Universidade Federal de Santa Catarina

Relatório T1 Construção de Compiladores

Isac de Souza Campos (17200449)

Arthur Philippi Bianco (17203358)

Enzo Coelho Albornoz (18100527)

Identificação dos tokens

A tabela gerada pelo código utilizado neste trabalho contém um total de 40 *tokens*, que são listados na tabela 1 e relacionados ao seu padrão, lexema e descrição. Os primeiros *tokens* a serem listados serão os oriundos de palavras reservadas, seguidos do restante. São eles:

Token	Padrão	Lexema	Descrição
kw_def	Sequência dos caracteres: d, e, f.	def	Palavra reservada
kw_break	Sequência dos caracteres: b, r, e, a, k.	break	Palavra reservada
kw_new	Sequência dos caracteres: n, e, w.	new	Palavra reservada
kw_for	Sequência dos caracteres: f, o, r.	for	Palavra reservada
kw_if	Sequência dos caracteres: i, f.	if	Palavra reservada
kw_else	Sequência dos caracteres: e, l, s, e.	else	Palavra reservada
kw_print	Sequência dos caracteres: p, r, i, n, t.	print	Palavra reservada
kw_read	Sequência dos	read	Palavra

	caracteres: r, e, a, d.		reservada
kw_return	Sequência dos caracteres: r, e, t, u, r, n.	return	Palavra reservada
kw_int	Sequência dos caracteres: i, n, t.	int	Palavra reservada
kw_float	Sequência dos caracteres: f, l, o, a, t.	float	Palavra reservada
kw_string	Sequência dos caracteres: s, t, r, i, n, g.	string	Palavra reservada
kw_null	Sequência dos caracteres: n, u, l, l.	null	Palavra reservada
kw_comma	,	,	Separador
kw_semicolon	;	;	Separador
kw_attrib	=	=	Comando de atribuição
kw_plus	+	+	Operador matemático
kw_minus	-	-	Operador matemático
kw_mult	*	*	Operador matemático
kw_div	/	/	Operador matemático
kw_mod	%	%	Operador matemático
kw_lt	<	<	Comparador lógico
kw_gt	>	>	Comparador

			lógico
kw_lte	<=	<=	Comparador lógico
kw_gte	>=	>=	Comparador lógico
kw_eq	==	==	Comparador lógico
kw_ne	!=	!=	Comparador lógico
kw_paren_open	(	(	Delimitador de início
kw_paren_close	)	)	Delimitador de fim
kw_bracket_open	[	[	Delimitador de início
kw_bracket_close	]	]	Delimitador de fim
kw_cur_bracket_open	{	{	Delimitador de início
kw_cur_bracket_close	}	}	Delimitador de fim
ident	Cadeia de caracteres que se iniciem com letra o sublinhado (“_”).	valor, _nome, ...	Identificador
int_constant	Valores numéricos inteiros	…, -5, …, 0, …, 10, ...	Inteiro constante
float_constant	Valores numéricos fracionários	…, -10.5, 0, …, 5.0, ...	Fração constante
string_constant	Aspas (“”) com caracteres ou não no meio	“Olá, Mundo!”, “”, ...	String constante
s_char_sequence	Sequência de	“a”, “ab”,	Sequência de

	caracteres interligados	“abc”, ...	caracteres
s_char	Caractere contido na tabela ASCII	‘a’, ‘b’, ‘3’, ‘(‘, ...	Caractere
s_char_sequence_scape	Sequência de caracteres que serão traduzidas em outros caracteres inviáveis de serem representados	\, \\, \’, \”, \?, \a, \b, \f, \n, \r, \t, \v.	Sequência de escape

Tabela 1: Tokens utilizados no trabalho e seus respectivos padrões, lexemas e descrições.

O trabalho tem como objetivo gerar uma tabela de *tokens* com o que foi listado na tabela 1 e demonstrar como esse processo foi desenvolvido e seu funcionamento.

Produção das definições regulares para cada token

Definição regulares são da forma:

D1 -> R1

D2 -> R2

...

Di -> Ri, onde Di é um nome distinto e cada Ri é uma expressão regular **Σ U {D1, D2, …, Di}.** Seguem as definições regulares para cada token (primeiro os triviais):

kw_def -> def
kw_break -> break
kw_new -> new
kw_for -> for
kw_if -> if
kw_else -> else
kw_print -> print
kw_read -> read
kw_return -> return
kw_paren_open -> (
kw_paren_close ->)
kw_bracket_open -> [
kw_bracket_close ->]
kw_cur_bracket_open -> {

```
kw_cur_bracket_close -> }
kw_int -> int
kw_float -> float
kw_string -> string
kw_null -> null
kw_comma -> ,
kw_semicolon -> ;
kw_attrib -> =
kw_plus -> +
kw_minus -> -
kw_mult -> *
kw_div -> /
kw_mod -> %
kw_lt -> <
kw_gt -> >
kw_lte -> <=
kw_gte -> >=
kw_eq -> ==
kw_ne -> !=
```

