

Acetatos da Cadeira de  
Linguagens de Programação II  
(3<sup>a</sup> versão 1996/1997)

A. Miguel Dias

Linguagens de Programação II  
Licenciatura em Engenharia Informática  
Faculdade de Ciências e Tecnologia  
Universidade Nova de Lisboa

# Linguagens de Programação II

## Lic. Eng. Informática (FCT/UNL)

### Ano lectivo de 1996/1997

## Objectivos

Expandir e sistematizar os conhecimentos dos alunos na área de Linguagens de Programação. Desenvolver as capacidades de conceptualização dos alunos. Benefícios esperados: capacidade de analisar de forma articulada e fundamentada linguagens existentes ou a existir, capacidade de aprender novas linguagens mais rapidamente e de forma mais aprofundada, acrescida capacidade de concepção e desenvolvimento de software.

Esta cadeira beneficia do percurso académico anterior dos alunos, o qual inclui as cadeiras de Programação I, Programação II e Linguagens de Programação I. Assume-se um bom conhecimento das linguagens Pascal, Caml Light, Smalltalk, e algum conhecimento de C, C++ e Cobol.

## Programa

Estudo sistemático dos conceitos sintácticos, semânticos e pragmáticos fundamentais ligados às linguagens de programação e paradigmas de programação. O tratamento semântico será informal, mas rigoroso, e frequentemente ligado ao estudo de modelos de implementação. Definição de princípios orientadores do estudo comparativo de linguagens de programação. Evolução histórica das linguagens de programação. Na fase final da cadeira serão ilustrados alguns formalismos de especificação da semântica das linguagens de programação.

Detalhe:

- Critérios de avaliação de linguagens de programação.
- Das origens ao Fortran: linguagem máquina, assembler, Fortran IV.
- Semiótica: sintaxe, semântica, pragmática. Categorias sintácticas. Sintaxe abstracta e Sintaxe concreta. Formalismos de descrição sintáctica. Semântica estática e semântica dinâmica.
- Conceitos básicos: ligação, estado, memória, ambiente, l-value, r-value, escopo de uma ligação, tempo de ligação, aliasing, transparência referencial.
- Tipos: sistemas de tipos, coerções, conversões, regras de equivalência de tipos, inferência de tipos, sistemas de tipos com inseguranças, tipagem forte, polimorfismo, overloading.

- Blocos, procedimentos e parâmetros: abstracção procedimental e funcional, escopo estático e dinâmico, implementação, métodos de passagem de parâmetros, macros.
- Mecanismos de estruturação de controlo: estruturas de controlo, goto, excepções.
- Mecanismos de abstracção: ocultação da informação, módulos, tipos abstractos de dados, classes. Herança: herança estática vs. herança dinâmica. Herança e subtipos. Polimorfismo de inclusão.
- Métodos de implementação de linguagens: compilação, interpretação, semi-interpretação. Da compilação à execução.
- Paradigmas de programação.
- Correção de programas: características geradoras de insegurança e características disciplinadoras. Asserções. Verificação de programas.
- Introdução às técnicas formais de especificação da semântica: semântica informal, máquinas abstractas, semântica de transições, semântica natural, semântica denotacional.

Sempre que possível, os conceitos serão discutidos e exemplificados usando linguagens em relação às quais já exista alguma familiaridade: Pascal, Caml Light, Smalltalk, C e C++. Serão ainda utilizadas outras linguagens que sejam particularmente relevantes do ponto de vista de alguma noção que esteja em discussão. Assim serão estudados aspectos parcelares das seguintes linguagens: Fortran IV, Algol 60, Modula-2, Ada, Eiffel, Sather e Java.

## Bibliografia Principal:

- **Concepts of Programming Languages** (2ª edição)  
*Robert W. Sebesta* - Benjamin Cummings 1993
- **Conceitos de Linguagens de Programação**  
*Carlo Ghezzi, Mehdi Jazayeri* - Editora Campus 1985
- Acetatos da cadeira
- Outra documentação diversa

## Regras de Avaliação

- 90%: Exame final.
- 10%: Três trabalhos de dimensão reduzida. (o enunciado destes trabalhos está incluído na lista de exercícios das práticas.)
- A frequência para apresentação a exame obtém-se automaticamente.
- O exame vale 100% para os alunos que já tenham feito a antiga cadeira de LP e se apresentem a exame para melhoria de nota.

