Compiladores

José Mendes 107188

2023

1 Tema 1 - Compiladores, Linguagens e Gramáticas

1.1 Enquadramento

1.1.1 Linguagens

Uma Linguagem pode definir-se pelas características:

1. Permite expressar, transmitir e receber ideias;
2. Comunicação entre pessoas ou seres vivos em geral;
3. Inclui a comunicação com e entre máquinas;
4. Requer várias entidades comunicantes, um código e regras para tornar a comunicação inteligível (isto é conjunto de regras comuns que os interlocutores reconheçam);

Diferentes linguagens podem ter um mesmo significado para diferentes palavras e entre estas podem ter mais do que um significado.

É necessário decidir se uma sequência de símbolos do alfabeto é válida.

Só sequências válidas é que permitem comunicação

A comunicação pode ter um efeito, sendo esta uma resposta ou o despoletar de ações.

1.1.2 Linguagens de Programação

Partilham as características das "Linguagens Naturais".

Diferem no facto de não poderem ter **ambiguidade**.

Ações despoletadas podem ser mudanças do estado computacional podendo estes estar ligado a entidades externas (como outros computadores, pessoas, ...).

Podemos defini-las por estruturas formais bem comportadas.

Nota: Desenvolvimento das linguagens de programação umbilicalmente ligado com as tecnologias de compilação!

1.2 Compiladores - Introdução

Compiladores - Compreensão, interpretação e/ou tradução automática de linguagens.

1.2.1 Processadores de Linguagens

Compiladores são programas que permitem:

1. Decidir sobre a correção de sequências de símbolos do respetivo alfabeto;
2. Despoletar acções resultantes dessas decisões.

Frequentemente "limitam-se" a fazer a tradução entre linguagens.

Nota: É o caso dos compiladores das linguagens de programação de alto nível (Java, C++, Eiffel, etc.), que traduzem o código fonte dessas linguagens em código de linguagens mais próximas do hardware do sistema computacional (e.g. assembly ou Java bytecode).

Nota: Nestes casos, na inexistência de erros, é gerado um programa composto por código executável direta ou indiretamente pelo sistema computacional:

Exemplo:

```java
public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello!");
  }
}
```

```
javac Hello.java
javap -c Hello.class
```

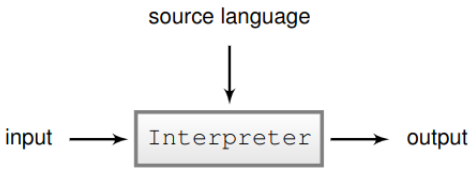
```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3  // String Hello!
       5: invokevirtual #4  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

- Código fonte:
-
```
1+2*3:4
```

- Uma possível compilação para Java:

```java
public class CodeGen {
   public static void main(String[] args) {
      int v2 = 1;
      int v5 = 2;
      int v6 = 3;
      int v4 = v5 * v6;
      int v7 = 4;
      int v3 = v4 / v7;
      int v1 = v2 + v3;
      System.out.println(v1);
   }
}
```

1.2.2 Interpretadores



São uma possível variante.

Neste caso a execução é feita instrução a instrução (ex: python, bash, ...).

Existem também aproximações hibridas em que existe compilação de código para uma linguagem intermédia, que depois é interpretada na execução.

A linguagem Java utiliza uma estratégia deste género em que o código fonte é compilado para Java bytecode, que depois é interpretado pela máquina virtual Java.

Nota: Em geral os compiladores processam código fonte em formato de texto, havendo uma grande variedade no formato do código gerado (texto, binário, interpretado, ...).

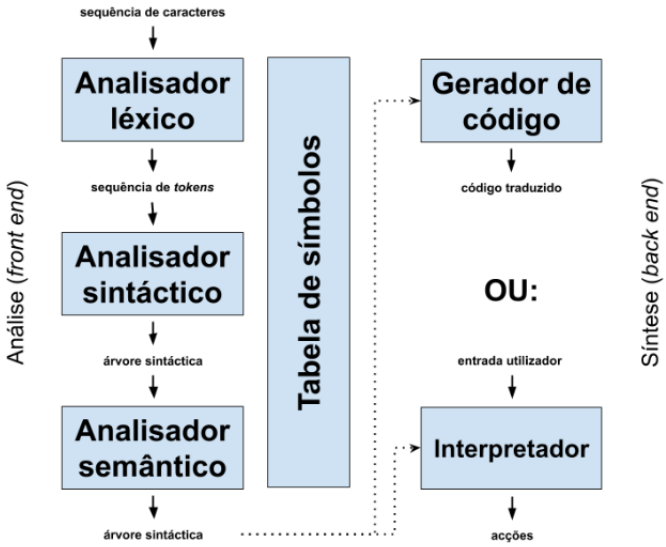
- Código fonte:

```
1+2*3:4
```

- Uma possível interpretação:

