prof. Éfren L. Souza

UFOPA – Universidade Federal do Oeste do Pará
IEG – Instituto de Engenharia e Geociências
PC – Programa de Computação
Disciplina – Compiladores

Trabalho I

Dadas as seguintes gramáticas:

1) Gramática que gera expressões regulares, com as operações de união (**+**), concatenação e fecho de Kleene (*****).

```
UNION ::= UNION + CONCAT | CONCAT
CONCAT ::= CONCAT KLEENE  | KLEENE
KLEENE  ::= KLEENE  * | ID
ID ::= a | b | ... | z
```

2) Gramática que gera expressões aritméticas que incluem: [i] operadores binários de soma (**+**), subtração (**–**), multiplicação (*****) e divisão (**/**); [ii] operadores unários de soma (**+**) e subtração (**–**); e [iii] o operador binário de potenciação (**^**), que é associativo à direita. As expressões podem conter parênteses e os operandos são letras ou dígitos.

```
EXPR ::= EXPR + TERM | EXPR - TERM | TERM
TERM ::= TERM * UNARY | TERM / UNARY | UNARY
UNARY ::= + UNARY | - UNARY | POW
POW ::= FACTOR ^ UNARY | FACTOR
FACTOR ::= (EXPR) | ID | DIGIT
ID ::= a | b | ... | z
DIGIT ::= 0 | 1 | ... | 9
```

3) Gramática que gera expressões aritméticas que incluem: [i] operadores binários de soma (**+**), subtração (**–**), multiplicação (*****) e divisão (**/**); [ii] operadores unários de soma (**+**) e subtração (**–**); [iii] chamadas para funções que podem ou não ter argumentos; e [iv] arranjos cujos índices vêm entre colchetes. As expressões podem conter parênteses e os operandos são letras ou dígitos.

```
EXPR ::= EXPR + TERM | EXPR - TERM | TERM
TERM ::= TERM * UNARY | TERM / UNARY | UNARY
```

```
UNARY ::= + UNARY | - UNARY | FACTOR
FACTOR ::= (EXPR) | DIGIT | ID | ID[EXPR] | ID() |
                ID(ARGS)
ARGS ::= ARGS, ARG | ARG
ARG ::= EXPR
ID ::= a | b | ... | z
DIGIT ::= 0 | 1 | ... | 9
```

4) Gramática que gera expressões aritméticas que incluem: [i] operadores binários de soma (**+**), subtração (**−**), multiplicação (*****) e divisão (**/**); [ii] operadores unários de soma (**+**) e subtração (**−**); [iii] operadores de pré-decremento (**<**) e pré-incremento (**>**); e [iv] operadores de pós-decremento (**<**) e pós incremento (**>**). As expressões podem conter parênteses e os operandos são letras ou dígitos.

```
EXPR ::= EXPR + TERM | EXPR - TERM | TERM
TERM ::= TERM * UNARY | TERM / UNARY | UNARY
UNARY ::= + UNARY | - UNARY | < ID | > ID | POST
POST ::= ID > | ID < | FACTOR
FACTOR ::= (EXPR) | ID | DIGIT
ID ::= a | b | ... | z
DIGIT ::= 0 | 1 | ... | 9
```

5) Gramática que gera atribuições. O lado esquerdo da atribuição pode ser uma variável escalar ou um arranjo. Os operadores de atribuição podem ser com incremento (**>**), com decremento (**<**) ou simples (**=**). O lado direito da atribuição é uma expressão aritméticas entre parênteses, contendo: [i] operadores binários de soma (**+**), subtração (**−**), multiplicação (*****) e divisão (**/**); [ii] operadores unários de soma (**+**) e subtração (**−**). A atribuição simples pode ser aplicada para múltiplas variáveis.

```
ASSIGN ::= LEFT > (EXPR) | LEFT < (EXPR) | LEFT= REST
LEFT   ::= ID | ID[EXPR]
REST   ::= LEFT = REST | (EXPR)
EXPR   ::= EXPR + TERM | EXPR - TERM | TERM
TERM ::= TERM * UNARY | TERM / UNARY | UNARY
UNARY ::= + UNARY | - UNARY | FACTOR
FACTOR ::= (EXPR) | DIGIT | LEFT
ID ::= a | b | ... | z
DIGIT ::= 0 | 1 | ... | 9
```