# Critérios de Avaliação de Linguagens de Programação

**Legibilidade** - Possibilidade de através do exame de um programa, seguir a sua lógica e descobrir a presença de erros. Ligado à manutenção de programas.

- **Simplicidade:** -muitas componentes (tendência para se aprender apenas um subconjunto da linguagem), -diversas formas de definir a mesma operação
- **Ortogonalidade:** +um conjunto relativamente reduzido de primitivas pode ser combinado de um número reduzido de formas, +todas as possíveis combinações devem ser legais e significativas não existindo excepções às regras gerais, -usada de forma irrestrita conduz a grande complexidade.
- **Estruturas de controlo:** -goto
- **Estruturas de dados:** +arrays, +records, +booleanos.
- **Sintaxe:** -restrições sobre identificadores +palavras reservadas, +formas diferentes para diferentes significados

**Redigibilidade** - Possibilidade de expressar um problema de uma forma natural, sem que a atenção do programador seja desviada por detalhes ou "truques" da linguagem. Ligada à legibilidade.

- **Simplicidade:** +fazem-se menos erros
- **Ortogonalidade:** +começa-se a escrever programas mais depressa, -se todas as combinações são legais então muitos erros lógicos não são detectáveis pelo compilador.
- **Suporte para abstracção:** +Estruturas de dados complicadas e operações podem ser usadas de forma simples, através do esquecimento deliberado de detalhes.
- **Expressividade:** +Suporte directo dos conceitos associados ao paradigma de programação usado, +construções simples para operações frequentes (*++*, *for*, *and then*).

**Segurança** - Ajuda à escrita de programas que obedecem à sua especificação.

- **Type Checking:** -run-time +compile-time.
- **Tratamento de excepções:** +mecanismos que permitem a definição de medidas correctivas em caso de erros.
- **Sinonímia (*aliasing*):** -Uma mesma entidade é conhecida por dois nomes diferentes.
- **Legibilidade:** -Surgem menos erros
- **Redigibilidade:** -Surgem menos erros

## Linguagem Máquina

Não existiam alternativas ao seu uso até princípios da década de 50.

Problemas:

- instruções especificadas por meio de códigos numéricos
- utilização directa de endereços absolutos na escrita dos programas

Resultado:

- programação entediante.
- facilidade em cometer erros.
- dificuldade em inserir ou remover instruções nos programas. (ver instrução *nop*).

Estes problemas comuns a todas as linguagens máquina constituíram a principal motivação para o desenvolvimento dos assemblers.

## Pseudo-Códigos

No início da década de 50 começaram a aparecer sistemas de *Pseudo-Códigos*:

- cada sistema define uma máquina virtual com instruções mais fáceis de usar do que as instruções da máquina real.
- são usados códigos numéricos para representar as operações.
- método de implementação usado em quase todos os casos: interpretação.
- velocidade de execução: muito baixa.

Exemplo: Sistema "Speedcoding" da IBM (1954)

- operações aritméticas de três endereços:  
$$r1 = r2 * r3$$
$$r3 = r2 - r1$$
- saltos condicionais e incondicionais
- instruções de input/output formatado

## *Assemblers*

Nos assemblers podem atribuir-se nomes a códigos de operação (*menmónicas*), a constantes, e a localizações de memória (*etiquetas*).

Geralmente existe uma relação de um para um entre as instruções-máquina e as instruções do assembler. Existem no entanto assemblers mais sofisticados em que tal não se verifica.

Instrução - traduz-se em princípio directamente numa instrução máquina.

Pseudo-Instrução ou Directiva - não gera código, mas influencia o processo de tradução para linguagem máquina (ex: definição de constantes ou etiquetas de dados).

A maior parte dos problemas de implementação de um assembler relacionam-se com o processamento de etiquetas. São habitualmente usadas duas técnicas distintas:

- Assembler de 2 passos: adequado ao processamento de programas grandes em máquinas com pouca memória.
- Assembler de 1 passo (usando *backpatching*): em geral mais prático e rápido.

## FORTRAN (FORmula TRANslator)

Anunciado por John Backus da IBM em 1954 antes de ser implementado.