```
2.5
```

1.3 Estrutura de um Compilador



Uma característica interessante da compilação de linguagens de alto nível, é o facto de, tal como no caso das linguagens naturais, essa compilação envolver **mais do que uma linguagem**:

- **Análise Léxica:** composição de letras e outros caracteres em palavras (tokens);
- **Análise Sintática:** composição de tokens numa estrutura sintática adequada;
- **Análise Semântica:** verificação se a estrutura sintática tem significado.

As ações consistem na geração do programa na linguagem destino e podem envolver também diferentes fases de geração de código e otimização.

1.3.1 Análise Lexical

Conversão da sequência de caracteres de entrada numa sequência de elementos lexicais.

Simplifica brutalmente a gramática da análise sintática e permite uma implementação mais eficiente do analisador léxico.

Cada elemento lexical pode ser definido por um tuplo com uma identificação do elemento e do seu valor (podendo ser omitido quando N/A).

$$< \textbf{token\_name, attribute\_value} >$$

Exemplo:

pos = pos + vel * 5;

pode ser convertido pelo analisador léxico (scanner) em:
$< id, pos >< = >< id, pos >< + >< id, vel >< * >< int, 5 >< ; >$

Nota: Em geral os espaços em branco, e as mudanças de linha e os comentários não são relevantes nas linguagens de programação, pelo que podem ser eliminados pelo analisador lexical.

Exemplo:

distance (0 , 0) (4 , 3)

pode ser convertido pelo analisador léxico (scanner) em:
$< distance >< (>< num, 0 >< ,>< num, 0 ><) >< (>< num, 4 >< ,>< num, 3 ><) >$

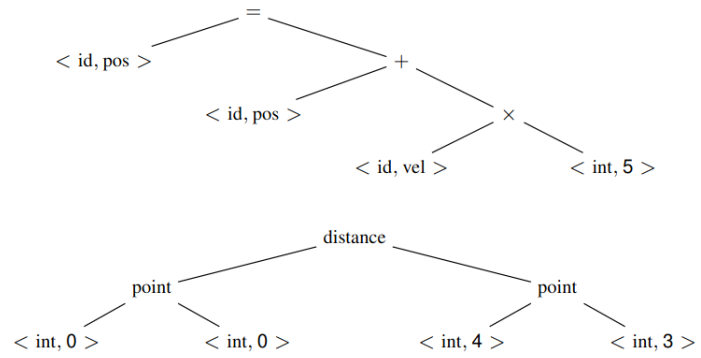
1.3.2 Análise Sintática

Após a análise lexical segue-se a chamada análise sintática (**parsing**), onde se verifica a conformidade da sequência de elementos lexicais (**tokens**) com a estrutura sintática da linguagem.

Nas linguagens que se pretende sintaticamente processar, podemos sempre fazer uma aproximação à sua estrutura formal através duma representação tipo árvore (**árvore sintática**).

Para esse fim é necessário uma gramática que especifique a estrutura desejada.

Exemplos:



Nota: Duas importantes características das árvores sintáticas são:

1. Não incluem alguns elementos lexicais (apenas relevantes para a sua estrutura formal);
2. Definem, **sem ambiguidade**, a ordem das operações.

1.3.3 Análise Semântica

Parte final do "front end" do compilador.

São verificadas todas as restantes restrições que não é possível, ou desejável, verificar nas fases anteriores.

Exemplo: verificar se um identificador foi declarado, verificar a conformidade no sistema de tipos da linguagem, etc.

Note-se que apenas restrições com verificação estática (i.e. em tempo de compilação), podem ser objeto de análise semântica pelo compilador.

Se no exemplo 2 existisse a instrução de um círculo do qual fizesse parte a definição do seu raio, não seria em geral possível, durante a análise semântica, garantir um valor não negativo para esse raio (essa semântica apenas poderia ser verificada dinamicamente, i.e., em tempo de execução).

Utiliza a árvore sintática da análise sintática assim como uma estrutura de dados designada por **tabela de símbolos** (assente em arrays associativos).

Esta última fase de análise deve garantir o sucesso das fases subsequentes (geração e eventual optimização de código, ou interpretação).

1.3.4 Síntese

Havendo garantia de que o código da linguagem fonte é válido, então podemos passar aos efeitos pretendidos com esse código.

Os efeitos podem ser:

1. simplesmente a indicação de validade do código fonte;
2. a tradução do código fonte numa linguagem destino;
3. ou a interpretação e execução imediata.

Em todos os casos, pode haver interesse na identificação e localização precisa de eventuais erros.

Como a maioria do código fonte assenta em texto, é usual indicar não só a instrução mas também a linha onde cada erro ocorre.

1.3.5 Geração de código: exemplo

Como dito antes, no processo de compilação, pode haver o interesse em gerar uma representação intermédia do código que facilite a geração final de código.

Uma forma possível para essa representação intermédia é o chamado **código de triplo endereço**.

Por exemplo para o exemplo 1 (**pos = pos + vel * 5;**):

```
t1 = inttofloat(5)
t2 = id(vel) * t1
t3 = id(pos) + t2
id(pos) = t3
```

Este código poderia depois ser optimizado na fase seguinte da compilação:

```
t1 = id(vel) * 5.0
id(pos) = id(pos) + t1
```

E por fim, poder-se-ia gerar assembly (pseudo-código):