E os não triviais:

```
ident -> (ASCII_ALPHA | _)(ASCII_ALPHANUMERIC | _)*
int_constant -> ASCII_DIGIT+
float_constant -> ASCII_DIGIT+ (. ASCII_DIGIT+) ((e | E) (+ | -)? ASCII_DIGIT+)?

string_constant -> " s_char_sequence? “
s_char_sequence -> (s_char s_char_sequence) | s_char
s_char -> (!( " | \ | NEWLINE) ANY) | s_char_sequence_escape
s_char_sequence_escape -> ' | \" | \? | \\ | \a | \b | \f | \n | \r | \t | \v
```

A biblioteca que usamos define *built in rules* (regras embutidas) para especificarmos a expressão regular mais convenientemente:

Regra embutida	Significado	Equivalente
ASCII_DIGIT	Dígitos	0 | 1 | … | 9
ASCII_ALPHA		a .. z | A .. Z
ASCII_ALPHANUMERIC	Qualquer dígito ou letra	ASCII_DIGIT | ASCII_ALPHA
NEWLINE	Qualquer quebra de linha	\n | \r\n | \r
ANY	Qualquer único caractere unicode	

*obs o NEWLINE é representado por um “~” em código.

Construção dos diagramas de transição

Os diagramas de transição usados na análise léxica são semelhantes a autômatos e descrevem o comportamento que deve ser seguido pelo analisador para que a tabela de *tokens* seja montada de maneira adequada.

Neste trabalho, foram desenvolvidos diagramas para dois grupos de gramáticas: *Keywords* e *Non Trivial Terminals*. O primeiro consiste em termos simples e diretos, sem quaisquer tipos de possível variação (“def”, “{”, “int”, “=”, “+”, etc). O segundo grupo, por outro lado, possui uma complexidade maior, permitindo diferentes resultados (estados finais) ao final de sua execução.

Para a criação de tais diagramas foi utilizada a ferramenta online LucidChart (https://lucid.app/). Não foi encontrada na ferramenta uma representação padrão do estado final de diagramas de transição (dois círculos concêntricos). Para contornar esta limitação, utilizou-se um círculo pontilhado para retratar tais estados.

Os resultados obtidos na elaboração dos diagramas das *keywords* podem ser vistos nas figuras 1 a 4:

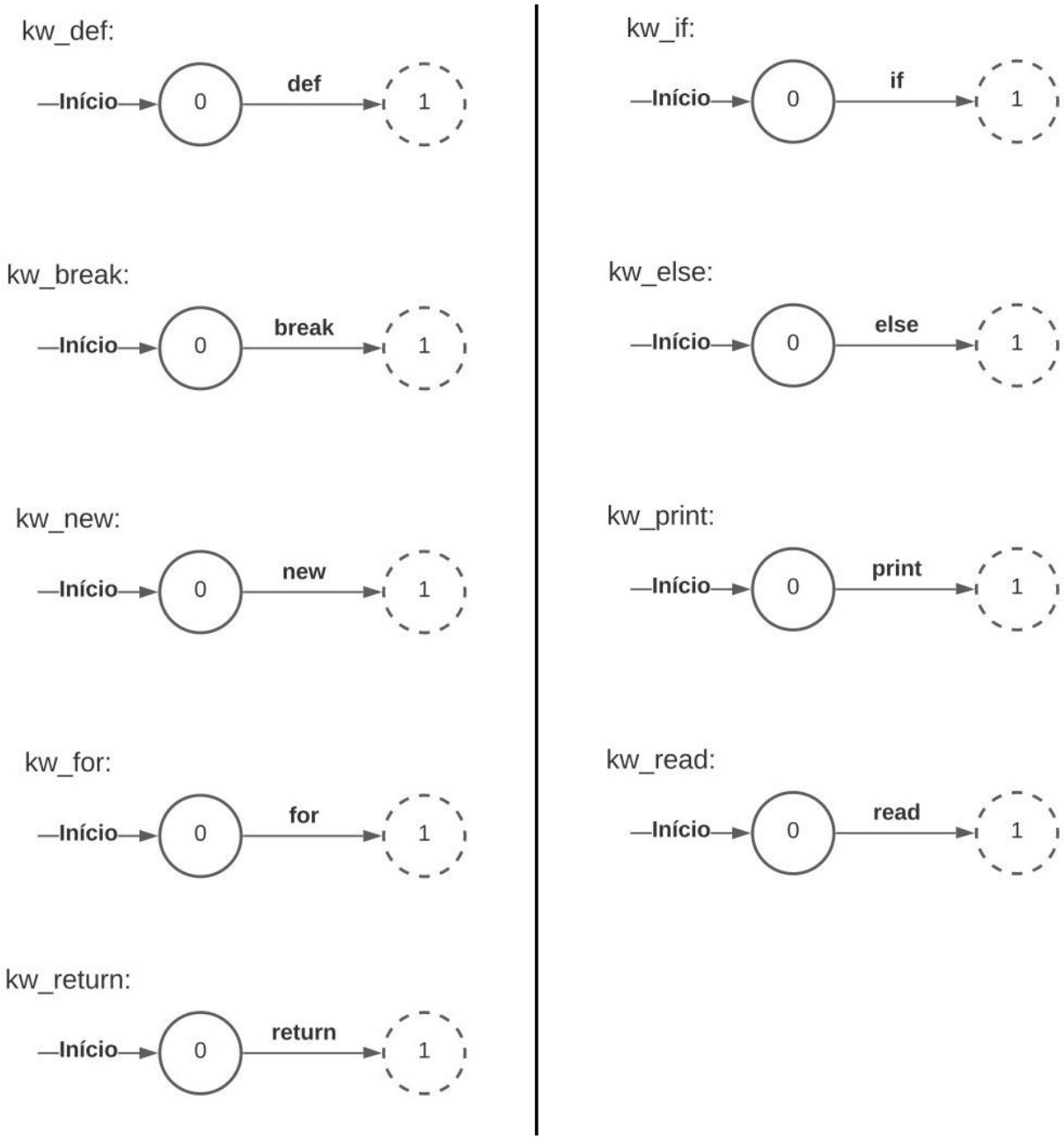

Figura 1: diagrama de transição dos *tokens*: kw_def, kw_break, kw_new, kw_for, kw_return, kw_if, kw_else, kw_print, kw_read.

