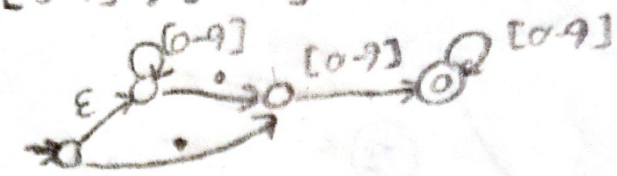
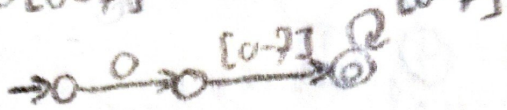
Teste EN 2025 (Teste reconstruído de memória, pode haver coisas erradas)

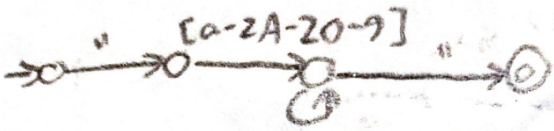
1a) DEC $[0-9]^*.[0-9]^+$



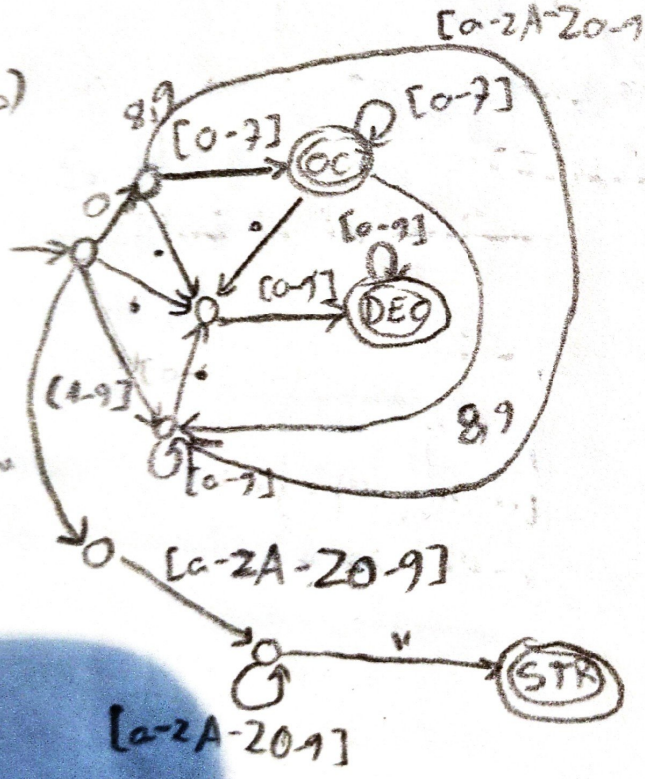
OCT $0[0-7]^+$



STR "[a-zA-Z0-9]"



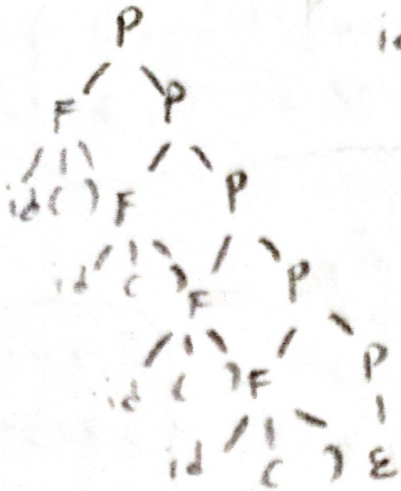
b)



c) 0"a"00"b"00.48.7"5.40"

ERR:0"
STR:"a"
OCT:00
STR:"b"
DEC:00.48
DEC:.7
ERR:"5.
DEC:5.40 ERR:"$

2 a)



id().id().id().id()

b)

Expandindo a gramática, temos:

P → id()P
| id var P
| ε

Uma gramática não é LL(1) se, para algum $A \to \alpha \mid \beta$,

$First(\alpha) \cap First(\beta) \neq \emptyset$

isso verifica-se nesta gramática pois

$First(id()P) \cap First(id\ var\ P) = \{id\}$

então a gramática não é LL(1)

c)