```
LOAD  R2, id(vel)    // load value from memory to register R2
MULT  R2, R2, #5.0   // mult. 5 with R2 and store result in R2
LOAD  R1, id(pos)    // load value from memory to register R1
ADD   R1, R1, R2     // add R1 with R2 and store result in R1
STORE id(pos), R1    // store value to memory from register R1
```

1.4 Linguagens: Definição como Conjunto

As linguagens servem para **comunicar**.

Uma mensagem pode ser vista como uma sequência de **símbolos**.

No entanto, uma linguagem não aceita todo o tipo de símbolos e de sequências

Uma linguagem é caracterizada por um conjunto de símbolos e uma forma de descrever sequências válidas desses símbolos (i.e. o conjunto de sequências válidas).

Se as linguagens naturais admitem alguma subjetividade e ambiguidade, as linguagens de programação requerem total objetividade.

Como definir linguagens de forma sintética e objetiva?

Definir por **extensão** é uma possibilidade.

No entanto, para linguagens minimamente interessantes não só teríamos uma descrição gigantesca como também, provavelmente, incompleta.

As linguagens de programação tendem a aceitar variantes infinitas de entradas.

Alternativamente podemos descrevê-la por **compreensão**.

Uma possibilidade é utilizar os formalismos ligados à definição de **conjuntos**.

1.5 Conceitos Básicos e Terminologia

Um conjunto pode ser definido por **extensão** (ou enumeração) ou por **compreensão**.

Um exemplo de um conjunto definido por extensão é o conjunto dos algarismos binários {0, 1}.

Na definição por compreensão utiliza-se a seguinte notação: $\{x \mid p(x)\}$ **ou** $\{x \ : \ p(x)\}$

x é a variável que representa um qualquer elemento do conjunto, e $p(x)$ um predicado sobre essa variável.

Assim, este conjunto é definido contendo todos os valores de x em que o predicado $p(x)$ é verdadeiro.

Exemplo: $\{n \mid n \ \in \ \mathbb{N} \ \wedge n \leq 9\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Símbolo (ou **Letra** -) é a unidade atómica (indivisível) das linguagens.

Em linguagens assentes em texto, um símbolo será um carácter.

Alfabeto - é um conjunto finito não vazio de símbolos.

Exemplo: $A = \{0, 1\}$ é o alfabeto dos algarismos binários.
$A = \{0, 1, \ldots, 9\}$ é o alfabeto dos algarismos decimais.

Palavra - (string ou cadeia) é uma sequência de símbolos sobre um dado alfabeto A.
$U = a_1 a_2 \ldots a_n$, com $a_i \in A \ \wedge n \geq 0$

$A = \{0, 1, \ldots, 9\}$ é o alfabeto dos algarismos decimais. 2016, 234523, 999999999999999, 0
$A = \{0, 1, \ldots, 0, a, b, \ldots, z, @, \ldots\}$ mos@ua.pt, Bom dia!

Palavra vazia é uma sequência de zero símbolos e denota-se por ε (épsilon).

Note que ε não pertence ao alfabeto.

Uma **Sub-palavra** de uma palavra u é uma sequência contígua de 0 ou mais símbolos de u.

Um **Prefixo** de uma palavra u é uma sequência contígua de 0 ou mais símbolos iniciais de u.

Um **Sufixo** de uma palavra u é uma sequência contígua de 0 ou mais símbolos terminais de u.

Exemplo:

- *as* é uma sub-palavra de *casa*, mas não prefixo nem sufixo;
- 001 é prefixo e sub-palavra de 00100111 mas não é sufixo;
- ε é prefixo, sufixo e sub-palavra de qualquer palavra u;
- qualquer palavra u é prefixo, sufixo e sub-palavra de si própria

O **Fecho (ou conjunto de cadeias)** do alfabeto A denominado por A^*, representa o conjunto de todas as palavras definíveis sobre o alfabeto A, incluindo a palavra vazia.

Exemplo: $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

Dado um alfabeto A, uma **linguagem** L sobre A é um conjunto finito ou infinito de palavras consideradas válidas definidas com símbolos de A (isto é $L \subseteq A^*$).

Exemplo de linguagens sobre o alfabeto $A = \{0,1\}$:

- $L_1 = \{u \mid u \in A^* \wedge |u| \leq 2\} = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$
- $L_2 = \{u \mid u \in A^* \wedge \forall_i\ u_i = 0\} = \{\varepsilon, 0, 00, 000, 0000, \dots\}$
- $L_3 = \{u \mid u \in A^* \wedge u.count(1)\ mod\ 2 = 0\} = \{000, 11, 000110101, \dots\}$
