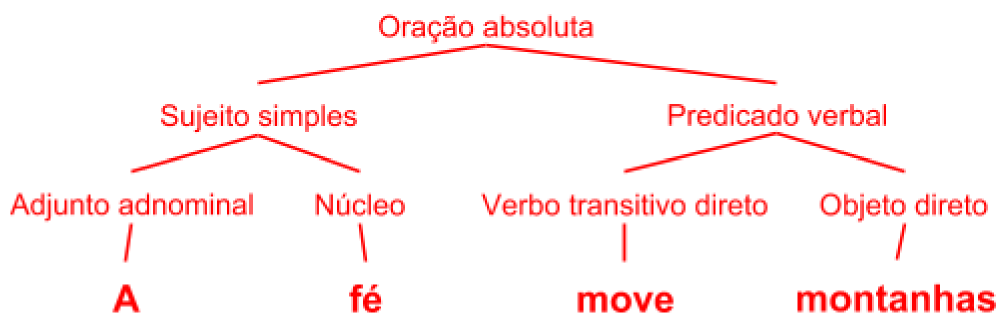
Construção de Compiladores 1 - 2018.1 - Profs. Mário César San Felice
(e Helena Caseli, Murilo Naldi, Daniel Lucrédio)
Tópico 03 - Introdução à Análise Sintática - Lista de Exercícios Resolvida

1. Qual é a função da etapa de análise sintática no processo de tradução de um compilador?

   R: Compreender a estrutura de um programa fonte. Procura detectar se a forma com que as unidades léxicas (tokens) se compõem está correta. Olha apenas a forma, e não o significado.

2. Faça a análise sintática do seguinte ditado: “A fé move montanhas” (pesquise um pouco de gramática da língua portuguesa)

   R:



3. Qual a importância da teoria da computação (linguagens e autômatos) para a análise sintática?

   R: Definir uma forma mais eficiente para descrever linguagens (gramáticas livres de contexto) e implementar parsers ou analisadores sintáticos (autômatos com pilha). Sem esse formalismo, o trabalho deve ser feito de forma manual, o que torna mais difícil a criação e manutenção dos compiladores.

4. Porque utiliza-se gramáticas livres de contexto em compiladores, para análise sintática, e não outras classes de gramáticas?

   R: Porque elas são simples o suficiente para serem concebidas e manipuladas. São adequadas para descrever a maioria das construções das linguagens utilizadas na prática. Além disso, permitem fácil implementação (autômatos com pilha). Outras classes não são adequadas. Linguagens regulares são simples demais, não permitindo recursividade. Linguagens sensíveis a contexto ou recursivamente enumeráveis são de difícil implementação (máquina de turing).

5. Qual a diferença entre uma árvore de análise sintática e uma árvore de sintaxe abstrata?

   R: A árvore de análise sintática representa exatamente o processo de análise. Cada nó-folha da árvore é um terminal da gramática, e cada nó-interior é um não-terminal da gramática. O mapeamento é um-para-um. Já a árvore de sintaxe abstrata contém somente as informações necessárias para a análise semântica.

6. Defina uma gramática não ambígua para expressões aritméticas onde os operandos são constantes numéricas e variáveis. Os operadores são os operadores aritméticos comuns: *,/,+,-,

além do exponencial ^. A precedência e associatividade dos operadores, porém, é o contrário do usual, ou seja: + e – tem precedência sobre * e / que tem precedência sobre ^. Todos os operadores são associativos à direita, com exceção do exponencial "^" que é associativo à esquerda.

R:
```
expr : expr op1 termo | termo;
termo : fator op2 termo | fator;
fator : elemento op3 fator | elemento;
elemento: NUM | VAR;
op1 : '^';
op2: '*' | '/';
op3: '+' | '-';
```

7. Modifique a gramática do exercício anterior de forma que expoentes só possam envolver constantes numéricas, e nunca variáveis

```
expr : expr op1 termoSemVar | termo;
termo : fator op2 termo | fator;
fator : elemento op3 fator | elemento;
elemento: NUM | VAR;
termoSemVar : fatorSemVar op2 termoSemVar | fatorSemVar;
fatorSemVar : elementoSemVar op3 fatorSemVar | elementoSemVar;
elementoSemVar: NUM;
op1 : '^';
op2: '*' | '/';
op3: '+' | '-';
```

8. Considere a seguinte gramática

```
E : E op E | (E) | NUM
```

Modifique essa gramática para que ela passe a utilizar uma notação pré-fixada, ou seja, ao invés de escrever 2 + 3 * 4, escreveríamos + 2 * 3 4. A gramática resultante precisa de parêntesis? A gramática resultante é ambígua? Justifique suas respostas.