6) Gramática que gera expressões lógicas e aritméticas que incluem: [i] os conectores lógicos *or* (**|**) e *and* (**&**); [ii] os operadores de igualdade (**=**) e diferença (**~**); [iii] os operadores relacionais menor (**<**) e maior (**>**); [iv] operadores binários de soma (**+**),

subtração (**-**), multiplicação (*****) e divisão (**/**); [v] operadores unários de soma (**+**), subtração (**-**) e negação(**!**). As expressões podem conter parênteses. Os operandos são letras (**a**, **b**, **c**); dígitos; ou as letras **t** e **f**, que representam respectivamente os literais lógicos *true* e *false*.

```
BOOL ::= BOOL '|' JOIN | JOIN
JOIN ::= JOIN & EQUAL | EQUAL
EQUAL ::= EQUAL = REL | EQUAL ~ REL | REL
REL ::= EXPR < EXPR | EXPR > EXPR | EXPR
EXPR ::= EXPR + TERM | EXPR - TERM | TERM
TERM ::= TERM * UNARY | TERM / UNARY | UNARY
UNARY ::= ! UNARY | - UNARY | + UNARY | FACTOR
FACTOR ::= (BOOL) | ID | DIGIT | LIT_BOOL
ID ::= a | b | c
DIGIT ::= 0 | 1 | ... | 9
LIT_BOOL ::= t | f
```

A ideia desta atividade é desenvolver um *parser* simplificado a fim de consolidar os conceitos sobre análise sintática. Todos os terminais das gramáticas dadas são compostos por um único caractere, isso simplifica tanto o *lexer* quanto o *parser*. Para isso, alguns operadores tradicionais tiveram que ser adaptados para ter um único caractere. Por exemplo, o operador de diferença que comumente é **!=** foi adaptado para ser **~**.

Cada equipe ficará responsável por apenas uma das gramáticas. As seguintes tarefas devem ser feitas para a gramática:

a) Ajustar a gramática para torná-la LL(1);
b) Apresentar 5 *strings* que são aceitas pela gramática, sendo que juntas elas devem utilizar todas as regras da gramática;
c) Mostrar a árvore de derivação para cada uma das *strings* do item anterior.

Depois disso, crie uma classe Java que implementa a interface abaixo para verificar a sintaxe da linguagem gerada pela gramática. Essa verificação de sintaxe deve ser feita por um algoritmo de Análise Descendente Preditiva.

```
public interface IParser {
    /*
     * Constante que simboliza que o consumo da string terminou
     */
     public static final char EOF = (char)-1;

    /*
     * Método que retorna o token atual. Ele retorna EOF caso
```

```
     * a string já tenha sido toda consumida.
     */
     public char lookahead();

    /*
     * Faz o papel de lexer. A cada chamada retorna o próximo
     * caractere ("token") que não é um espaço em branco.
     */
     public char next();

    /*
     * Verifica se o lookahead combina com um dado char. Ele
     * avança para o próximo caractere caso combine, caso
     * contrário imprime um erro.
     */
     public void match(char c); {

    /*
     * Imprime uma mensagem de erro, indicando a coluna onde o
     * erro ocorreu.
     */
     public void error(String msg);

    /*
     * Método que verifica a sintaxe de uma dada string,
     * retornando true caso ela seja aceita. Esse chama o método
     * que representa o não-terminal inicial da gramática.
     */
     public boolean parse(String string);
}
```

A *string* passada para o método *parse* ficará armazenada em um atributo do tipo *string*, cada caractere dessa *string* corresponde a um *token* (exceto os espaços em branco que podem estar acumulados em quaisquer posições).

Para manter o controle sobre o consumo da *string*, será usado um atributo do tipo inteiro para registrar a posição do caractere atual (*lookahead*). Lembre-se que toda a entrada deve ser consumida para que a sintaxe esteja correta. Caso algum erro de sintaxe seja detectado, o parser deverá apresentar uma mensagem de erro, indicando a coluna da *string* onde o erro ocorreu.

Você deve entregar o código fonte do parser e um arquivo PDF contendo a gramática ajustada e os exemplos de entradas válidas com suas respectivas árvores de derivação. O código deve ter um arquivo README explicando como executá-lo.