- $L_4 = \{\} = \emptyset$ (conjunto vazio)
- $L_5 = \{\varepsilon\}$
- $L_6 = A$
- $L_7 = A^*$

Note que $\{\}$, $\{\varepsilon\}$, A, A^*, são linguagens sobre o alfabeto A qualquer que seja A.

Uma vez que as linguagens são conjuntos, todas as operações matemáticas sobre conjuntos são aplicáveis: reunião, intercepção, complemento, diferença, ...

1.6 Operações sobre palavras

O **comprimento** de uma palavra u denota-se por $|u|$ e representa o seu número de símbolos.

O comprimento da palavra vazia é zero: $|\varepsilon| = 0$.

É habitual interpretar-se a palavra u como uma função de acesso aos seus símbolos (tipo "array" de símbolos): $u : \{1, 2, \dots, n\} \to A$, com $n = |u|$ (em que u_i representa o iésimo símbolo de u).

O **reverso** de uma palavra u é a palavra, denota-se por u^R , e é obtida invertendo a ordem dos símbolos de u, $u = \{u_1, u_2, \ldots, u_n\} \Longrightarrow u^R = \{u_n, \ldots, u_2, u_1\}$

A **concatenação (ou produto)** das palavras u e v denota-se por $u.v$, ou simplesmente uv, e representa a justaposição de u e v, i.e, a palavra constituída pelos símbolos de u seguidos pelos símbolos de v.

Propriedades da concatenação:

- $|u.v| = |u| + |v|$
- $u.(v.w) = (u.v).w = u.v.w$ (associatividade)
- $u.\varepsilon = \varepsilon.u = u$ (elemento neutro)
- $u \neq \varepsilon \wedge v \neq \varepsilon \wedge u \neq v \Longrightarrow u.v \neq v.u$ (não comutativo)

A **potência** de ordem n, com $n \leq 0$, de uma palavra u denota-se por um u^n e representa a concatenação de n réplicas de u, ou seja, $\underbrace{uu\ldots u}_{\text{nx}}$

$u^0 = \varepsilon$

1.7 Operações sobre linguagens

1.7.1 Reunião

Denota-se por: $L_1 \cup L_2$ e é dada por: $L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$

Se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

- $L_1 = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$
- $L_2 = \{u \mid u \text{ termina por } a\} = \{wa \mid w \in A^*\}$

$L_1 \cup L_2 = \{w_1\ a\ w_2 \mid , w_1, w_2 \in A^* \wedge (w_1 = \varepsilon \vee w_2 = \varepsilon)\}$

1.7.2 Interceção

Denota-se por: $L_1 \cap L_2$ e é dada por: $\{u \mid u \in L_1 \wedge u \in L_2\}$

Usando o mesmo exemplo de cima: $L_1 \cap L_2 = \{a\ w\ a \mid w \in A^*\} \cup \{a\}$
(esta última parte é necessário pois mesmo que $w = \varepsilon$, a seria seguido por outro, aa)

1.7.3 Diferença

Denota-se por: $L_1 - L_2$ e é dada por: $\{u \mid u \in L_1 \wedge u \notin L_2\}$

Usando o mesmo exemplo de cima: $L_1 - L_2 = \{a\ w\ x \mid w \in A^* \wedge x \in A \wedge x \neq a\}$
ou
$L_1 - L_2 = \{a\ w\ b \mid w \in A^*\}$

1.7.4 Complementação

Denota-se por: $\overline{L}$ e é dada por: $A^* - L = \{u \mid u \notin L\}$
Usando o mesmo exemplo de cima: $\overline{L_1} = \{x\ w \mid w \in A^* \wedge x \in A \wedge x \neq a\} \cup \{\varepsilon\}$
ou
$\overline{L_1} = \{b\ w \mid w \in A^*\} \cup \{\varepsilon\}$

1.7.5 Concatenação

Denota-se por: $L_1.L_2$ e é dada por: $\{uv \mid u \in L_1 \wedge v \in L_2\}$
Usando o mesmo exemplo de cima: $L_1.L_2 = \{a\ w\ a \mid w \in A^*\}$

1.7.6 Potenciação

Denota-se por: L^n e é dada individualmente por:
$\begin{cases} L^0 = \{\varepsilon\} \\ L^{n+1} = L^n.L \end{cases}$
Usando o mesmo exemplo de cima: $L_1^2 = \{a\ w_1\ a\ w_2 \mid w_1, w_2 \in A^*\}$

1.7.7 Fecho de Kleene

Denota-se por: L^* e é dado por:

$$L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$$

Usando o mesmo exemplo de cima: $L_1^* = L_1 \cup \{\varepsilon\}$
Note que para $n > 1$ $L_1^n \subset L_1$

1.7.8 Notas adicionais

Note que nas operações binárias sobre conjuntos não é requerido que as duas linguagens estejam definidos sobre o mesmo alfabeto.

Assim se tivermos duas linguagens L_1 e L_2 definidas respetivamente sobre os alfebetos A_1 e A_2, então o alfabeto resultante da aplicação duma qualquer operação binária sobre as linguagens é: $A_1 \cup A_2$

1.8 Introdução às gramáticas

1.8.1 Gramáticas

A utilização de conjuntos para definir linguagens não é frequentemente a forma mais adequada e versátil para as descrever.