### Prometia:

- a eficiência dos programas escritos directamente em assembler.
- simplificação da escrita de programas através do uso de notação matemática.
- erros de codificação deixariam naturalmente de ocorrer sendo portanto eliminado todo o processo de depuração (!?).

FORTRAN I	1957
FORTRAN II	1958
FORTRAN IV	1962 <==
FORTRAN 77	1978
FORTRAN 90	1990

Os dois primeiros objectivos foram quase atingidos no *FORTRAN I* por uma equipa de 9 pessoas trabalhando durante 2 anos.

A maioria das instruções de controlo foram baseadas nas instruções dos computadores IBM704 e IBM709 (??).

# Programa em FORTRAN IV

```
DIMENSION DTA(900)
SUM = 0.0
READ 10,N
10  FORMAT(I3)
    DO 20 I=1,N
    READ 30,DTA(I)
30  FORMAT(F10.6)
    IF (DTA(I)) 25, 20, 20
25  DTA(I) = -DTA(I)
20  CONTINUE
    DO 40 I = 1,N
    SUM = SUM + DTA(I)
40  CONTINUE
    AVG = SUM/FLOAT(N)
    PRINT 50, AVG
50  FORMAT(1H , F10.6)
STOP
```

(3HOLA <- constante de Hollerith : escreve string de comprimento n; neste caso n=3)

## Instruções principais (fortran IV)

```
A= B+1
GOTO 50
GOTO (10,20,30,40), I      (goto computado)
IF (I-1) 20, 30, 40       (if aritmético)
IF(I .EQ. 2) GOTO 20
DO 20 I = 1, 30, 3
CALL FACT(5)
RETURN
CONTINUE
STOP
WRITE
READ
```

- while, repeat, if-else têm de ser simulados com o GOTO

O GOTO é muito poderoso mas torna os programas difíceis de compreender, especialmente se for usado de forma não disciplinada.

Segundo Dijkstra:

Deve existir uma correspondência simples entre a estrutura estática de um programa e a sua estrutura dinâmica.

Por outras palavras deve ser possível visualizar o comportamento de uma programa a partir da sua forma escrita.

## Problemas sintáticos do FORTRAN

- Linhas têm um formato fixo.

1-5	label
6	continuação
7-72	instrução
73-80	numeração

- Brancos são ignorados:

DO 20 I = 5,23	<=>
DO20I=5,23	<=>
D O20I=5,2 3	

- Não há palavras reservadas:

IF( IF(5) .EQ. 4 ) GOTO 40

- As variáveis não precisam de ser declaradas:

DO 20 I=23,5	(ciclo)
DO 20 I=23.5	(atribuição)

(Como as variáveis não precisam de ser declaradas este erro passa despercebido)

## Tipos de dados (fortran IV)

INTEGER, REAL, COMPLEX, LOGICAL, ARRAY

### Declarações

```
REAL A(500), C(100), D
INTEGER B(200)
EQUIVALENCE ((A(1), B(1), D), (A(150), C(1)))
```

### Subrotinas

(só há passagem de parâmetros por referência)

```
SUBROUTINE DIST (D, X, Y)
D = X-Y
IF(D .LT. 0 D = -D
RETURN
END
```

```
SUBROUTINE A (X)
COMMON /Z/ P(100), Q(200)
DIMENSION X(20)
EQUIVALENCE (X(4), P(20))
```

...

```
SUBROUTINE B
COMMON /Z/ P(100), R(200)
```

...

EQUIVALENCE, COMMON são mecanismos de controlo explícito de *aliasing*. Este pode também ocorrer com a passagem por referência.

Todos os nomes são locais às subrotinas excepto os próprios nomes das subrotinas e os nomes dos blocos *common*.

O Fortran IV tem um carácter profundamente estático: todas as ligações de variáveis (mesmo locais) a localizações e a tipos de dados são estáticas.

## Implementação de subrotinas e funções em FORTRAN IV

Existe um registo de activação (numa zona de memória fixa [recursividade?]) associado a cada subrotina contendo:

- Loc - variáveis locais
- Par - parâmetros
- Ret - endereço de retorno
- DL - dynamic link
- Tmp - para salvaguarda de registos

A: B(P1,P2)

-----

CallFromA:

B.Par[1] = &P1 (*passagem por referência*)

B.Par[2] = &P2

B.Ret = resume

B.DL = &A

ProgCounter = Code(B)

resume:

-----

Enter B:

SaveRegs(B.Tmp)

...

