

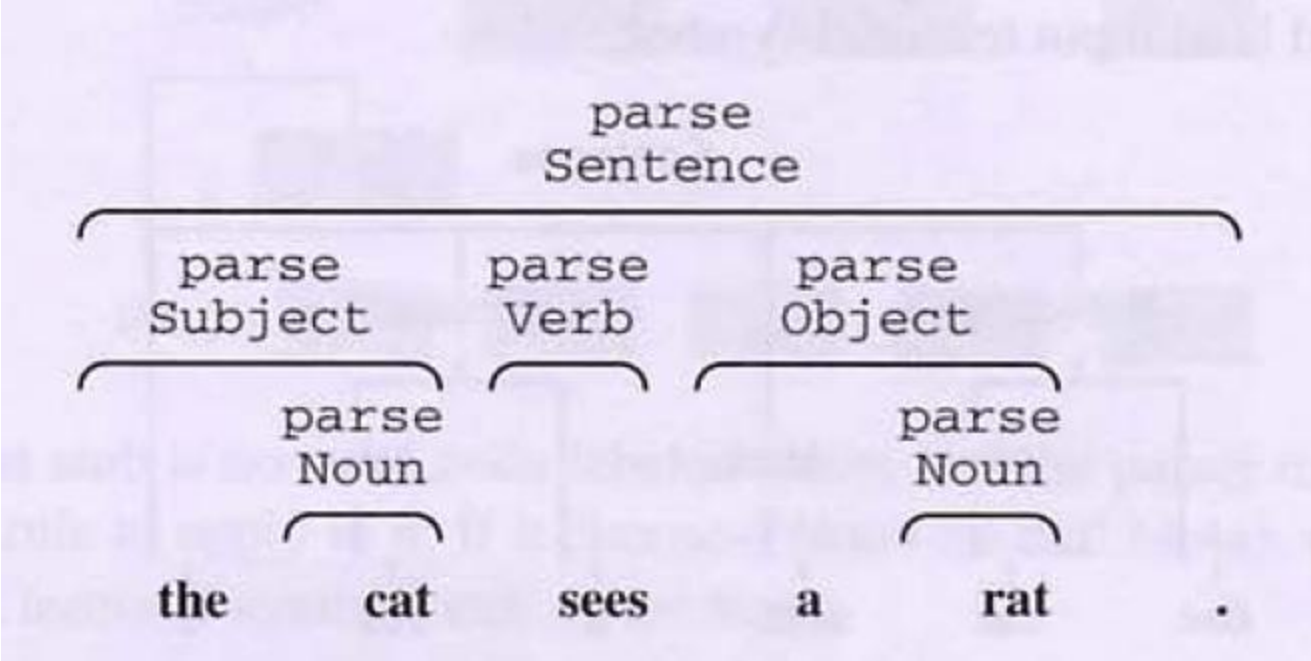
Sentence ::= Subject Verb Object .

Subject ::= **I** | a Noun | **the** Noun

Object ::= **me** | a Noun | **the** Noun

Noun ::= **cat** | **mat** | **rat**

Verb ::= **like** | **is** | **see** | **sees**



```
private void parseNoun ();  
// Parse a noun, i.e., 'cat', 'mat', or 'rat'.  
  
private void parseVerb ();  
// Parse a verb, e.g., 'like' or 'sees'.  
  
private void parseSubject ();  
// Parse a subject, e.g., 'I' or 'a rat'.  
  
private void parseObject ();  
// Parse an object, e.g., 'me' or 'a rat'.  
  
private void parseSentence ();  
// Parse a complete sentence.
```

```
public class Parser {  
    private TerminalSymbol currentTerminal;  
    ... // Auxiliary methods will go here.  
    ... // Parsing methods will go here.  
}
```

```
private void accept (TerminalSymbol expectedTerminal) {  
    if (currentTerminal matches expectedTerminal)  
        currentTerminal = next input terminal;  
    else  
        report a syntactic error2  
}
```

```
private void parseSentence () {  
    parseSubject();  
    parseVerb();  
    parseObject();  
    accept('.');  
}
```

```
Sentence ::=  
    Subject  
    Verb  
    Object  
    .
```

```

private void parseSubject () {
    if (currentTerminal matches 'I')
        accept ('I');
    else
        if (currentTerminal matches 'a') {
            accept ('a');
            parseNoun ();
        } else
            if (currentTerminal matches 'the') {
                accept ('the');
                parseNoun ();
            } else
                report a syntactic error
}

