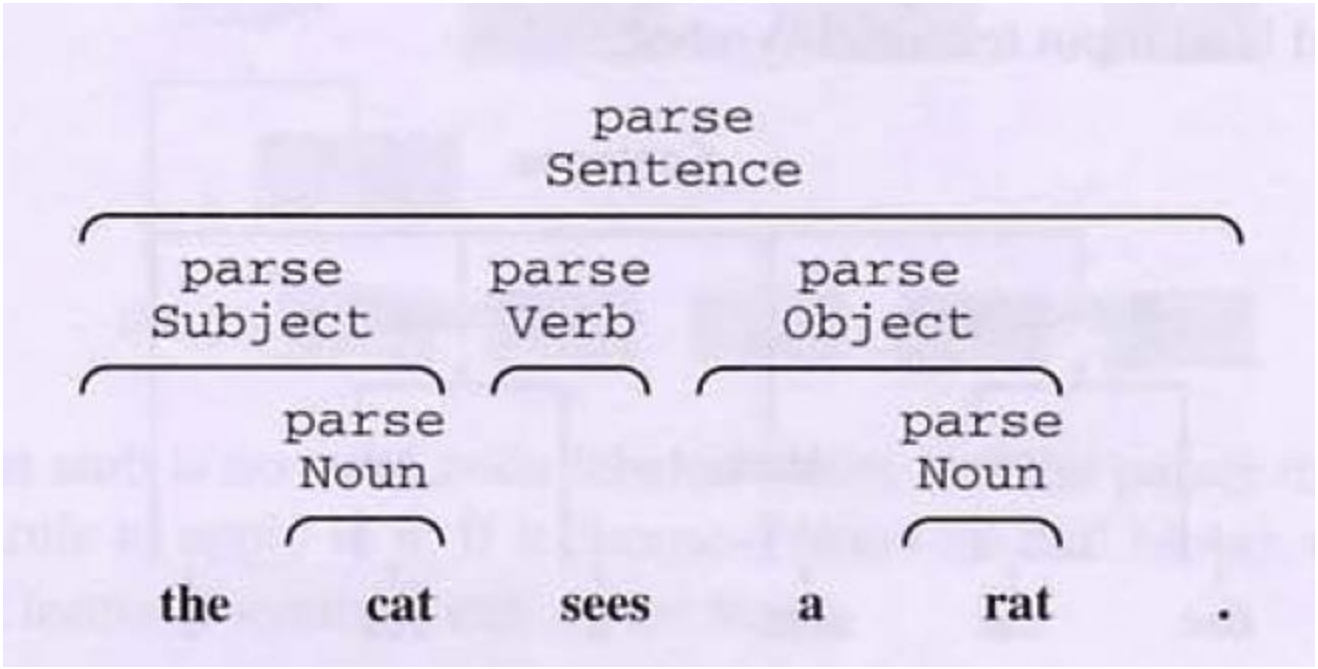
ANÁLISE SINTÁTICA MÉTODO RECURSIVO DESCENDENTE

Baseado no Capítulo 4 de Programming Language Processors in Java, de Watt & Brown

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**


parse
Sentence
parse
Subject
parse
Verb
parse
Object
parse
Noun
parse
Noun
the
cat
sees
a
rat
.

```
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 error[2]
}
```

```
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 privativos;
- 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* eol

```
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) {
    case Token.CONST:
        {
            acceptIt();
            parseIdentifier();
            accept(Token.IS);
            parseExpression();
        }
        break;

    case Token.VAR:
        {
            acceptIt();
            parseIdentifier();
            accept(Token.COLON);
            parseTypeDenoter();
        }
        break;

    default:
        report a syntactic error
    }
}
```

single-Declaration ::=

const
Identifier
~
Expression

|

var
Identifier
:
Type-denoter

```
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 () {
    switch (currentToken.kind) {

    case Token.IDENTIFIER:
        {
            parseIdentifier();
            switch (currentToken.kind) {
            case Token.BECOMES:
                {
                    acceptIt();
                    parseExpression();
                }
                break;

            case Token.LPAREN:
                {
                    acceptIt();
                    parseExpression();
                    accept(Token.RPAREN);
                }
                break;

            default:
                report a syntactic error
            }
        }
        break;
```

single-Command ::=

Identifier

(

:=

Expression

|

(

Expression

)

)

```
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
    }
}
```

```
begin
Command
end
```

```
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 → SNE
S → + | - | ε
N → I.I | I. | .I
I → dI | d
E → eSI | ε
```

sobre o alfabeto Σ={d,e,+,-,.}.

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.**

(+|-|ε)(dd*.dd*|dd*.|.dd*)(e(+|-|ε)dd*|ε)

(+|-|ε)(dd*.d*|.dd*)(e(+|-|ε)dd*|ε)

```
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();
```

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

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).