ReturnFromB:

RestoreRegs(B.Tmp)

B.DL.Res = Result

ProgCounter = B.Ret

## Exemplo de Função em FORTRAN IV

### *PRODUTO VECTORIAL*

```
FUNCTION PRVE(B, C)
REAL B(100), C(100)
PRVE = 0.0
DO 20 I = 1, 100
PRVE = PRVE + B(I) * C(I)
20 CONTINUE
RETURN
END
```

## FORTRAN 90

(Standard ANSI X3.198-1992)

Inclui:

- Variáveis dinâmicas e apontadores
- Recursividade
- Módulos
- Estruturas de controlo

## Semiótica das linguagens

Semiótica =  
Sintaxe + Semântica + Pragmática

**Sintaxe** - estudo/descrição da *forma* das construções de uma linguagem. Usa-se um conjunto de regras formais que especificam a composição de programas a partir de letras, dígitos e outros símbolos. Estas são regras de estruturação de símbolos que deliberadamente ignoram o seu *significado*.

Formalismos mais usados: BNF, EBNF, diagramas sintáticos (para a sintaxe básica), gramáticas de atributos (para a semântica estática.)

**Semântica** - estudo/descrição do *significado* das construções sintacticamente válidas da linguagem. Cada construção da linguagem é aplicada num domínio que se assume conhecido. (Ver exemplo das expressões aritméticas.)

- Descrições *informais* em linguagem natural são úteis para dar uma visão intuitiva de uma linguagem de programação; no entanto estas descrições são geralmente ambíguas e incompletas. É preferível o recurso a métodos *formais*.

Abordagens formais mais usadas: semântica operacional, semântica denotacional, semântica axiomática, semântica natural.

**Pragmática** - considerações práticas sobre diversos aspectos das linguagens:

- utilidade dos mecanismos,
- facilidade em escrever e manter programas,
- eficiência,
- ...

Sintaxe =

Sintaxe básica + Semântica estática

Sintaxe básica

- Parte da sintaxe que pode ser descrita de forma simples usando um formalismo do tipo BNF.

Semântica estática

- Parte da sintaxe que não pode ser descrita em BNF de forma simples + parte da sintaxe que é impossível de descrever em BNF. Em muitos casos envolve aspectos de natureza contextual.

- Exemplos de regras de semântica estática:

- "uma variável tem de ser definida antes de ser usada",
- "uma variável não pode ser definida duas vezes".
- "numa atribuição, a variável que recebe o valor tem de ter um tipo compatível com a expressão que lhe é atribuída".
- "o comprimento máximo de um identificador é 30".

-> Os erros detectados em tempo de compilação são os erros de sintaxe básica + os erros de semântica estática.

Semântica dinâmica (ou só *semântica*)

- Descrição da linguagem que capta todos os efeitos da execução dos programas.
- Erros semânticos dinâmicos só podem ser detectados em tempo de execução.

- Exemplos de erros semânticos:

- divisão por zero,
- acesso a um vector com o índice fora dos limites,
- atribuição a uma variável *subrange*, de um valor fora dos limites,
- envio de mensagem para um objecto que não lhe possa responder (Smalltalk),
- tentativa de ler de um ficheiro não inicializado para leitura.

# Linguagem

=

Conjunto de cadeias de símbolos  
pertencente a um alfabeto

- **Alfabeto** = conjunto de símbolos
- **Palavra** = cadeia de símbolos
- **Linguagem** = conjunto de palavras
  - em **Pascal** uma *palavra* é um programa completo.
  - em **Português** uma *palavra* é uma frase completa, ou uma sequência de frases.
  - na **linguagem das expressões aritméticas** uma *palavra* é uma expressão válida.

## Exemplo: Linguagem **Pascal**

Alfabeto: {'A', ..., 'Z', 'a', ..., 'z', '0', ..., '9', '+', '-', ...}

"program ola ; begin end." ∈ **Pascal**

"program ola(output) ; begin write('ola') end." ∈ **Pascal**

"program ola(output) ; begin write(a) end." ∉ **Pascal**

"while true do writeln('ola')" ∉ **Pascal**

"+while-45 /%ola" ∉ **Pascal**

# Léxico -> Linguagem

Dada qualquer linguagem baseada em texto, a sua sintaxe pode ser descrita directamente a partir dos caracteres elementares: das letras, dos algarismos, dos sinais de pontuação, etc. Neste caso considera-se que o alfabeto da linguagem é o conjunto dos caracteres da linguagem.

