Prova com consulta. Duração: 1h30m. Primeiro Mini-Teste

```
S →  "url" "(" STRING ")"          // basic services
   | S "?" S                       // sequential execution
   | S "|" S                       // concurrent execution
   | "timeout" "(" REAL "," S ")"  // timeout combinator
   | "repeat" "(" S ")"            // repetition
   | "stall"                       // nontermination
   | "fail"                        // failure
```

Trecho da gramática apresentada em [Cardelli and Davies, 1999]

Grupo 1. Análise Lexical e Sintáctica (11 valores)

Considere a gramática apresentada à esquerda na qual as produções estão comentadas (texto iniciado com os símbolos "//").

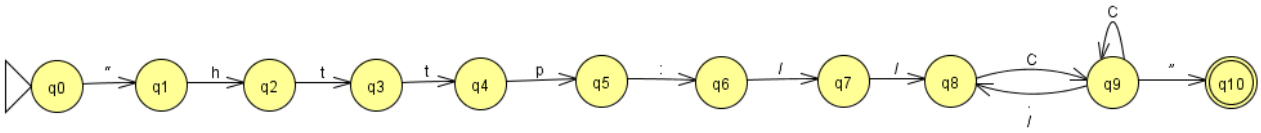
1.a) [1v] Apresente as expressões regulares para definir os tokens STRING e REAL para a gramática apresentada. Considere que STRING deve representar possíveis URLs HTTP (considere que podem conter "/", ".", ":", letras do alfabeto, e algarismos) entre aspas duplas, e REAL deve representar números inteiros ou reais, ambos sem sinal. Use o "." para separar a parte inteira da fraccionária dos números reais.

R: Possíveis expressões regulares:
*STRING = "*http://*[a-zA-Z0-9]+ (.[a-zA-Z0-9]+)*("/" [a-zA-Z0-9]+ (.[a-zA-Z0-9]+)*)*"*
REAL = [0-9]+ ("." [0-9]+)?
■

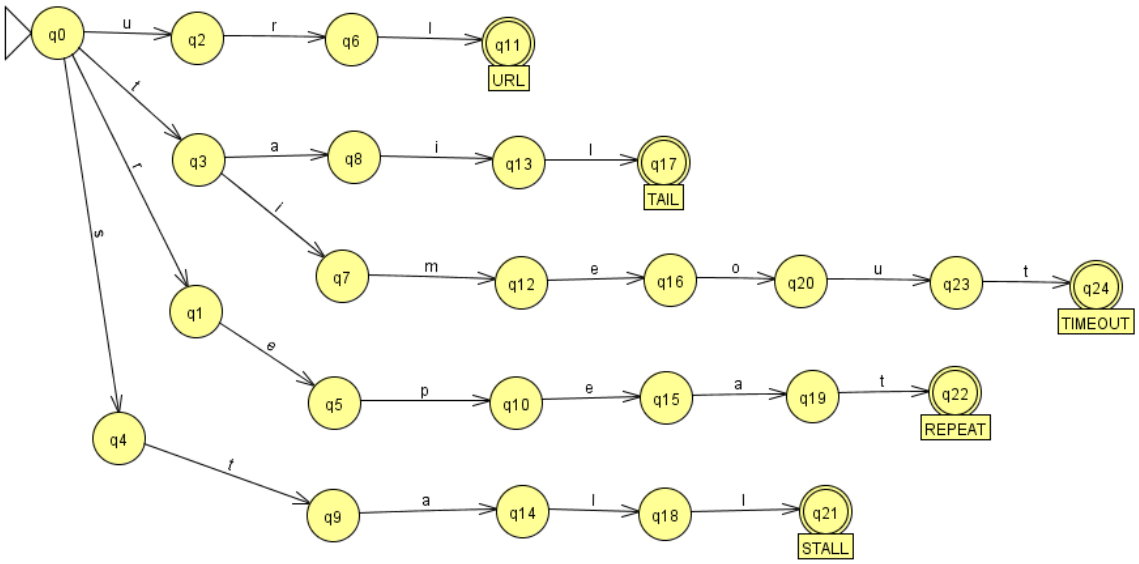
1.b) [2v] Desenhe um DFA para a análise lexical.

