representação textual tradicional, introduzindo a dimensão visual como parte integrante da lógica computacional.

7. Referências

AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas.* 2. ed. São Paulo: Pearson Addison Wesley, 2007.

UNIVERSITY OF VIRGINIA. *Lexical Analysis With Flex, for Flex 2.6.3.* Department of Computer Science, University of Virginia, 2017. Disponível em: https://www.cs.virginia.edu/~cr4bd/flex-manual/. Acesso em: 20 out. 2025.

8. Anexos

https://github.com/JoaoSCoelho/trab-compiladores