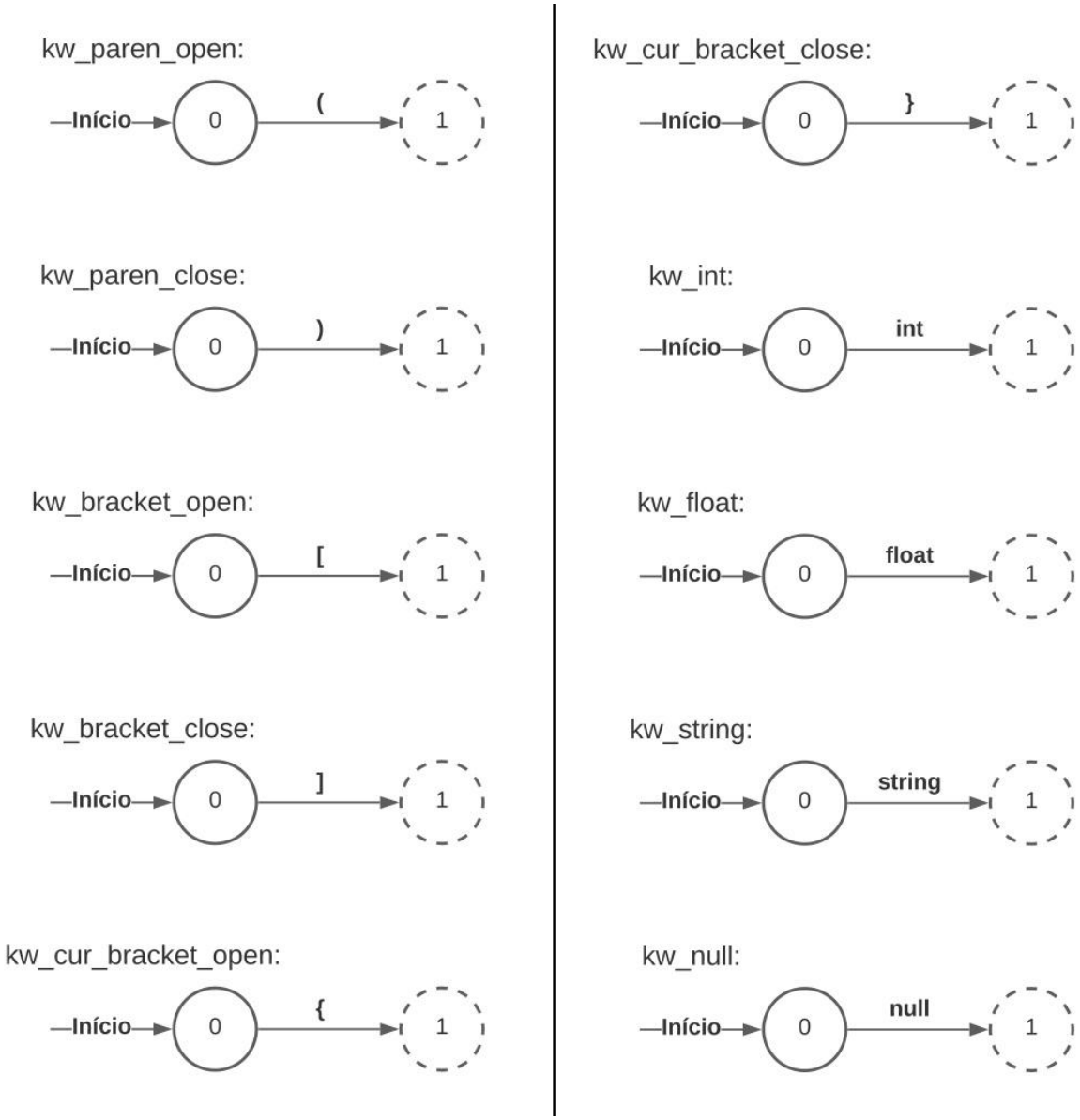

Figura 2: diagramas de transição dos *tokens*: kw_paren_open, kw_paren_close, kw_bracket_open, kw_bracket_close, kw_cur_bracket_open, kw_cur_bracket_close, kw_int, kw_float, kw_string, kw_null.