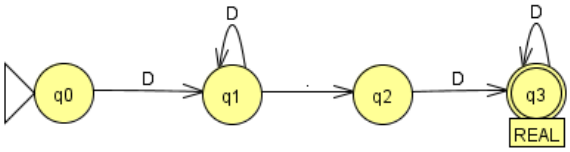
R: Notar que o DFA é apresentado sob forma parcial e que o DFA completo é constituído pelos DFAs apresentados sendo o estado q0 o estado inicial. (é assumido que o estado morto e as transições de e para este estado estão implícitos)

STRING:

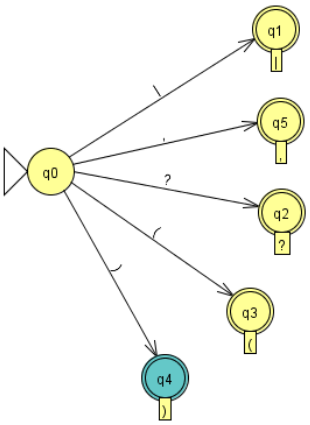


C no DFA representa: a..zA..Z0..9





D representa 0..9

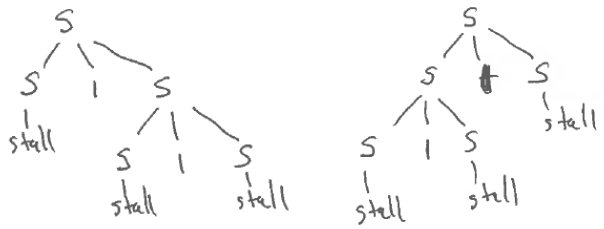


Nota: a cor diferente de q4 não tem significado especial.

■

1.c) [1v] A gramática apresentada é ambígua? Caso seja, desenhe duas árvores sintácticas diferentes para uma dada entrada.

R:
A gramática é ambígua.
Possíveis árvores sintácticas concretas para a entrada:
stall | stall | stall



■

1.d) [1v] A gramática tem recursividade à esquerda. Elimine a recursividade e indique a gramática modificada para a linguagem.

R:

S → *S1*
| S1 "?" S
| S1 "|" S

S →	*S1 S2*
S2 →	*"?" S	"|" S	ε*

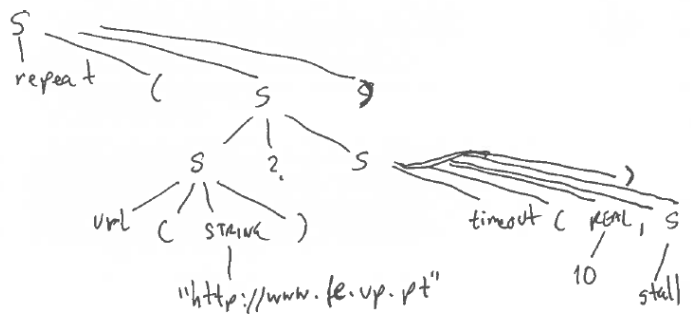
S →	*S1 S'	S1 S''*
S' →	*"|" S S'	ε*
S'' →	*"?" S S''	ε*

S1 → *"url" "(" STRING ")"*
| "timeout" "(" REAL "," S ")"
| "repeat" "(" S ")"
| "stall"
| "fail"

■

1.e) [0,5v] Apresente a árvore sintáctica concreta para a entrada:
repeat(url("http://www.fe.up.pt") ? timeout(10, stall)).

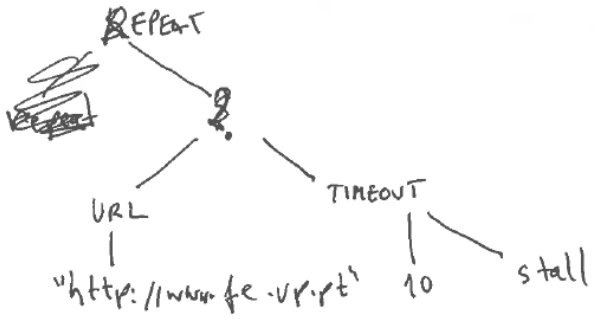
R:



■

1.f) [1v] Apresente uma possível árvore sintáctica abstracta (AST[1]) para o exemplo da alínea anterior.

R:



■

1.g) [2v] Apresente as funções necessárias e o respectivo pseudo-código para implementar a linguagem definida pela gramática apresentada com um analisar sintáctico descendente preditivo LL(1). Assuma a existência do analisador lexical que devolve a sequência de tokens, a existência da variável global *token*, a função *next()* que devolve o token a seguir na sequência de tokens na entrada (no início a variável *token* identifica o primeiro token na sequência), e do atributo *id* de *token* que identifica o tipo de token. Indique os IDs que utilizou considerando os tipos de tokens utilizados.