```

Subject ::=

I

|

a
Noun

|

the
Noun

```

private void parseNoun () {
    if (currentTerminal matches 'cat')
        accept ('cat');
    else
        if (currentTerminal matches 'mat')
            accept ('mat');
        else
            if (currentTerminal matches 'rat')
                accept ('rat');
            else
                report a syntactic error
}

```

Noun ::=
 cat
 |
 mat
 |
 rat

```
public void parse () {  
    currentTerminal = first input terminal ;  
    parseSentence() ;  
    check that no terminal follows the sentence  
}
```

Dada uma gramática GLC G :

- Obter G' tal que $L(G')=L(G)$ e G' seja LL(1);
- Conforme a conveniência, efetuar eliminação de regras e de recursões à direita, usando a notação EBNF;
- Criar, para cada símbolo não-terminal N um método `parseN()` {...} cujo corpo será determinado pela definição do mesmo;
- Criar uma classe `Parser` composta pela variável `currentToken` (símbolo corrente), métodos auxiliares e métodos `parse`; todos privados;
- Criar um método público `parse` para inicializar a operação e servir como interface para outros módulos do compilador.

```
Program      ::= single-Command
Command      ::= single-Command
              | Command ; single-Command
single-Command ::= V-name := Expression
              | Identifier ( Expression )
              | if Expression then single-Command
                else single-Command
              | while Expression do single-Command
              | let Declaration in single-Command
              | begin Command end
```

Expression	::=	primary-Expression Expression Operator primary-Expression
primary-Expression	::=	Integer-Literal V-name Operator primary-Expression (Expression)
V-name	::=	Identifier
Declaration	::=	single-Declaration Declaration ; single-Declaration
single-Declaration	::=	const Identifier ~ Expression var Identifier : Type-denoter
Type-denoter	::=	Identifier
Operator	::=	+ - * / < > = \
Identifier	::=	Letter Identifier Letter Identifier Digit
Integer-Literal	::=	Digit Integer-Literal Digit
Comment	::=	! Graphic* eof

Program ::= single-Command
 Command ::= single-Command (; single-Command) *
 single-Command ::= Identifier (:= Expression | (Expression))
 | **if** Expression **then** single-Command
 | **else** single-Command
 | **while** Expression **do** single-Command
 | **let** Declaration **in** single-Command
 | **begin** Command **end**
 Expression ::= primary-Expression
 (Operator primary-Expression) *
 primary-Expression ::= Integer-Literal
 | Identifier
 | Operator primary-Expression
 | (Expression)
 Declaration ::= single-Declaration (; single-Declaration) *
 single-Declaration ::= **const** Identifier ~ Expression
 | **var** Identifier : Type-denoter
 Type-denoter ::= Identifier

```
private void parseProgram ();  
private void parseCommand ();  
private void parseSingleCommand ();  
private void parseExpression ();  
private void parsePrimaryExpression ();  
private void parseDeclaration ();  
private void parseSingleDeclaration ();  
private void parseTypeDenoter ();  
private void parseIdentifier ();  
private void parseIntegerLiteral ();  
private void parseOperator ();
```

```

private void parseSingleDeclaration () {
    switch (currentToken.kind) {          single-Declaration ::=

case Token.CONST:
    {
        acceptIt();                    const
        parseIdentifier();              Identifier
        accept(Token.IS);                ~
        parseExpression();              Expression
    }
    break;

case Token.VAR:                          |
    {
        acceptIt();                    var
        parseIdentifier();              Identifier
        accept(Token.COLON);            :
        parseTypeDenoter();            Type-denoter
    }
    break;

default:
    report a syntactic error
    }
}