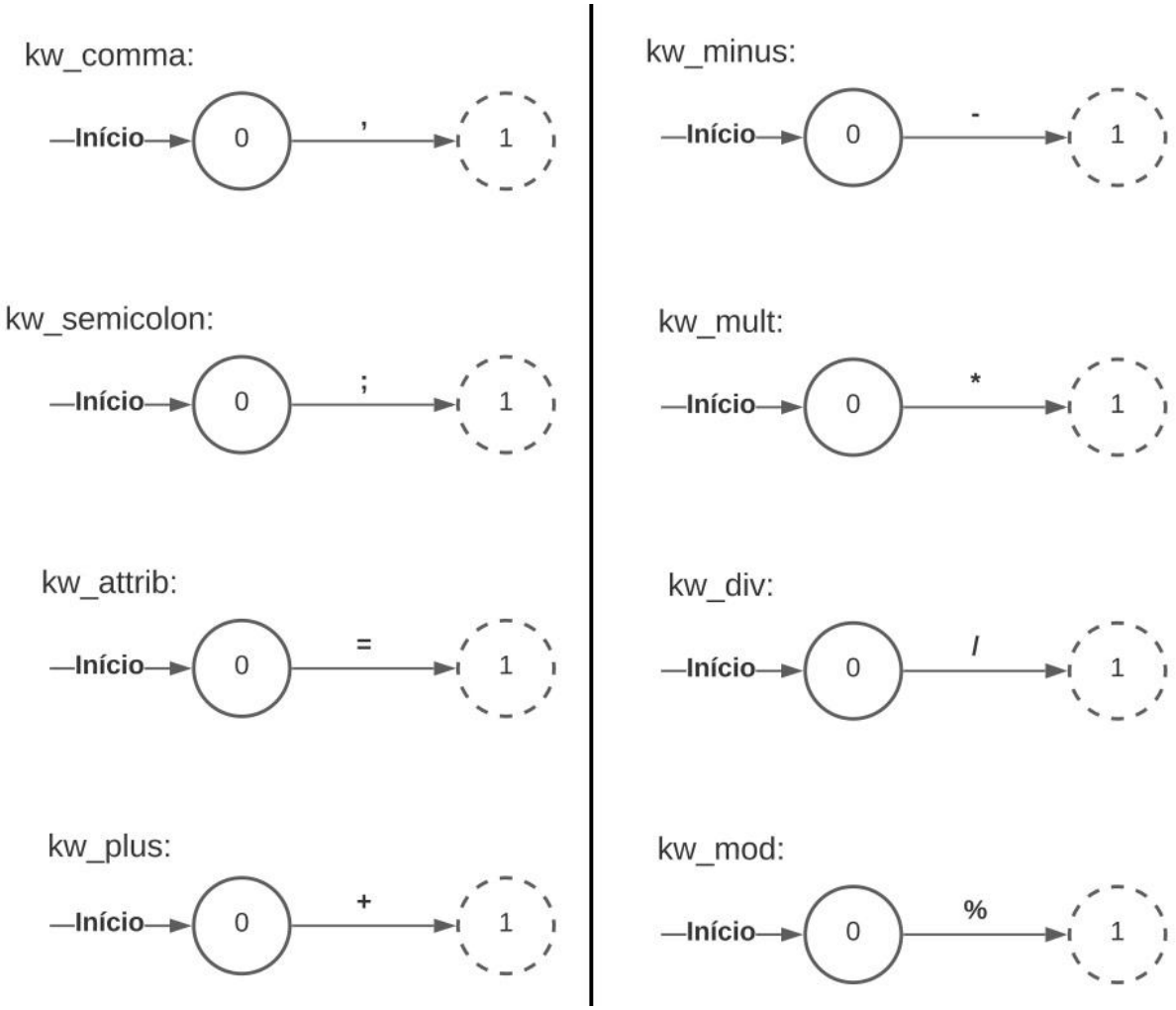

Figura 3: diagramas de transição dos *tokens*: kw_comma, kw_semicolon, kw_attrib, kw_plus, kw_minus, kw_mult, kw_div, kw_mod.

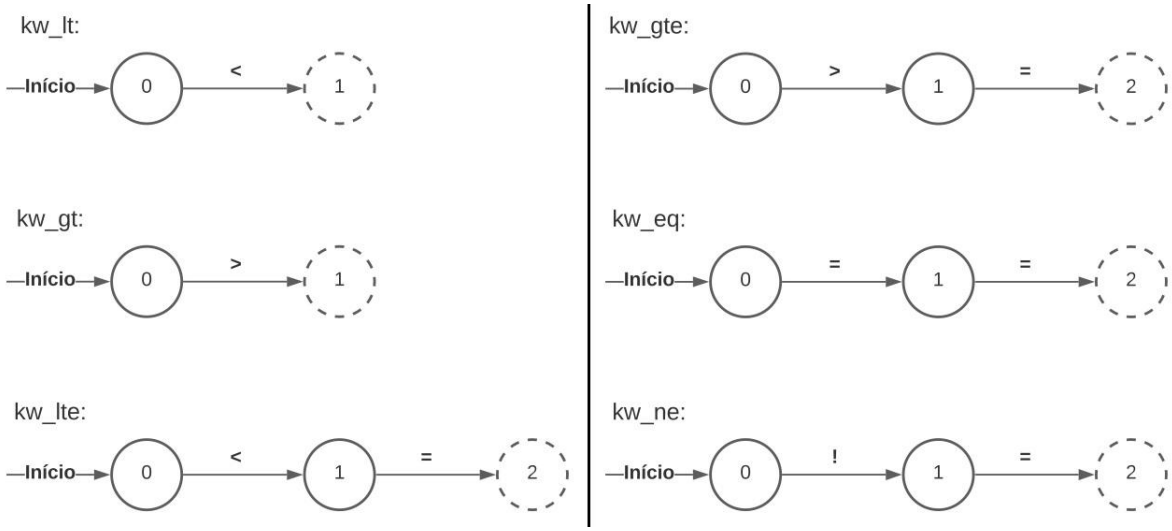

Figura 4: diagramas de transição dos *tokens*: kw_lt, kw_gt, kw_lte, kw_gte, kw_eq, kw_ne.