R:

Nesta resolução optou-se por utilizer como IDs dos tokens o próprio valor ou STRING e REAL, para os casos dos tokens definidos como STRING e REAL, respectivamente.

Possível Gramática LL(1):

S → S1 ("?" S | "|" S)?
S1 → "url" "(" STRING ")"
| "timeout" "(" REAL "," S ")"
| "repeat" "(" S ")"
| "stall"
| "fail"

Possível Parser descendente para a gramática (nota: assume-se que o objective não era que o parser construísse a árvore sintáctica):

```
main() {
    …
    if(S() && token==null) println("Input accepted!");
    else println("Input not accepted!");
    …
}

boolean S() {
```

[1] Do Inglês: *Abstract Syntax Tree*

```
    if(!S1()) return false;
    If (token.id == "?") {
        token = next(); return S();
    } elsif(token.id == "|") {
        token = next(); return S();
    } else {
       return true;
    }
}
boolean S1() {
    if(token.id== "url") {
        token = next();
        if(token.id== "(") {
            token = next();
            if(token.id== STRING) {
                token = next();
                if(token.id== ")") {
                    token = next(); return true;
                } else return false;
            } else return false;
        } else return false;
    } elsif(token.id== "timeout") {
        if(token.id== "(") {
            token = next();
            if(token.id== REAL) {
                token = next();
                if(token.id== ",") {
                    token = next();
                    if(!S()) return false;
                    if(token.id == ")") {token=next(); return true;
                    } else return false;
                } else return false;
            } else return false;
      } else return false;
    } elsif(token.id== "repeat") {
        if(token.id== "(") {
            if(!S()) return false;
            if(token.id== ")") {
                token = next(); return true;
            }
    } elsif(token.id== "stall") {
        token = next(); return true;
    } elsif(token.id== "fail") {
        token = next(); return true;
    } else return false;
}
```
■

1.h) [1v] Indique como poderemos adicionar as produções

"gateway" "get" "(" STRING ")"

e

"gateway" "post" "(" STRING ")"

à variável S mantendo um *lookahead* de 1.

R:

S → "gateway" ("get" | "post") "(" STRING ")"

ou:

S → "gateway" S1 "(" STRING ")"

S1 → "get" | "post"

■

1.i) [0,5v] Defina as produções para a variável *Args* para que esta substitua STRING nas produções da alínea anterior, e de forma a que *Args* possa derivar um ou mais terminais do tipo STRING separados pelo símbolo ",".

R:
S → "gateway" ("get" | "post") "(" Args ")"

Args → STRING {"," STRING}

Ou

Args → STRING Args1
Args1 → "," STRING | ε
■

1.j) [0,5v] Apresente as modificações à gramática original de forma a que a derivação "timeout" "(" REAL "," S ")" possa aceitar opcionalmente a seguir a REAL a identificação das unidade de tempo a usar: "ms", "us", e "s".

R:
S → "timeout" "(" REAL Units "," S ")"
Units → "us" | "ms" | "s" | ε

Ou:
S → "timeout" "(" REAL [Units] "," S ")"
Units → "us" | "ms" | "s"
■

1.k) [0,5v] Indique se a adição das produções

"limit" "(" REAL "," REAL "," S ")"

e

"index" "(" STRING "," STRING ")"

à variável S implica alterações no valor do *lookahead* do parser em 1.g). Justifique a resposta dada.

R:
A adição dessas produções não implica alterações no valor de lookahead (= 1) dado que ambas apresentam dois tokens distintos como primeiro token, "limit" e "index", respectivamente, e que são também distintos de todos os primeiros tokens das outras produções de S.
■

Grupo 2. Análise Semântica I (5 valores)

Considere o trecho de código Java do exemplo seguinte.