```

```
private void parseCommand () {  
    parseSingleCommand();  
    while (currentToken.kind  
           == Token.SEMICOLON)  
    {  
        acceptIt();  
        parseSingleCommand();  
    }  
}
```

```
Command ::=  
    single-Command  
  
    (  
        ;  
        single-Command  
    )*
```

```
private void parseProgram () {  
    parseSingleCommand();  
}
```

```
Program ::=  
    single-Command
```

```

private void parseSingleCommand () { single-Command ::=
    switch (currentToken.kind) {
    case Token.IDENTIFIER:
        {
            parseIdentifier();           Identifier
            switch (currentToken.kind) { (
            case Token.BECOMES:
                {
                    acceptIt();           ::=
                    parseExpression();    Expression
                }
                break;
            case Token.LPAREN:           |
                {
                    acceptIt();           (
                    parseExpression();    Expression
                    accept (Token.RPAREN); )
                }
                break;
            default:
                report a syntactic error
        }
    }
}
break;

```

```

case Token.IF: |
{
    acceptIt();           if
    parseExpression();    Expression
    accept(Token.THEN);   then
    parseSingleCommand(); single-Command
    accept(Token.ELSE);   else
    parseSingleCommand(); single-Command
}
break;

case Token.WHILE: |
{
    acceptIt();           while
    parseExpression();    Expression
    accept(Token.DO);     do
    parseSingleCommand(); single-Command
}
break;

case Token.LET: |
{
    acceptIt();           let
    parseDeclaration();   Declaration
    accept(Token.IN);     in
    parseSingleCommand(); single-Command
}
break;

```

```
case Token.BEGIN: |
{
    acceptIt();
    parseCommand();
    accept(Token.END);
}
break;
default:
    report a syntactic error
}
}
```

```
public class Parser {  
    private Token currentToken;  
    private void accept (byte expectedKind) {  
        if (currentToken.kind == expectedKind)  
            currentToken = scanner.scan();  
        else  
            report a syntactic error  
    }  
    private void acceptIt () {  
        currentToken = scanner.scan();  
    }  
    ... // Parsing methods, as above.  
    public void parse () {  
        currentToken = scanner.scan();  
        parseProgram();  
        if (currentToken.kind != Token.EOT)  
            report a syntactic error  
    }  
}
```

Exercício

Considere a linguagem definida pela gramática:

$$R \rightarrow SNE$$
$$S \rightarrow + \mid - \mid \varepsilon$$
$$N \rightarrow I.I \mid I. \mid .I$$
$$I \rightarrow dI \mid d$$
$$E \rightarrow eSI \mid \varepsilon$$

sobre o alfabeto $\Sigma = \{d, e, +, -, ., \varepsilon\}$.

Pede-se:

1. Verificar se a gramática é LL(1) e converter se necessário;
2. Um conjunto de métodos parse para essa linguagem;
3. Obter uma única expressão regular que gere essa linguagem;
4. Um método parse baseado nessa expressão regular.

$(+|-|\epsilon)(dd^*.dd^*|dd^*|.dd^*)(e(+|-|\epsilon)dd^*|\epsilon)$

$(+|-|\epsilon)(dd^*.d^*|.dd^*)(e(+|-|\epsilon)dd^*|\epsilon)$

```

void parseR() {
    switch cT {
        case '+':
        case '-': acceptIt();
                break;
    }
    switch cT {
        case 'd': acceptIt();
                    while cT='d' acceptIt();
                    accept ('.');
                    while cT='d' acceptIt();
                    break;
        case ".": acceptIt();
                    accept ('d');
                    while cT='d' acceptIt();
                    break;
        default: ERRO();
    }
    if cT='e' {
        acceptIt();
        switch cT {
            case '+':
            case '-': acceptIt();
                    break;
        }
        accept('d');
        while cT='d' acceptIt();
    }
    if cT!=EOF ERRO();
}