Diferentemente dos diagramas vistos nas figuras 1 a 4, os *Non Trivial Terminals* das figuras 5 a 11 são um pouco mais complexos. Os diagramas usados para reconhecer *strings*, caracteres e suas variantes, presentes na figura 8 a 11, utilizam como condição de transição outros pré-existentes. A ordem em que eles são dispostos no relatório foi escolhida de modo que sempre quando um diagrama pré-existente for usado em outro como transição, este já tenha sido demonstrado anteriormente. Antes, porém, serão apresentados os que não possuem interdependências.

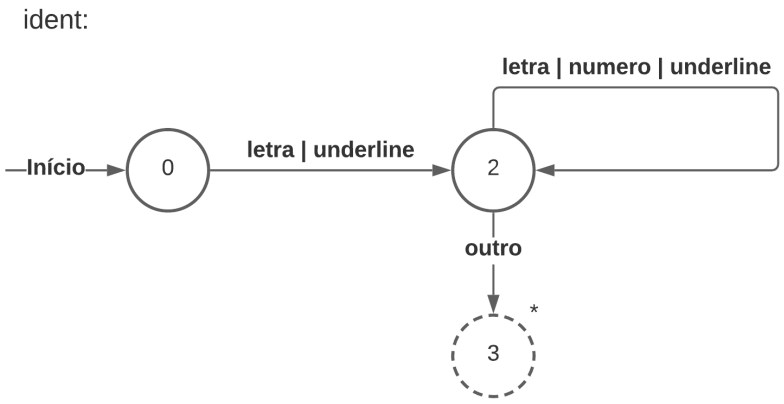

Figura 5: diagrama de transição do *token* ident.

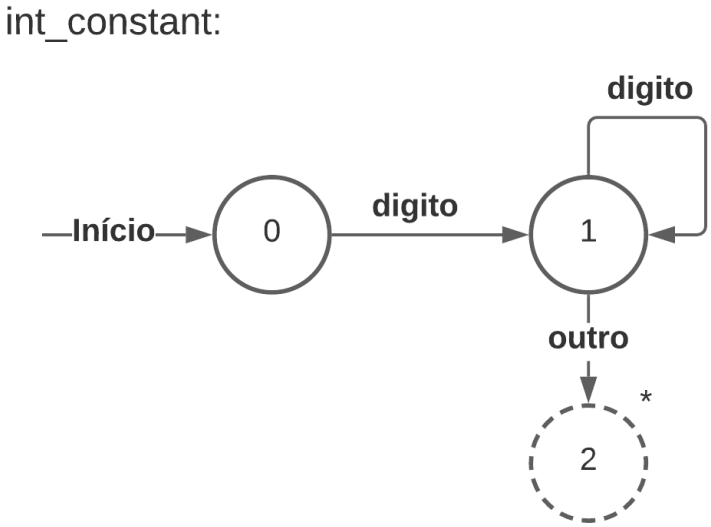

Figura 6: diagrama de transição do *token* int_constant.

float_constant:

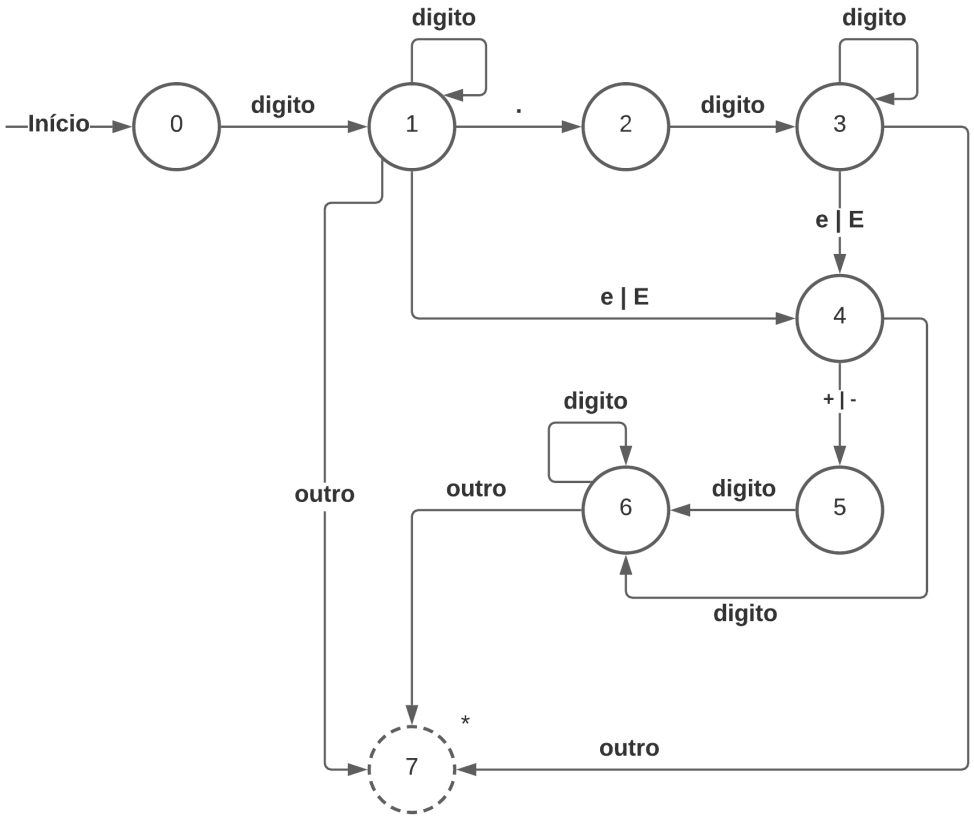

Figura 7: diagrama de transição do *token* float_constant.

O diagrama seguinte da figura 8 dá início às interdependências vistas nos *tokens* das variantes de *strings* que vão até a figura 11.

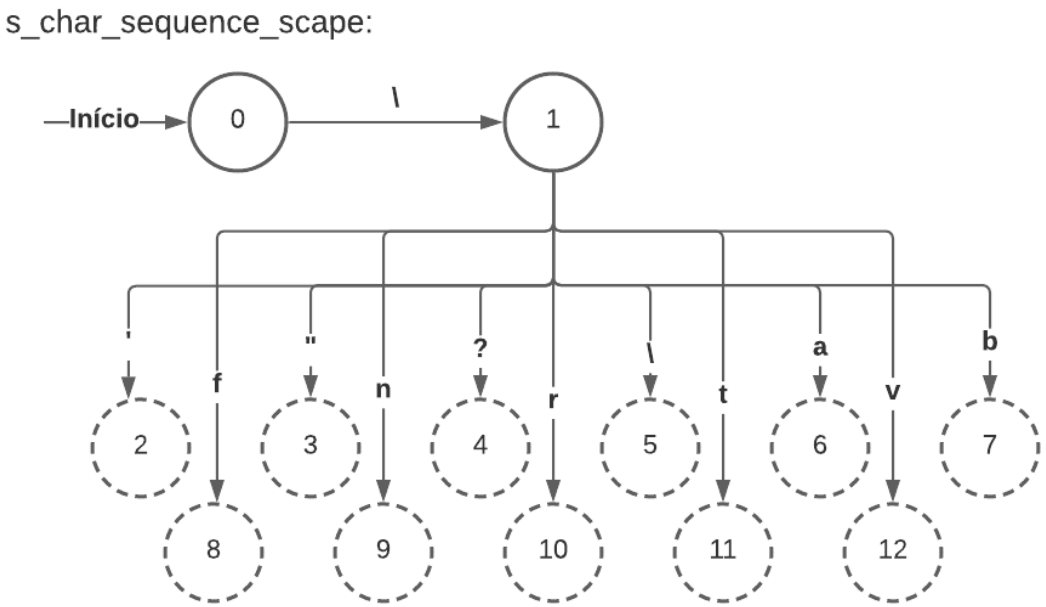

Figura 8: diagrama de transição do *token* s_char_sequence_scape.

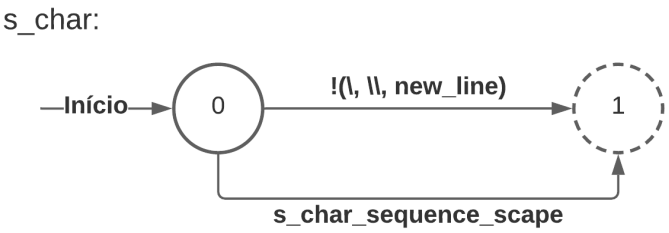

Figura 9: diagrama de transição do *token* s_char com dependência de s_char_sequence_scape.

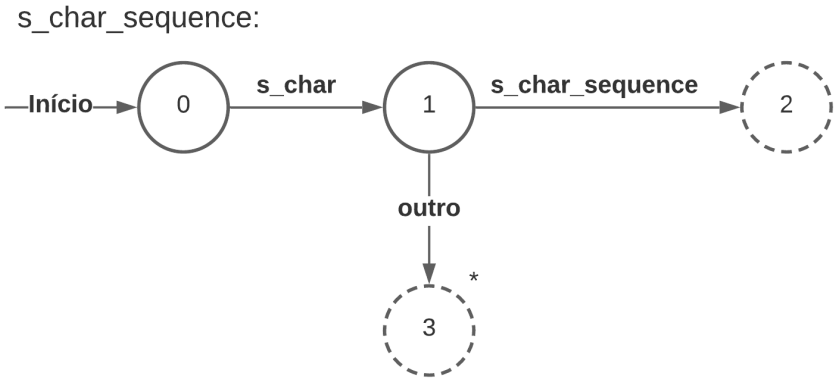

Figura10: diagrama de transição do *token* s_char_sequence com dependência de s_char.