Muitas vezes é preferível identificar estruturas intermédias, que abstraem partes ou subconjuntos importantes, da linguagem.

Tal como em programação, muitas vezes descrições recursivas são bem mais simples, sem perda da objetividade e do rigor necessários.

É nesse caminho que encontramos as **gramáticas**.

As **gramáticas** descrevem linguagens por compreensão recorrendo a representações **formais** e (muitas vezes) **recursivas**.

Vendo as linguagens como sequências de símbolos (ou palavras), as gramáticas definem formalmente as sequências **válidas**.

Exemplo: Em português a frase "O cão ladra" pode ser gramaticalmente descrita por:

$$\text{frase} \rightarrow \text{sujeito predicado}$$
$$\text{sujeito} \rightarrow \text{artigo substantivo}$$
$$\text{predicado} \rightarrow \text{verbo}$$
$$\text{artigo} \rightarrow \textbf{O} \mid \textbf{Um}$$
$$\text{substantivo} \rightarrow \textbf{cão} \mid \textbf{lobo}$$
$$\text{verbo} \rightarrow \textbf{ladra} \mid \textbf{uiva}$$

Esta gramática descreve 8 possíveis frases e contém mais informação do que a frase original.

Contém 6 **símbolos terminais** e 6 **símbolos não terminais**.

Um símbolo não terminal é definido por uma **produção** descrevendo possíveis representações desse símbolo, em função de símbolos terminais e/ou não terminais.

Formalmente, uma gramática é um quádruplo G = (T , N, S, P), onde:

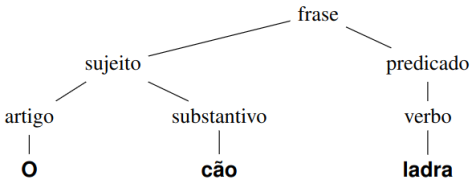
1. T é um conjunto finito não vazio designado por alfabeto terminal, onde cada elemento é designado por **símbolo terminal**;
2. N é um conjunto finito não vazio, disjunto de $T (N \cap T = \emptyset)$, cujos elementos são designados por **símbolos não terminais**;
3. $S \in N$ é um símbolo não terminal específico designado por **símbolo inicial**;
4. P é um conjunto finito de **regras** (ou **produções**) da forma:
   $\alpha \rightarrow \beta$ onde $\alpha \in (T \cup N)^* N (T \cup N)^*$ e $\beta \in (T \cup N)^*$, isto é, α é uma cadeia de símbolos terminais e não terminais contendo, pelo menos, um símbolo não terminal e β é uma cadeia de símbolos, eventualmente vazia, terminais e não terminais.

Exemplos: Formalmente, a gramática anterior será:

G = ({**O, Um, cão, lobo, ladra, uiva**}, {frase, sujeito, predicado, artigo, substantivo, verbo}, frase, P)

Em que P é constituído pelas regras já apresentadas (frase → sujeito predicado, ...)

Podemos descrever a frase "O cão ladra" com a seguinte árvore (denominada sintática).



Considere a seguinte gramática G = ({0, 1}, {S, A}, S, P), onde P é constituído pelas regras:

$$\begin{aligned} &S \to 0\ S \\ &S \to 0\ A \\ &A \to 0\ A\ 1 \\ &A \to \varepsilon \end{aligned}$$

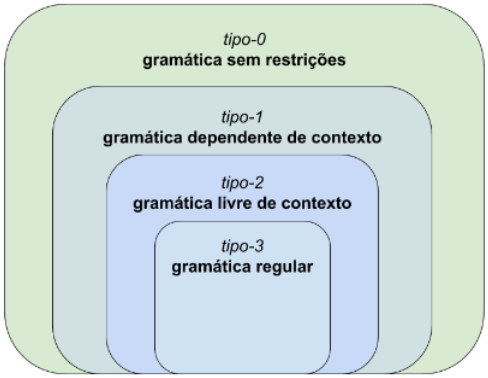
Qual será a linguagem definida por esta gramática?

$$L = \{0^n 1^m : n \in \mathbb{N} \wedge m \in \mathbb{N}_0 \wedge n > m\}$$

1.9 Hierarquia de Chomsky

Restrições sobre αe β permitem definir uma taxonomia das linguagens - hierarquia de Chomsky:

1. Se não houver nenhuma restrição, G é designada por gramática do **tipo-0**.
2. G será do **tipo-1**, ou gramática **dependente do contexto**, se se cada regra $\alpha \to \beta$ de P obedece a $|\alpha| \leq |\beta|$ (com a excepção de também poder existir a produção vazia: $S \to \varepsilon$)
3. G será do **tipo-2**, ou gramática **independente, ou livre, do contexto**, se cada regra $\alpha \to \beta$ de P obedece a $|\alpha| = 1$, isto é: α é constituído por um só não terminal.
4. G será do **tipo-3**, ou gramática **regular**, se cada regra tiver uma das formas: $A \to cB$, $A \to c$ ou $A \to \varepsilon$, onde A e B são símbolos não terminais (A pode ser igual a B) e c um símbolo terminal. Isto é, em todas as produções, o β só pode ter no máximo um símbolo não terminal sempre à direita (ou, alternativamente, sempre à esquerda)



Para cada um desses tipos podem ser definidos diferentes tipos de máquinas (algoritmos, autómatos) que as podem reconhecer.