```

ANÁLISE LÉXICA

- 1. Relacionar os tokens da linguagem;**
- 2. Construir uma gramática léxica;**
- 3. Manipular a gramática léxica, para satisfazer a condição LL(1);**
- 4. Aplicar o método recursivo descendente;**
- 5. Reconhecer e classificar os tokens;**
- 6. Retornar um objeto da classe Token;**
- 7. Em caso de erro, consumir o caracter corrente e retornar o token ERRO;**
- 8. Consumir separadores antes de cada token;**
- 9. Retornar o token EOF quando atingir o final do arquivo;**
- 10. Embutir todos os detalhes de acesso ao arquivo no meio externo.**

Token ::= Identifier | Integer-Literal | Operator |
; | : | := | ~ | (|) | eol

Identifier ::= Letter | Identifier Letter | Identifier Digit

Integer-Literal ::= Digit | Integer-Literal Digit

Operator ::= + | - | * | / | < | > | = | \

Separator ::= Comment | space | eol

Comment ::= ! Graphic* eol

Token ::= Letter (Letter | Digit)* | Digit Digit* |
+ | - | * | / | < | > | = | \ |
; | : (= | ϵ) | ~ | (|) | eot

Separator ::= ! Graphic* eol | space | eol

Sintático	Léxico
class Parser	class Scanner
currentToken	currentChar
parse...()	scan...()
accept(...)	take(...)
acceptIt()	takeIt()

```

private byte scanToken () {           Token ::=
    switch (currentChar) {
    case 'a': case 'b': case 'c':
    ...      case 'y': case 'z':
        takeIt();                       Letter
        while (isLetter(currentChar)
               || isDigit(currentChar))
            takeIt();                   (Letter | Digit)*
        return Token.IDENTIFIER;

    case '0': case '1': case '2':
    case '3': case '4': case '5':
    case '6': case '7': case '8':
    case '9':                             |
        takeIt();                       Digit
        while (isDigit(currentChar))
            takeIt();                   Digit*
        return Token.INTLITERAL;

```

```

case '+': case '-': case '*':
case '/': case '<': case '>':
case '=': case '\\':
    takeIt();
    return Token.OPERATOR;

case ';':
    takeIt();
    return Token.SEMICOLON;

case ':':
    takeIt();
    if (currentChar == '=') {
        takeIt();
        return Token.BECOMES;
    }
    else
        return Token.COLON;

case '~':
    takeIt();
    return Token.IS;

case '(':
    takeIt();
    return Token.LPAREN;