No entanto, geralmente, este tipo de descrição não é muito prático. Muitas vezes é preferível descrever a sintaxe de uma linguagem em dois níveis:

- O nível inferior define uma linguagem cujas palavras são muito compactas e têm uma estrutura muito simples. Esta linguagem chama-se **léxico** e as suas palavras **lexemas**. Uma classe de lexemas chama-se **token**.
- O nível superior define a linguagem principal e usa os lexemas como símbolos.

Portanto as *palavras* do nível inferior (os lexemas) são consideradas os *símbolos* no nível superior.

Exemplos:

- lexemas do **Pascal**: "while", "23.45", ":="
- lexemas do **Português**: "ola", "consuetudinariamente", "ábaco", "zurzir".
- lexemas das **expressões aritméticas**: "+", "34.56", "394.056", "-".

Especificação da sintaxe básica:

## Gramáticas independentes de contexto

Em geral, uma **gramática** é uma meta-linguagem que permite especificar a sintaxe de linguagens (por exemplo de linguagens de programação.) Um caso particular são as **gramáticas independentes de contexto**, as quais são especialmente simples de usar sendo por isso muito populares.

Uma **gramática livre de contexto** é constituída por quatro partes:

- Um conjunto de símbolos terminais
- Um conjunto de símbolos não-terminais
- Um conjunto de regras (ou produções) da forma:  
  
símbo\_terminal ::= sequência de símbolos (terminais ou não)
- Um símbolo não-terminal que é escolhido como símbolo de partida.

-> Os **símbolos terminais** são os tokens da linguagem: Exs. *while*, *identificador*, *inteiro*, +, etc.

-> Os **símbolos não-terminais** são novos símbolos que são introduzidos para representarem **abstracções sintácticas** (chamadas **categorias sintácticas**). Exs: *programa*, *declaração*, *expressão*, *atribuição*, *instrução*, ....

-> O **símbolo de partida** funciona como *semente* de toda a linguagem.

Uma gramática é um mecanismo gerador de palavras. Cada palavra é gerada por aplicação sucessiva de produções a partir do símbolo de partida. Chama-se **derivação** a cada uma destas gerações em particular.

Notação para gramáticas:

# Gramáticas BNF (Backus-Naur Form)

Exemplo: Linguagem Lilith

```

<prog> ::=      begin <stmt_lst> end
<stmt_lst> ::=  <stmt>
                | <stmt> ; <stmt_lst>
<stmt> ::=      id := <exp>
<exp> ::=       <exp>+<exp>
                | <exp>-<exp>
                | <exp>*<exp>
                | <exp>/<exp>
                | <s_exp>
                | -<s_exp>
<s_exp> ::=     (<exp>) | id | int
Símbolo de partida: <prog>

```

Exemplo de derivação de um programa (*derivação mais à esquerda* neste caso):

```

<prog>  => begin <stmt_lst> end
        => begin <stmt> end
        => begin id := <exp> end
        => begin id := <exp>+<exp> end
        => begin id := <s_exp>+<exp> end
        => begin id := int +<exp> end
        => begin id := int +<s_exp> end
        => begin id := int + id end

```

(No final da derivação só há símbolos terminais.)

Substituindo os tokens `id` e `int` por lexemas concretos obtém-se, por exemplo, o seguinte programa:

**begin A := 12 + B end**

# Árvore de parsing

As **árvores de parsing** (*parse trees*) exibem a estrutura das palavras depois de terem sido geradas por uma gramática ou reconhecidas por um parser.

**Palavra ambígua:**

Palavra para as quais existem duas ou mais árvores de parsing distintas.

Exemplo: na linguagem Lilith a palavra  $2+3*4$  é ambígua.

**Gramática ambígua:**

Gramática que gera pelo menos uma palavra ambígua.