Exemplo	1.	void insertionSort(int[] num, int size) {
	2.	int j;
	3.	for (int i = 1; i < size; i++) {
	4.	int key = num[i];
	5.	j = i - 1;
	6.	while((j>=0) && (key < num[j])) {
	7.	num[j+1] = num[j];
	8.	j--;
	9.	}
	10.	num[j+1] = key;
	11.	}
	12.	}

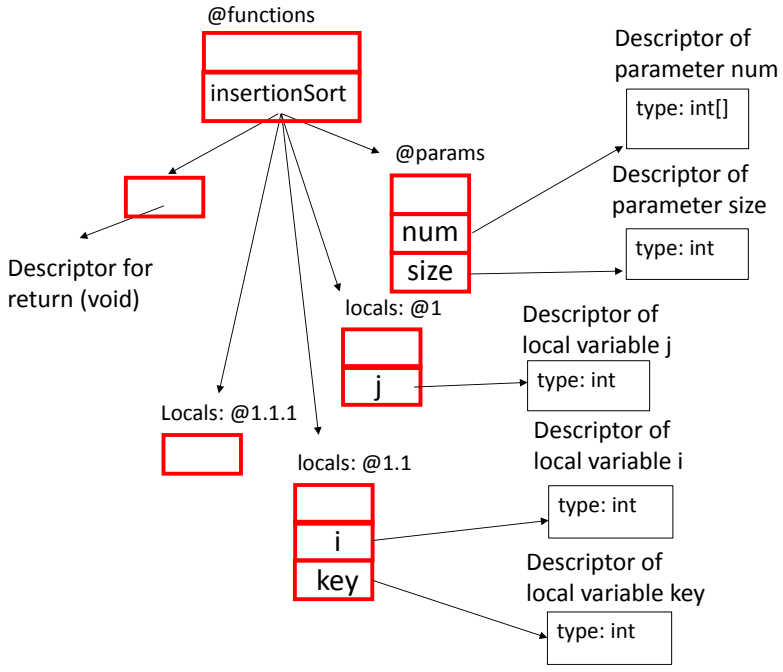
2.a) [2v] Assumindo que as variáveis usadas no código só podem ser parâmetros ou variáveis locais, indique uma possível tabela de símbolos para este exemplo indicando a informação a incluir nos descriptores.

R:

A última coluna da tabela seguinte indica o escopo de cada uma das instruções no exemplo.

				Escopo
Exemplo		1.	void insertionSort(int[] num, int size) {	@params
		2.	int j;	@1
		3.	for (int i = 1; i < size; i++) {	@1.1
		4.	int key = num[i];	@1.1
		5.	j = i - 1;	@1.1
		6.	while((j>=0) && (key < num[j])) {	@1.1.1
		7.	num[j+1] = num[j];	@1.1.1
		8.	j--;	@1.1.1
		9.	}	@1.1.1
		10.	num[j+1] = key;	@1.1
		11.	}	@1.1
		12.	}	@1

Possível representação da tabela de símbolos:

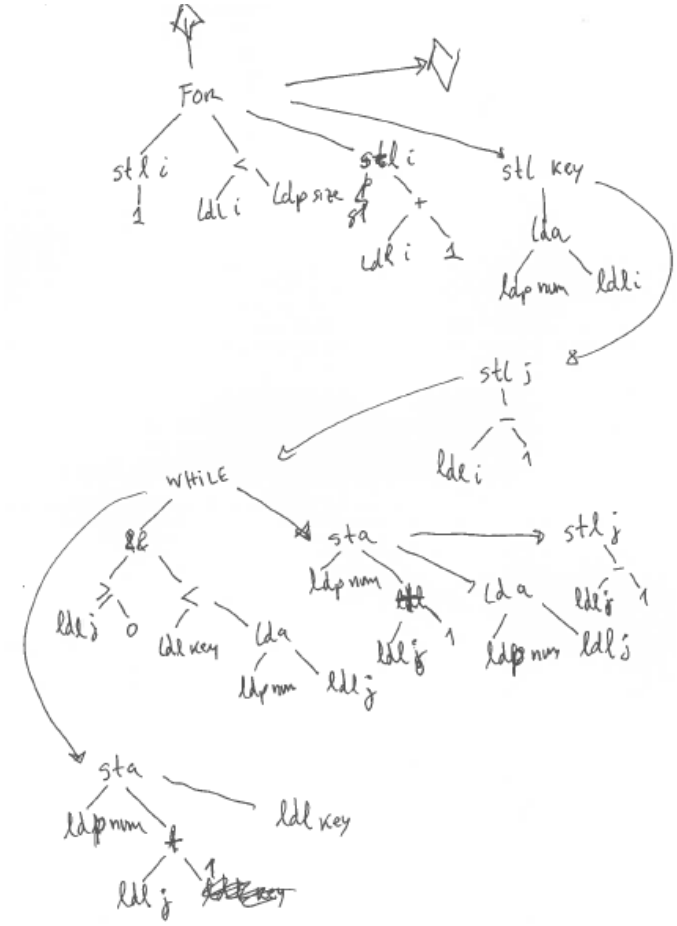


Nota: esta tabela não inclui uma tabela de símbolos para tipos e a hierarquia e aninhamento de escopos está representada pelos @'s.