```

```
case ')': | )
    takeIt();
    return Token.RPAREN;

case '\000': | eot
    return Token.EOT;

default:
    report a lexical error
}
}
```

```
private void scanSeparator () { Separator ::=
  switch (currentChar) {
  case '!': {
    takeIt();
    while (
      isGraphic(currentChar))
      takeIt();
      take('\n');
    }
    break;
```

```
case ' ': case '\n':
  takeIt();
  break;
}
}
```

```
public class Scanner {  
    private char currentChar = first source character;  
    // Kind and spelling of the current token:  
    private byte currentKind;  
    private StringBuffer currentSpelling;  
    private void take (char expectedChar) {  
        if (currentChar == expectedChar) {  
            currentSpelling.append(currentChar);  
            currentChar = next source character;  
        } else  
            report a lexical error  
    }  
    private void takeIt () {  
        currentSpelling.append(currentChar);  
        currentChar = next source character;  
    }  
}
```

```
private boolean isDigit (char c) {  
    ... // Returns true iff the character c is a digit.  
}  
  
private boolean isLetter (char c) {  
    ... // Returns true iff the character c is a letter.  
}  
  
private boolean isGraphic (char c) {  
    ... // Returns true iff the character c is a graphic.  
}
```

```
private byte scanToken () {
    ... // As above.
}

private void scanSeparator () {
    ... // As above.
}

public Token scan () {
    while (currentChar == '!'
           || currentChar == ' '
           || currentChar == '\n')
        scanSeparator();           Separator*
    currentSpelling =
        new StringBuffer("");
    currentKind = scanToken();     Token
    return new Token(currentKind,
                     currentSpelling.toString());
}
}
```

```
public class Token {  
  
    public byte kind;  
    public String spelling;  
  
    public Token (byte kind, String spelling) {  
        this.kind = kind; this.spelling = spelling;  
        // If kind is IDENTIFIER and spelling matches one  
        // of the keywords, change the token's kind accordingly:  
        if (kind == IDENTIFIER)  
            for (int k = BEGIN; k <= WHILE; k++)  
                if (spelling.equals(spellings[k])) {  
                    this.kind = k; break;  
                }  
    }  
}
```

```

// Constants denoting different kinds of token:
public final static byte
    IDENTIFIER = 0, INTLITERAL = 1, OPERATOR = 2,
    BEGIN = 3, CONST = 4, DO = 5, ELSE = 6, END = 7,
    IF = 8, IN = 9, LET = 10, THEN = 11, VAR = 12,
    WHILE = 13, SEMICOLON = 14, COLON = 15,

    BECOMES = 16, IS = 17, LPAREN = 18,
    RPAREN = 19, EOT = 20;

// Spellings of different kinds of token (must correspond to the
// token kinds above):
private final static String[] spellings = {
    "<identifier>", "<integer-literal>",
    "<operator>", "begin", "const", "do", "else",
    "end", "if", "in", "let", "then", "var",
    "while", ";", ":", ":=", "~", "(", ")" , "<eot>"
}
}

```

PROJETO

Observações gerais:

- A documentação deverá ser entregue sempre em versão impressa; a entrega da mesma em versão digital é opcional;
- Os arquivos-fonte dos programas deverão ser entregues apenas em formato digital; eles não deverão ser entregues em formato impresso;
- O projeto é incremental: todo material (documentação e arquivos) elaborado para uma fase pode e deve ser revisto, corrigido e melhorado para as etapas seguintes;
- Devem ser observados os prazos publicados na página da disciplina.

PROJETO

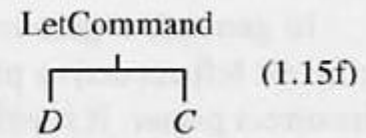
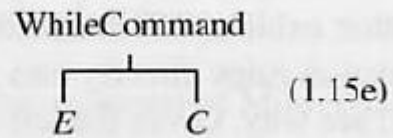
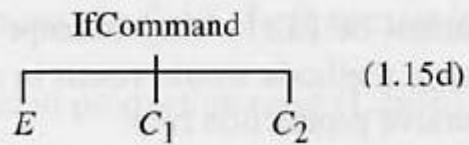
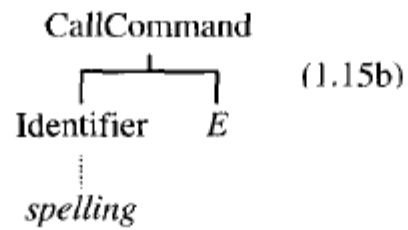
Etapa 1 - MANIPULAÇÃO GRAMATICAL

- Criar, a partir da gramática fornecida, uma relação dos tokens da linguagem.
- Verificar se a gramática da linguagem é LL(1). Justificar a sua resposta.
- Obter uma gramática equivalente que seja LL(1).
- Demonstrar, através do cálculo dos conjuntos first e follow, que a nova gramática é LL(1).
- Obter, a partir da nova gramática, uma gramática léxica e uma gramática sintática para a linguagem.