A seguinte gramática das expressões não é ambígua. Para cada palavra da linguagem ela gera uma árvore de parsing única cuja estrutura segue as regras tradicionais de prioridade e associatividade dos operadores:

$\langle \text{exp} \rangle ::=$	$\langle \text{exp} \rangle + \langle \text{term} \rangle$
	$\langle \text{exp} \rangle - \langle \text{term} \rangle$
	$\langle \text{term} \rangle$
$\langle \text{term} \rangle ::=$	$\langle \text{term} \rangle * \langle \text{factor} \rangle$
	$\langle \text{term} \rangle / \langle \text{factor} \rangle$
	$\langle \text{factor} \rangle$
$\langle \text{factor} \rangle ::=$	$\langle \text{s\_exp} \rangle \mid -\langle \text{s\_exp} \rangle$
$\langle \text{s\_exp} \rangle ::=$	$(\langle \text{exp} \rangle) \mid \text{id} \mid \text{int}$

Outras notações para gramáticas:

## EBNF -Extended BNF

Exemplo: Linguagem **Lilith** em EBNF

```

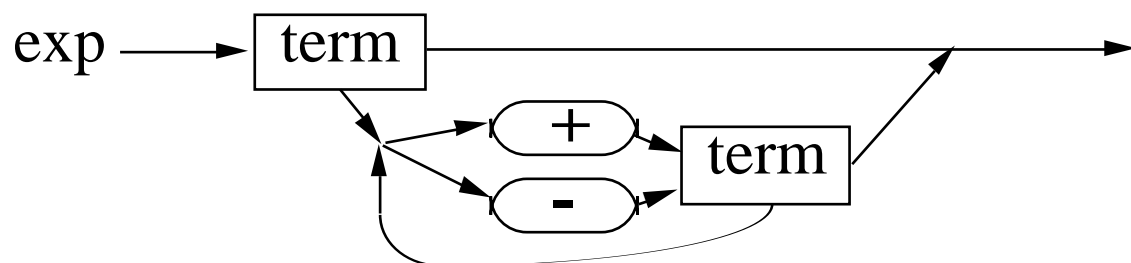
<prog> ::= begin <stmt> { ; <stmt> } end
<stmt> ::= id := <exp>
<exp> ::= <term> { (+|-) <term> }
<term> ::= <factor> { (*|/) <factor> }
<factor> ::= [-] (( <exp> ) | id | int)
Símbolo de partida: <prog>
  
```

A notação EBNF estende a notação BNF para permitir formulações mais compactas e legíveis.

## Diagramas sintácticos

- Caixas: símbolos não-terminais
- Balões: símbolos terminais
- Define-se um grafo independente para cada categoria sintáctica.

Exemplo:



-> BNF, EBNF e Diagramas sintácticos são equivalentes em termos de poder expressivo.

## Sintaxe Concreta vs. Sintaxe Abstracta

Sintaxe concreta:

Define quais as palavras de uma linguagem com todos os detalhes sintáticos.

Sintaxe abstracta:

Ignora a detalhes notacionais particulares de cada linguagem. Capta a estrutura das palavras e identifica as suas subcomponentes.

Exemplo:

Sintaxe concreta em Pascal:

```
for i := 1 to 10 do  
  writeln(i)
```

Sintaxe concreta em C:

```
for(i = 1 ; i < 10 ; i++)  
  printf("%d\n", i) ;
```

Sintaxe abstracta:

```
For(i,1,10,Writeln(i))
```

- > A sintaxe abstracta pode ser vista como uma notação para árvores de parsing.
- > A semântica dinâmica dos programas define-se por norma com base em sintaxe abstracta.

# Parser

Mecanismo que reconhecedor de linguagens. Dada uma palavra, um parser indica se essa palavra pertence ou não à linguagem.

Para qualquer gramática independente de contexto, é possível produzir algoritmicamente um reconhecedor (parser) para a linguagem gerada pela gramática (isto será estudado na cadeira de LFA.)

Qualquer compilador inclui um parser como componente.

====\*

Especificação da semântica estática:

## Gramáticas de atributos

A sintaxe da generalidade das linguagens de programação não pode ser descrita usando simples gramáticas independente do contexto.

De facto a componente de semântica estática da sintaxe exige um mecanismo de descrição mais poderoso: por exemplo, gramáticas de atributos.

Gramática de atributos:

Gramática independente de contexto enriquecida com um mecanismo que permite expressar excepções e restrições em relação à linguagem base.

As gramáticas de atributos serão estudadas na cadeira de LFA.