Quanto mais simples for a gramática, mas simples e eficiente é a máquina que reconhece essas linguagens.

Cada classe de linguagens do **tipo-i** contém a classe de linguagens **tipo-(i +1)** (i = 0, 1, 2)

Esta hierarquia não traduz apenas as características formais das linguagens, mas também expressam os requisitos de computação necessários:

1. As **máquinas de Turing** processam gramáticas sem restrições (tipo-0);
2. Os **autómatos linearmente** limitados processam gramáticas dependentes do contexto (tipo-1);
3. Os **autómatos de pilha** processam gramáticas independentes do contexto (tipo-2);
4. Os **autómatos finitos** processam gramáticas regulares (tipo-3).

1.10 Autómatos

1.10.1 Máquina de Turing

(Alan Turing, 1936)

Modelo abstrato de computação.

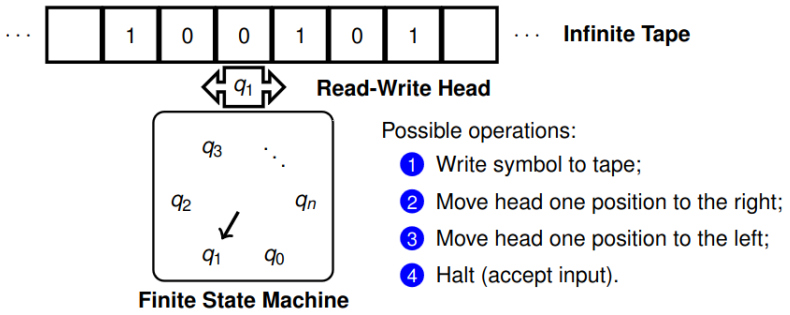
Permite (em teoria) implementar qualquer programa computável.

Assenta numa máquina de estados finita, numa "cabeça" de leitura/escrita de símbolos e numa fita infinita (onde se escreve ou lê esses símbolos).

A "cabeça" de leitura/escrita pode movimentar-se uma posição para esquerda ou direita.

Modelo muito importante na teoria da computação.

Pouco relevante na implementação prática de processadores de linguagens.



A máquina de estados finita (FSM) tem acesso ao símbolo atual e decide a próxima ação a ser realizada.

A ação consiste na transição de estado e qual a operação sobre a fita.

Se não for possível nenhuma acção, a entrada é rejeitada.

Exemplo

Dado o alfabeto A = {0, 1}, e considerando que um número inteiro não negativo n é representado pela sequência de n + 1 símbolos 1, vamos implementar uma MT que some os próximos (i.e à direita da posição actual) dois números inteiros existentes na fita (separados apenas por um 0).

O algoritmo pode ser simplesmente trocar o símbolo 0 entre os dois números por 1, e trocar os dois últimos símbolos 1 por 0.

Por exemplo: 3 + 2 a que corresponde o seguinte estado na fita (símbolo a negrito é a posição da "cabeça"):
· · · **0**111101110 · · · (o resultado pretendido será:
· · · **0**111111000 · · ·).

Considerando que os estados são designados por E_i , $i \geq 1$ (sendo E_1 o estado inicial); e as operações:

- **d** mover uma posição para a direita;
- **e** mover uma posição para a esquerda;
- **0** escrever o símbolo 0 na fita;
- **1** escrever o símbolo 1 na fita;
- **h** aceitar e terminar autómato.

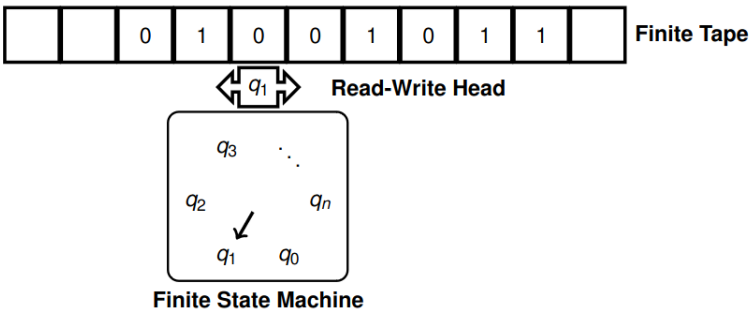
Uma solução possível é dada pela seguinte diagrama de transição de estados:

	Entrada	
Estado	**0**	**1**
E_1	E_1/d	E_2/d
E_2	$E_3/1$	E_2/d
E_3	E_4/e	E_3/d
E_4	$--$	$E_5/0$
E_5	E_5/e	$E_6/0$
E_6	E_7/e	$--$
E_7	E_1/h	E_7/e

- $E_1 \cdots \mathbf{0}111101110 \cdots \rightarrow E_1 \cdots 0111101110 \cdots \overset{*}{\rightarrow} E_2 \cdots 0111\mathbf{0}1110 \cdots \rightarrow$
  $E_3 \cdots 0111\mathbf{1}1110 \cdots \rightarrow E_3 \cdots 0111111110 \cdots \overset{*}{\rightarrow} E_3 \cdots 011111111\mathbf{0} \cdots \rightarrow$
  $E_4 \cdots 011111111\mathbf{0} \cdots \rightarrow E_5 \cdots 011111110\mathbf{0} \cdots \rightarrow E_5 \cdots 01111111\mathbf{0}0 \cdots \rightarrow$
  $E_6 \cdots 0111111\mathbf{0}00 \cdots \rightarrow E_7 \cdots 0111111000 \cdots \overset{*}{\rightarrow} E_7 \cdots \mathbf{0}111111000 \cdots$

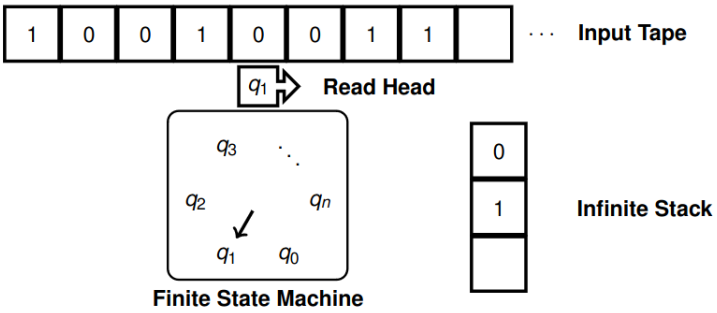
1.10.2 Autómatos linearmente limitados

Diferem das MT pela finitude da fita.



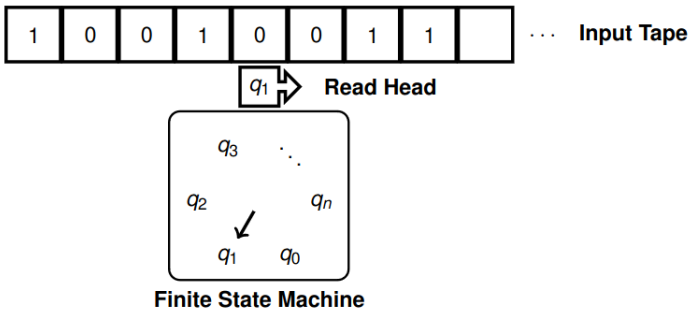
1.10.3 Autómatos de pilha

"Cabeça" apenas de leitura e suporte de uma pilha sem limites.
Movimento da "cabeça" apenas numa direção.
Autómatos adequados para análise sintática.



1.10.4 Autómatos finitos

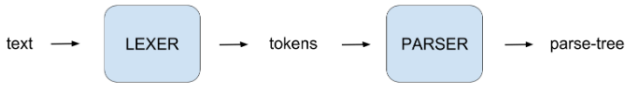
Sem escrita de apoio à máquina de estados.
Autómatos adequados para análise léxica.



2 Tema 2 - ANTLR4

ANTLR4 - ANother **T**ool for **L**anguage **R**ecognition é um gerador de processadores de linguagens que pode ser usado para ler, processar, executar ou traduzir linguagens.

2.1 Exemplos Introdutórios



Exemplo Hello:

```
// (this is a line comment)
grammar Hello;                  // Define a grammar called Hello
// parser (first letter in lower case):
r : 'hello' ID ;                // match keyword hello followed by an identifier
// lexer (first letter in upper case):
ID : [a-z]+ ;                   // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ;       // skip spaces, tabs, newlines, (Windows)
```

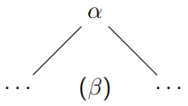
As duas gramáticas - lexical e sintática - são expressas com instruções em a seguinte estrutura: $\alpha : \beta$;

Em que α corresponde a um único símbolo lexical ou sintático (dependendo dada sua primeira letra ser, respetivamente, maiúscula ou minúscula); e em que β é uma expressão simbólica equivalente a α.

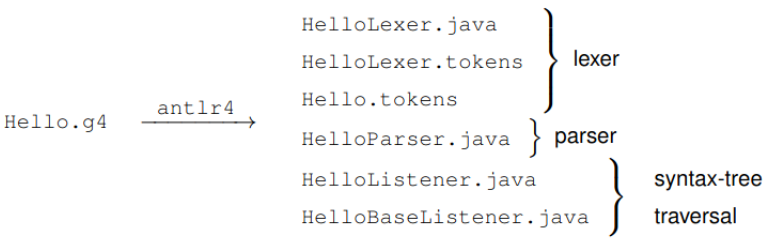
Parser - Composição de Tokens numa estrutura sintática adequada.

Lexer - Composição de letras e outros caracteres em palavras (Tokens).

Uma sequência de símbolos na entrada que seja reconhecido por esta regra gramatical pode sempre ser expressa por uma estrutura tipo árvore (chamada sintática), em que a raiz corresponde a α e os ramos à sequência de símbolos expressos em β:



Executando o comando antlr4 sobre a gramática Hello.g4 temos:



Ficheiros gerados:

- **HelloLexer.java**: código **Java** com a análise léxica (gera tokens para a análise sintática);
- **Hello.tokens** e **HelloLexer.tokens**: ficheiros com a identificação de tokens (pouco importante nesta fase, mas serve para modularizar diferentes analisadores léxicos e/ou separar a análise léxica da análise sintática);
- **HelloParser.java**: código **Java** com a análise sintática (gera a árvore sintática do programa);
- **HelloListener.java** e **HelloBaseListener.java**: código **Java** que implementa automaticamente um padrão de execução de código tipo listener (observer, callbacks) em todos os pontos de entrada e saída de todas as regras sintáticas do compilador.

Podemos executar o **ANTLR4** com a opção **-visitor** para gerar também código **Java** para o padrão tipo visitor (difere do listener porque a visita tem de ser explicitamente requerida).

HelloVisitor.java e **HelloBaseVisitor.java**: código **Java** que implementa automaticamente um padrão de execução de código tipo visitor todos os pontos de entrada e saída de todas as regras sintáticas do compilador.

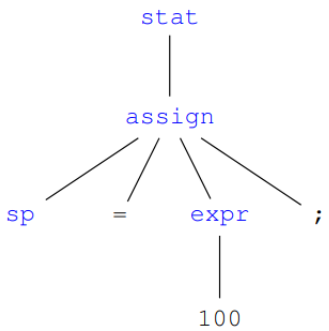
Exemplo Expr