PROJETO

Etapa 2 - ANÁLISE SINTÁTICA

- Implementar, através do método recursivo descendente, um analisador léxico para a linguagem;
- Implementar, através do método recursivo descendente, um analisador sintático para a linguagem;
- Integrar os analisadores léxico e sintático;
- Projetar e implementar uma interface com o usuário (linha de comando ou janela);
- Desenvolver os casos de teste;
- Documentar o trabalho (sintaxe da linguagem-fonte, estrutura léxica, estrutura sintática, exemplos de programas-fonte, transformações gramaticais efetuadas, técnicas de análise empregadas, estruturas de dados e algoritmos utilizados, descrição da interface com o usuário, mensagens de erro emitidas, exemplos de entradas e saídas, testes efetuados, manual de instalação e manual de operação).



```
public abstract class AST {  
    ...  
}  
  
public abstract class Command extends AST { ... }  
  
public class AssignCommand extends Command {  
    public Vname V;           // left-side variable  
    public Expression E;     // right-side expression  
    ...  
}  
  
public class CallCommand extends Command {  
    public Identifier I;     // procedure name  
    public Expression E;    // actual parameter  
    ...  
}  
  
public class SequentialCommand extends Command {  
    public Command C1, C2;   // subcommands  
    ...  
}
```

```
public class IfCommand extends Command {  
    public Expression E;           // if condition  
    public Command C1, C2;       // true and false commands  
    ...  
}  
  
public class WhileCommand extends Command {  
    public Expression E;         // loop condition  
    public Command C;           // body of loop  
    ...  
}  
  
public class LetCommand extends Command {  
    public Declaration D;       // block declarations  
    public Command C;          // body of block  
    ...  
}
```

```
private  $AST_N$  parseN () {  
     $AST_N$  itsAST;  
    parse X, at the same time constructing itsAST  
    return itsAST;  
}
```

```
private Program          parseProgram ();
private Command         parseCommand ();
private Command         parseSingleCommand ();
private Expression      parseExpression ();
private Expression      parsePrimaryExpression ();
private Declaration     parseDeclaration ();
private Declaration     parseSingleDeclaration ();
private TypeDenoter     parseTypeDenoter ();
private Identifier      parseIdentifier ();
private IntegerLiteral  parseIntegerLiteral ();
private Operator       parseOperator ();
```

```
private Command parseSingleCommand () {  
    Command comAST;  
    switch (currentToken.kind) {  
  
    case Token.IDENTIFIER: {  
        Identifier iAST = parseIdentifier();  
        switch (currentToken.kind) {  
        case Token.BECOMES: {  
            acceptIt();  
            Expression eAST = parseExpression();  
            comAST = new AssignCommand(iAST, eAST);  
        }  
        break;  
    case Token.LPAREN: {  
        acceptIt();  
        Expression eAST = parseExpression();  
        accept(Token.RPAREN);  
        comAST = new CallCommand(iAST, eAST);  
    }  
    break;  
}
```

```

        default:
            report a syntactic error
        }
    }
    break;

case Token.IF:
    ...
case Token.WHILE:
    ...
case Token.LET: {
    acceptIt();
    Declaration dAST = parseDeclaration();
    accept(Token.IN);
    Command cAST = parseSingleCommand();
    comAST = new LetCommand(dAST, cAST);
}
break;

case Token.BEGIN: {
    acceptIt();
    comAST = parseCommand();
    accept(Token.END);
}
break;

default:
    report a syntactic error
}
return comAST;
}

```

```
public class Parser {  
    private Token currentToken;  
    ... // Auxiliary methods.  
    ... // Enhanced parsing methods, as above.  
    public Program parse () {  
        currentToken = scanner.scan();  
        Program progAST = parseProgram();  
        if (currentToken.kind != Token.EOT)  
            report a syntactic error  
        return progAST;  
    }  
}
```

PROJETO

Etapa 3: MONTAGEM DA AST

- Construir uma estrutura de dados que represente a estrutura sintática do programa-fonte (AST); para isso, deverão ser especificadas as classes abstratas e concretas que serão utilizadas para representar os nós da árvore. Depois, os métodos de análise sintática deverão ser adaptados para construir a árvore durante o fluxo de processamento do programa-fonte.
- O programa deverá prever uma opção que permita ao usuário visualizar a árvore depois de montada. Utilizar o padrão de projeto VISITOR.