Laboratório 09

Q1

Front-end

1. Entrada do código vai para o analisador léxico, que lê uma sequência de caracteres e gera tokens no formato ‘<token, atributo>’.

2. Os tokens vão para o analisador sintático, que cria uma representação tipo árvore que mostra a estrutura gramatical da sequência de tokens.

3. A árvore vai para o analisador semântico, que utiliza a árvore de sintaxe e a tabela de símbolos para verificar a consistência semântica do programa e a conversão de tipos. A árvore é então, corrigida.

4. A árvore corrigida vai para o gerador de código intermediário, que gera uma representação intermediária de baixo nível para uma máquina abstrata.

Back-end

1. A representação intermediária vai para o otimizador de código independente de máquina, que faz algumas transformações com o objetivo de produzir uma representação intermediária otimizada.

2. A representação intermediária otimizada vai para o gerador de código, que mapeia a representação intermediária em código de máquina de alguma arquitetura.

3. O código de máquina vai para o otimizador de código dependente de máquina, que faz algumas transformações com o objetivo de produzir um código de máquina melhor.

4. O código de máquina otimizado vai para o Hardware.

Q2

É um método de implementação que usa um analisador (Parser) para traduzir a linguagem.

Q3

A gramática é formada por símbolos terminais e não-terminais. Os símbolos não-terminais podem se transformar em outros símbolos não-terminais e/ou podem se transformar em símbolos terminais.

Ex:
expr → expr + term
→ expr – term
| term

term → 0 | … | 9

Os números de 0 até 9 são terminais. Não se transformam em outros símbolos. ‘expr’ e ‘term’ são não-terminais. Se transformam em outros símbolos.

Q4

program → **class Name** block

block → **{ private:** variables **public:** methods **}**

variables → pvt variables
| **ϵ**

methods → pub methods
| **ϵ**

pvt → **type id;**

pub → **Method();**

Q5

1. expr
2. term
3. fact
4. (expr)
5. (expr – term)
6. (term – term)
7. (fact – term)
8. ((expr) – term)
9. ((expr + term) – term)
10. ((term + term) – term)
11. ((term * fact + term) – term)
12. ((fact * fact + term) – term)
13. ((digi * fact + term) – term)
14. ((5 * fact + term) – term)
15. ((5 * digi + term) – term)
16. ((5 * 2 + term) – term)
17. ((5 * 2 + fact) – term)

18. ((5 * 2 + digi) – term)

19. ((5 * 2 + 6) – term)

20. ((5 * 2 + 6) – term / fact)

21. ((5 * 2 + 6) – fact / fact)

22. ((5 * 2 + 6) – digi / fact)

23. ((5 * 2 + 6) – 9 / fact)

24. ((5 * 2 + 6) – 9 / digi)

25. ((5 * 2 + 6) – 9 / 7)

Q6

Esquerda:	expr	→	expr + term
		|	expr – term
		|	term

Direita:	expr	→	term + expr
		|	term – expr
		|	term

Q7

Através de uma árvore de derivação é possível fazer isso. Geralmente a análise da árvore é feita da esquerda para a direita. Então os nós da esquerda são lidos e juntados com o símbolo da direita e o operando no meio desses nós.

Q8

É a descrição da tradução que mostra uma prévia da implementação na gramática. Geralmente a instrução ‘print’ é usada para mostrar onde mostrar um char/string e em qual parte da gramática.

Q9

```
expr   →   { print('+') } expr + term
       |   { print('-') } expr – term
       |   term

term   →   { print('*') } term * fact
       |   { print('/') } term / fact
       |   fact

fact   →   digi
       |   (expr)

digi   →   0 { print('0') } | ... | 9 { print('9') }
```

Q10

```
expr   →   term oper

oper   →   { print('+') } + term oper
       |   { print('-') } – term oper
       |   term

term   →   fact mult

mult   →   { print('*') } * fact mult
       |   { print('/') } / fact mult
       |   fact

fact   →   digi
       |   (expr)

digi   →   0 { print('0') } | ... | 9 { print('9') }
```