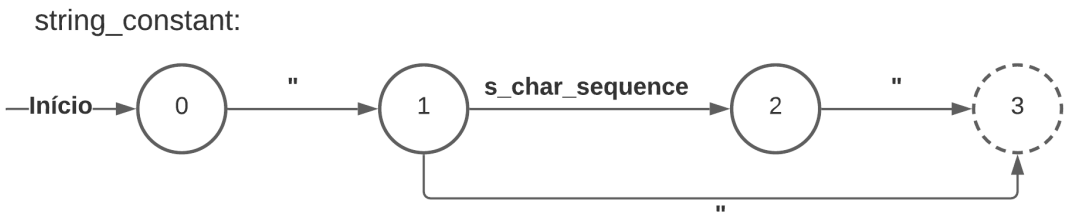

Figura11: diagrama de transição do *token* string_constant com dependência de s_char_sequence.

Tabela de Símbolos

A tabela de símbolos foi desenvolvida baseada no modelo de *Hash Table*, onde temos a entrada sendo o texto identificado como o símbolo, que mapeia para a estrutura de dados *SymbolTableEntry*. Essa estrutura contém os seguintes atributos:

- *Lexeme*: Guarda a própria cadeia identificada como o símbolo;
- *Length*: Guarda o tamanho da cadeia;
- *Type*: Enumeração do tipo de token do símbolo;
- *Occurrences:* Vetor contendo todas as incidências do símbolo no código fonte.
- *Attributes*: Enumeração Estruturada que guarda os atributos do símbolo. Os atributos possíveis variam de acordo com o tipo do símbolo identificado.

Na tabela de símbolo são armazenados os símbolos dos tokens de tipo:

- Identificador: Não possui atributos especiais, ainda.
- Constante de Cadeia: Possui o próprio valor armazenado, porém no tipo apropriado.
- Constante de Inteiro: Possui o próprio valor armazenado, porém no tipo apropriado.
- Constante de Ponto Flutuante: Possui o próprio valor armazenado, porém no tipo apropriado.

Uso de Ferramenta

A ferramenta escolhida para auxiliar no desenvolvimento do trabalho foi a biblioteca ***Pest***. Essa biblioteca permite definir diversas regras de análise de gramática. Com ela, definimos as regras de formação de tokens através de um arquivo `.pest` (o qual possui uma linguagem própria). Segue abaixo um exemplo dessas regras:

```
kw_def = @{ "def" }
kw_break = @{ "break" }
kw_new = @{ "new" }
kw_for = @{ "for" }
kw_if = @{ "if" }
kw_else = @{ "else" }
kw_print = @{ "print" }
kw_read = @{ "read" }
kw_return = @{ "return" }
```

```
ident = @{ (ASCII_ALPHA | "_") ~ (ASCII_ALPHANUMERIC | "_")* }

int_constant = @{ ASCII_DIGIT+ }

float_constant = @{ ASCII_DIGIT+ ~ ("." ~ ASCII_DIGIT+) ~ (("e"|"E") ~ ("+"|"-")? ~ ASCII_DIGIT+)? }
```

Como podemos notar, a sintaxe é simples. No código, importamos o arquivo e utilizamos ele definindo uma regra de entrada.

```
// Configure Parser

You, a week ago | 1 author (You)
#[derive(Parser)]
#[grammar = "grammar.pest"]
1 implementation
struct ParserCC20211;
```

```
// Try Parse Program
let parsing_timing: Instant = Instant::now();
match ParserCC20211::parse(Rule::tokens, input: &file_content) {
    Ok(pairs: Pairs<Rule>) => {
        // Success Parsed
        let elapsed_time: Duration = parsing_timing.elapsed();
        // Generate Tokens List
```

Como resultado, temos os tokens identificados em forma de um iterador. Assim conseguimos ir avançando pela árvore de derivação e gerando a tabela de símbolos e a lista de tokens.
Abaixo temos alguns exemplos de entradas e saídas com os tokens da linguagem **CC20211**.

- Análise léxica de um *AttribStat* :
    - Entrada:

```
fn test_atribstat_rule() {
    let string: &str = "k = 10.5";
    let pairs: Pairs<Rule> = ParserCC20211::parse(Rule::atribstat, input: string).expect(msg: "Error ocurred");
    println!("{}", pairs);
}
```

    - Saída :

```
running 1 test
[atribstat(0, 8, [lvalue(0, 2, [ident(0, 1)]), kw_attrib(2, 3), expression(4, 8, [numexpression(4, 8, [term(4, 8, [unaryexpr(4, 8, [factor(4, 8,
 [float_constant(4, 8)])])])])])])]
test test::test_atribstat_rule ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in 0.00s
```

- Análise léxica de uma Constante de Cadeia
    - Entrada:

```
fn test_string_rule() {
    let string: &str = r#""hello, \nworld!""#;
    let pairs: Pairs<Rule> = ParserCC20211::parse(Rule::string_constant, input: string).expect(msg: "Error ocurred");
    println!("{}", pairs);
}       You, a week ago • Better token detection
```

    - Saída

```
running 1 test
[string_constant(0, 17, [s_char_sequence(1, 16)])]
test test::test_string_rule ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in 0.00s
```