1 P → id P'
2 | ε
3 P' → ()P
4 | var P

$First(P) = \{id, \varepsilon\}$

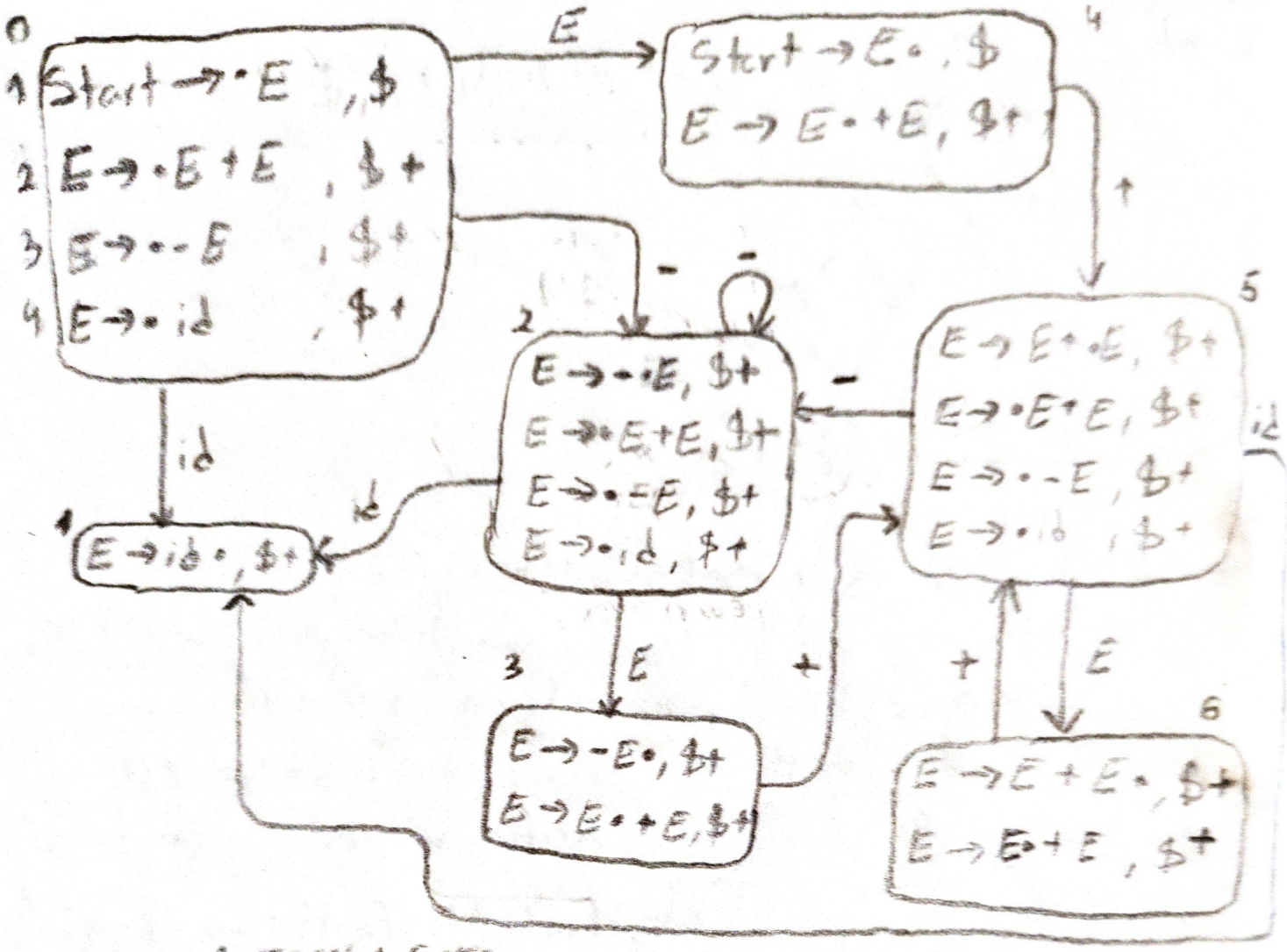
$First(P') = \{(, var\}$

$Follow(P) = \{\$\} \cup Follow(P) = \{\$\}$

$Follow(P') = Follow(P) = \{\$\}$

	(	)	id	var	$
P			1		2
P'	3			4	

3
a)



b)

	ACTION				GOTO	
	+	-	id	$	E	S
0		s2	s1		g4	
1	r4			r4		
2		s2	s1		g3	
3	r3 / s5			r3		
4	s5			Acc		
5		s2	s1		g6	
6	r2 / s5			r2		