R:
```
E : op E E | NUM
```
Não é mais necessário parêntesis, pois agora é possível agrupar as expressões simplesmente pela ordem dos operadores. Também não há mais ambiguidade, pois não há recursividade dupla.

9. Considere as seguintes regras gramaticais:

```
comando : comando ';' listaComandos
comando : comando ';' 'if' '(' expr ')' 'then' comando
expr : expr '+' termo | expr '-' termo | termo
termo : termo '*' fator | termo '/' fator | fator
fator : fator '.' VAR | VAR
comando : 'while' '(' expr ')' comando 'endwhile'
listacomandos : '{' comando '}'
```

Elimine toda recursividade à esquerda

R:

```
comando : 'while' '(' expr ')' comando 'endwhile' comando2
comando2 : ';' listaComandos comando2 | ';' 'if' '(' expr ')' 'then'
            comando | <<vazio>>
expr : termo expr2
expr2 : '+' termo expr2 | '-' termo expr2 | <<vazio>>
termo : fator termo2
termo2 : '*' fator termo2 | '/' fator termo2 | <<vazio>>
fator : VAR fator2
fator2: '.' VAR fator2 | <<vazio>>
listacomandos : '{' comando '}'
```

10. Considere as seguintes regras gramaticais

```
comandoCondicao
  :  'SE' expressaoRelacional 'ENTAO' comando 'FIMSE'
  |  'SE' expressaoRelacional 'ENTAO' comando 'SENAO' comando 'FIMSE'
  ;
```

Fatore à esquerda, caso necessário.

R:

```
comandoCondicao : 'SE' expressaoRelacional 'ENTAO' comando
            comandoSenao 'FIMSE'
comandoSenao : 'SENAO' comando | <<vazio>>
```

11. Considere as seguintes regras gramaticais

```
lexp : atomo | lista
atomo : numero | identificador
lista : ( lexpseq )
lexpseq : lexpseq lexp | lexp
```

Remova a recursividade à esquerda e modifique as regras de forma a empregar EBNF

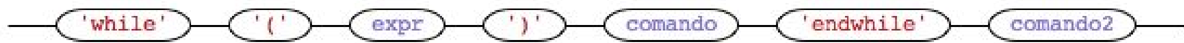
R:

```
lexp : atomo | lista
atomo : numero | identificador
lista : ( lexp+ )
```

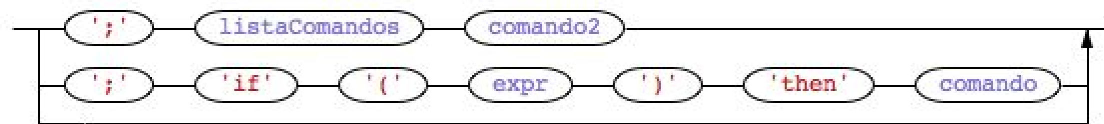
12. Desenhe os diagramas sintáticos para as regras da gramática dos exercícios 9, 10 e 11 (com e sem EBNF)

R - Ex 9:

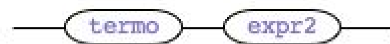
comando



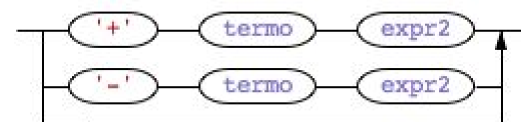
comando2



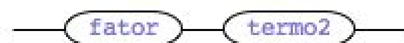
expr



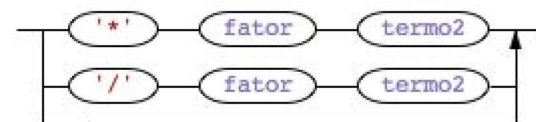
expr2



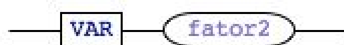
termo



termo2



fator



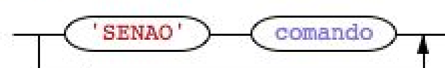
fator2

listaComandos

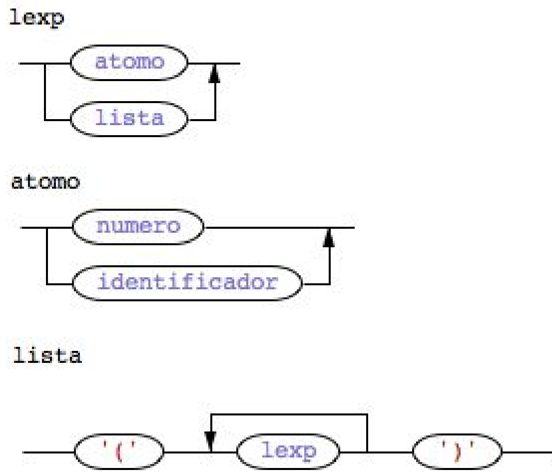
R - Ex 10:

comandoCondicao

comandoSenao



R - Ex 11:



13. Dada a gramática a seguir

```
S : ( L ) | a
L : L , S | S
```

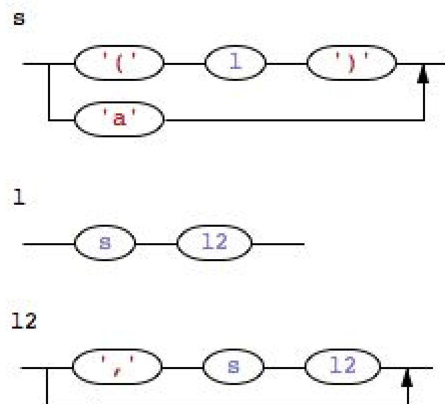
a) Elimine a recursividade à esquerda.

R:

```
S : '(' L ')' | 'a'
L : S L2
L2 : ',' S L2 | <<vazio>>
```

b) Desenhe os grafos sintáticos correspondentes.

R: (obs: renomeei para letras minúsculas para poder usar o ANTLR para gerar os grafos sintáticos)



c) Construa os procedimentos recursivos para os grafos sintáticos construídos na letra b) bem como o programa principal, utilizando pseudo-código.

R:

```
void s() {
   if(la(1) == '(') {
      match('(')
```

```
      l()
      match(')')
   } else {
      match('a')
   }
}

void l() {
   s()
   l2()
}

void l2() {
   if(la(1) == ',') {
      match(',')
      s()
      l2()
   } else {
      // vazio
   }
}

void principal() {
   s()
}
```

14. Construa os procedimentos recursivos e o programa principal, usando pseudo-código, da gramática do exercício 9 (já sem recursividade à esquerda)

R:

```
void comando() {
   match('while')
   match('(')
   expr()
   match(')')
   comando()
   match('endwhile')
   comando2()
}

void comando2() {
   match(';') // o ponto e vírgula vem antes do teste, pois
              // é comum a ambas as alternativas
              // outra solução seria fatorar a gramática
   if(la(1) == '{') {
      listaComandos()
      comando2()
   } else if(la(1) == 'if') {
      match('if')
      match('(')
```

```
        expr()
        match(')')
        match('then')
        comando()
    } else {
        // vazio
    }
}
void expr() {
    termo()
    expr2()
}
void expr2() {
    if(la(1) == '+') {
        match('+')
        termo()
        expr2()
    } else if(la(1) == '-') {
        match('-')
        termo()
        expr2()
    } else {
        // vazio
    }
}
void termo() {
    fator()
    termo2()
}
void termo2() {
    if(la(1) == '*') {
        match('*')
        fator()
        termo2()
    } else if(la(1) == '/') {
        match('/')
        fator()
        termo2()
    } else {
        // vazio
    }
}
void fator() {
    match(TipoToken.VAR)
    fator2()
}
void fator2() {
    if(la(1) == '.') {
        match('.')
        match(TipoToken.VAR)
        fator2()
```

```
    } else {
        // vazio
    }
}
listacomandos : '{' comando '}'
void listaComandos() {
    match('{')
    comando()
    match('}')
}
void principal() {
    comando()
}
```

15. Construa os procedimentos recursivos e o programa principal, usando pseudo-código, da gramática do exercício 10 (já fatorada à esquerda)

R:

```
void comandoCondicao() {
    match('SE')
    expressaoRelacional()
    match('ENTAO')
    comando()
    comandoSenao()
    match('FIMSE')
}
void comandoSenao() {
    if(la(1) == 'SENAO') {
        match('SENAO')
        comando()
    } else {
        // vazio
    }
}
void principal() {
    comandoCondicao()
}
```

16. Construa os procedimentos recursivos e o programa principal, usando pseudo-código, da gramática do exercício 11 (já sem recursividade à esquerda e usando EBNF)

R:

```
void lexp() {
    if(la(1)==TipoToken.numero | la(1)==TipoToken.identificador) {
        atomo()
    } else if(la(1) == '(') {
        lista()
    }
}
void atomo() {
    if(la(1) == TipoToken.numero) {
```

```
        match(TipoToken.numero)
    } else {
        match(TipoToken.identificador)
    }
}
void lista() {
    match('(')
    while(la(1) != ')') {
        lexp()
    }
    match(')')
}
void principal() {
    lexp()
}
```