- Análise Léxica de um código fonte
    - Entrada :

```
t.rs > test_example1
    {
      float x;
      float z;
      int i;
      int max;
      x = 0;
      max = 10000;
      for (i = 1; i ≤ max; i = i + 1){
        print x;
        x = x + 0.001;
        z = x;
        if (z ≠ x){
          print \"Erro numérico na atribuição de números na notação ponto flutuante!\";
          break;
        }
      }
    }


    {
      int y;
      int j;
      int i;
      y = new int[10];
      j = 0;
      for (i = 0; i < 20; i = i + 1)
        if (i % 2 == 0){
          y[j] = i + 1;
          j = j + 1;
        }
        else
          print 0;

      for (i = 0; i < 10; i = i + 1)
        print y[i];

      return;
    }
    }
  ";

  let pairs_result: Result<Pairs<Rule>, Error<Rule>> = ParserCC20211::parse(Rule::tokens, input: test_program);
```

    - Saída :

```
running 1 test
[tokens(0, 846, [kw_cur_bracket_open(5, 6), kw_cur_bracket_open(15, 16), kw_float(27, 32), ident(33, 34), kw_semicolon(34, 35), kw_float(46, 51)
, ident(52, 53), kw_semicolon(53, 54), kw_int(65, 68), ident(69, 70), kw_semicolon(70, 71), kw_int(82, 85), ident(86, 89), kw_semicolon(89, 90),
 ident(101, 102), kw_attrib(103, 104), int_constant(105, 106), kw_semicolon(106, 107), ident(118, 121), kw_attrib(122, 123), int_constant(124, 1
29), kw_semicolon(129, 130), kw_for(141, 144), kw_paren_open(145, 146), ident(146, 147), kw_attrib(148, 149), int_constant(150, 151), kw_semicol
on(151, 152), ident(153, 154), kw_lte(155, 157), ident(158, 161), kw_semicolon(161, 162), ident(163, 164), kw_attrib(165, 166), ident(167, 168),
 kw_plus(169, 170), int_constant(171, 172), kw_paren_close(172, 173), kw_cur_bracket_open(173, 174), kw_print(187, 192), ident(193, 194), kw_sem
icolon(194, 195), ident(208, 209), kw_attrib(210, 211), ident(212, 213), kw_plus(214, 215), float_constant(216, 221), kw_semicolon(221, 222), id
ent(235, 236), kw_attrib(237, 238), ident(239, 240), kw_semicolon(240, 241), kw_if(254, 256), kw_paren_open(257, 258), ident(258, 259), kw_ne(26
0, 262), ident(263, 264), kw_paren_close(264, 265), kw_cur_bracket_open(265, 266), kw_print(281, 286), string_constant(287, 361, [s_char_sequenc
e(288, 360)]), kw_semicolon(361, 362), kw_break(377, 382), kw_semicolon(382, 383), kw_cur_bracket_close(396, 397), kw_cur_bracket_close(408, 409
), kw_cur_bracket_close(418, 419), kw_cur_bracket_open(446, 447), kw_int(458, 461), ident(462, 463), kw_semicolon(463, 464), kw_int(475, 478), i
dent(479, 480), kw_semicolon(480, 481), kw_int(492, 495), ident(496, 497), kw_semicolon(497, 498), ident(509, 510), kw_attrib(511, 512), kw_new(
513, 516), kw_int(517, 520), kw_bracket_open(520, 521), int_constant(521, 523), kw_bracket_close(523, 524), kw_semicolon(524, 525), ident(536, 5
37), kw_attrib(538, 539), int_constant(540, 541), kw_semicolon(541, 542), kw_for(553, 556), kw_paren_open(557, 558), ident(558, 559), kw_attrib(
560, 561), int_constant(562, 563), kw_semicolon(563, 564), ident(565, 566), kw_lt(567, 568), int_constant(569, 571), kw_semicolon(571, 572), ide
nt(573, 574), kw_attrib(575, 576), ident(577, 578), kw_plus(579, 580), int_constant(581, 582), kw_paren_close(582, 583), kw_if(597, 599), kw_par
en_open(600, 601), ident(601, 602), kw_mod(603, 604), int_constant(605, 606), kw_eq(607, 609), int_constant(610, 611), kw_paren_close(611, 612),
 kw_cur_bracket_open(612, 613), ident(628, 629), kw_bracket_open(629, 630), ident(630, 631), kw_bracket_close(631, 632), kw_attrib(633, 634), id
ent(635, 636), kw_plus(637, 638), int_constant(639, 640), kw_semicolon(640, 641), ident(656, 657), kw_attrib(658, 659), ident(660, 661), kw_plus
(662, 663), int_constant(664, 665), kw_semicolon(665, 666), kw_cur_bracket_close(679, 680), kw_else(693, 697), kw_print(712, 717), int_constant(
718, 719), kw_semicolon(719, 720), kw_for(740, 743), kw_paren_open(744, 745), ident(745, 746), kw_attrib(747, 748), int_constant(749, 750), kw_s
emicolon(750, 751), ident(752, 753), kw_lt(754, 755), int_constant(756, 758), kw_semicolon(758, 759), ident(760, 761), kw_attrib(762, 763), iden
t(764, 765), kw_plus(766, 767), int_constant(768, 769), kw_paren_close(769, 770), kw_print(783, 788), ident(789, 790), kw_bracket_open(790, 791)
, ident(791, 792), kw_bracket_close(792, 793), kw_semicolon(793, 794), kw_return(814, 820), kw_semicolon(820, 821), kw_cur_bracket_close(830, 83
1), kw_cur_bracket_close(840, 841), EOI(846, 846)])]
test test::test_example1 ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 2 filtered out; finished in 0.00s
```