T[3, +] = r3 } Prioridade

T[6, +] = r2 } Associativo à esquerda

c) Start → E { $$ = newnode(Program, NULL), addchild($$, $1); }

E → E + E { $$ = newnode(ADD, NULL), addchild($$, $1), addchild($$, $3); }

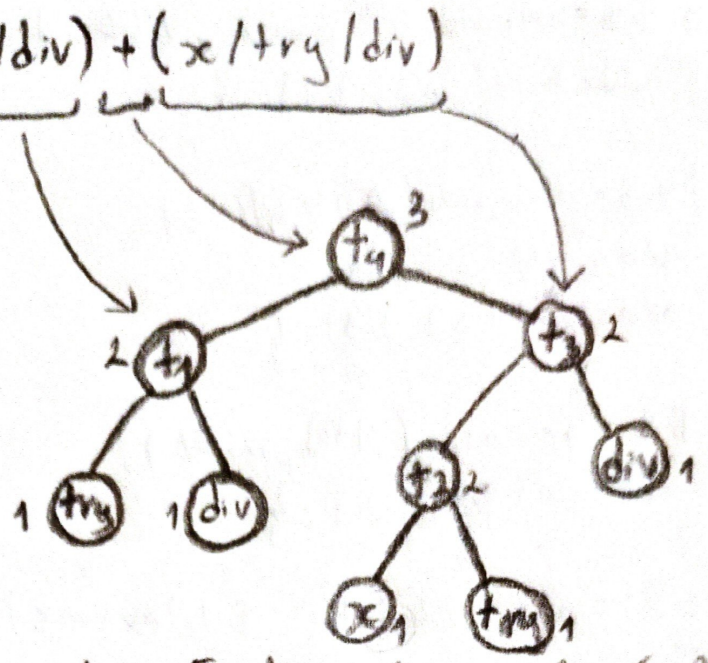
| - E { $$ = newnode(MIN, NULL); addchild($$, $2); }

| id { $$ = newnode(ID, $1.lexeme), }

4 a)

```
sqrt:
 1  x := param[0]
 2  err := 0,000006
 3  div := 2
 4  try := x
 5  t1 := try / div
 6  t2 := x / t1
 7  next := t1 + t2
 8  t3 := next - try
 9  iFFALSE t3 < err goto 13
10  t4 := try - next
11  iFFALSE t4 < err goto 13
12  goto 15
13  try := next
14  goto 5
15  t5 := try
16  return try
```

b) (try/div) + (x/try/div)



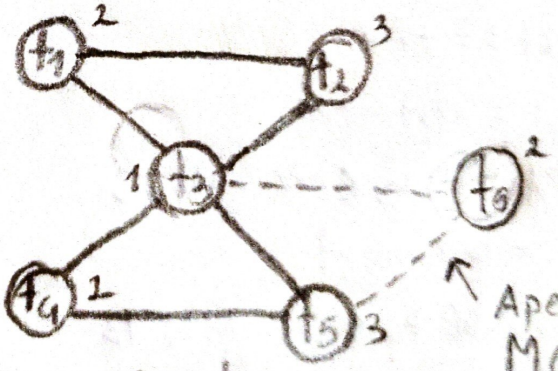
O número de Ershov da root é 3, portanto são precisos, no mínimo 3 registos.

Como?

Lifetimes

$t_1 := try$	t_1	t_2
$t_2 := div$		
$t_3 := t_1 / t_2$		
$t_4 := x$	t_4	t_3
$t_5 := t_4 / t_3$		
$t_6 := t_3 + t_5$	t_5	t_6

Podemos fazer o grafo de interferência



Apenas MOV

Colorindo de forma greedy, começando em t_3, temos:

Node	Cor
t_1	2
t_2	3
t_3	1
t_4	2
t_5	3
t_6	2

Então, com os registos R1, R2 e R3 podemos reescrever a expressão como:

```
MOV  R2  try
MOV  R3  div
DIV  R1  R2  R3
MOV  R2  x
DIV  R3  R2  R1
ADD  R2  R1  R3
```

4 e) Return value, old stack pointer, frame pointer, parâmetros, espaço para spilling, variáveis locais e temporárias

Alternativas :- Passar parâmetros por registos
✓ velocidade
X maior chance de spilling

- Passar parâmetros pela heap:
X não é útil pois "double" é de tamanho estático
- Saved registers; caso haja registos convencionados "callee-save"
- Guardar return address num registo: neste caso é adequado porque se trata de uma função folha
- Usar display, lambda lifting ou apenas um static link para voltar à ~~frame~~ da caller function.

- etc ...