■

2.b) [2v] Indique a representação intermédia de alto nível para o exemplo tendo como referência a representação intermédia baseada em árvores de espressões apresentada nos slides das aulas teóricas.

R:



■

2.c) [1v] Descreva possíveis verificações semânticas a ter em conta no código das linhas 6 a 9 e indique como poderiam ser realizadas essas verificações.

R:

Line	Código	Possíveis análises semânticas
6	while((j>=0) && (key < num[j])) {	j,key,num: verificar se foram declaradas j: verificar se está inicializado e se é uma variável com tipo que possa ser usado na expressão j>=0 e como índice de um array: num[j] key: verificar se está inicializada e se é uma variável com tipo que possa ser usado na key < num[j] num: verificar se é um array unidimensional e se os elementos são de um tipo que possam ser usados na expressão key < num[j]; no caso de ser um array com ais do que uma dimensão num[j] não poderá ser usado neste contexto.
7	num[j+1] = num[j];	j, num: verificar se foram declaradas j: verificar se está inicializado e se é uma variável com tipo que possa ser usado na expressão j+1 e como índice de um array: num[j] j+1: verificar se o tipo que resulta da expressão pode ser usado como índice de um array: num[j+1]

8	j--;	j: verificar se foi declarado j: verificar se está inicializado e se é uma variável com tipo que possa ser usado na expressão j--
9	}	s/ análise

Como poderiam ser realizadas:

- *Verificar declarações: consulta da tabela de símbolos*
- *Verificar Inicializações: percorrer a AST pela ordem do programa (assumindo a inexistência de gotos) e verificar que não existe nenhum caminho que permita que a variável não tenha sido atribuída antes da primeira instrução que a usa*
- *Verificar expressões: analisar expressões e verificar semântica de cast*

■

Grupo 3. Análise Semântica II (4 valores)

3.a) [2v] Explique se no contexto de linguagens de programação, o uso de gramáticas livres de contexto (CFGs[2]) permite incluir verificações semânticas a nível das regras da gramática.

R:
Por exemplo:
A inclusão de verificações semânticas em gramáticas livres de contexto (CFGs) é sempre muito limitada. Por exemplo, para incluir a verificação semântica no âmbito de expressões aritméticas a linguagem de programação teria de forçar os programadores a incluir o tipo no nome dos identificadores das variáveis. A verificação dos argumentos nas invocações de funções é uma tarefa impossível de realizar no contexto de linguagens em que as funções e os parâmetros das mesmas não são conhecidas na gramática.
A verificação relacionada com a inicialização de variáveis pode ser implementada via CFG ao forçar que a linguagem requeira a inicialização das variáveis aquando da sua declaração.

[excluir o uso de gramáticas de atributos por não terem sido focadas]
■

3.b) [2v] Indique exemplos de verificações semânticas que eventualmente só podem ser feitas em tempo de execução. Justifique sucintamente os exemplos indicados.

R:
Por exemplo:

- *Verificação se a indexação em arrays está sempre de acordo com o tamanho dos mesmos. Em alguns casos esta verificação poderia ser feita em tempo de compilação (e.g., quando é possível verificar os limites inferiors e superiores das expressões que definem os indices dos arrays com tamanho estaticamente conhecido), mas será sempre impossível para muitos outros casos dado que o tamanho do array pode ser apenas definido em tempo de execução, e/ou as expressões dos indíces nos arrays podem depender de valores apenas conhecidos em tempo de execução, ou podem não existir análises em tempo de compilação que possam avaliar os limites de valores que essas expressões podem ter.*
- *A verificação de tipos em linguagens dinâmicas pode ter de ser feita apenas em tempo de execução pois os tipos das variáveis podem não ser possíveis de inferior em tempo de compilação e apenas serem conhecidos durante a execução do programa.*

■

(Fim.)

[2] Do Inglês: Context-Free Grammars.