```
grammar Expr;
stat: assign ;
assign: ID '=' expr ';' ;
expr: INT ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

Se executarmos o compilador criado com a entrada:

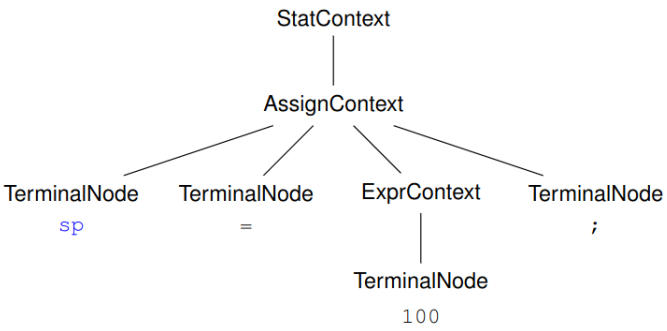
$$\mathbf{sp} = 100;$$

Vamos obter a seguinte árvore sintática:



Para facilitar a análise semântica e a síntese, o **ANTLR4** tenta ajudar na resolução automática de muitos problemas (como é o caso dos visitors e dos listeners).

No mesmo sentido são geradas classes (e em execução os respetivos objetos) com o contexto de todas as regras da gramática:




```
grammar Expr;                    → classes: ExprLexer and ExprParser
stat : assign ;                  → class StatContext in ExprParser
assign : ID '=' expr ';' ;       → class AssignContext in ExprParser
expr : INT ;                     → class ExprContext in ExprParser

ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

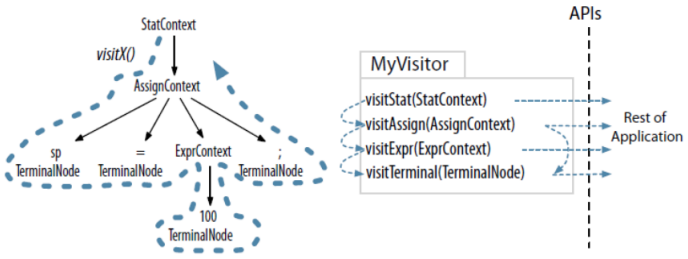
```
public class ExprParser extends Parser {
  public static class StatContext extends ParserRuleContext {
    public AssignContext assign() {
      ...
    }
    ...
  }
  ...
}
```

2.1.1 Visitor

Os objetos de contexto têm a si associada toda a informação relevante da análise sintática (tokens, referência aos nós filhos da árvore, etc.)

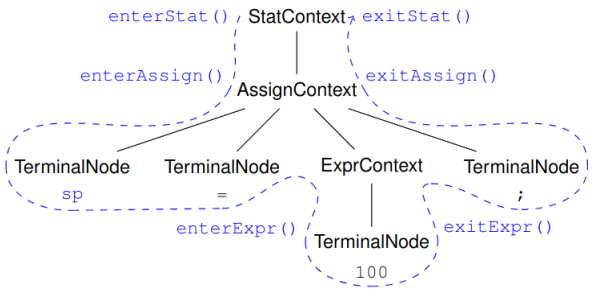
Por exemplo o contexto **AssignContext** contém métodos **ID** e **expr** para aceder aos respetivos nós.

No caso do código gerado automaticamente do tipo visitor o padrão de invocação é ilustrado a seguir:



2.1.2 Listener

O código gerado automaticamente do tipo listener tem o seguinte padrão de invocação (ordem de execução):



A sua ligação à restante aplicação é a seguinte:

2.1.3 Atributos e ações

É possível associar atributos e ações às regras:

```
grammar ExprAttr;
stat: assign ;
assign: ID '=' e=expr ';'
   {System.out.println($ID.text+" = "+$e.v);} // action
   ;
expr returns[int v]: INT  // result attribute named v in expr
   {$v = Integer.parseInt($INT.text);}        // action
   ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

Ao contrário dos visitors e listeners, a execução das ações ocorre durante a análise sintática.

A execução de cada ação ocorre no contexto onde ela é declarada. Assim se uma ação estiver no fim de uma regra (como exemplificado acima), a sua execução ocorrerá após o respetivo reconhecimento.

A linguagem a ser executada na ação não tem de ser necessariamente Java (existem muitas outras possíveis, como C++ e python).

Também podemos passar atributos para a regra (tipo passagem de argumentos para um método):

```
assign: ID '=' e=expr[true] ';' // argument passing to expr
   {System.out.println($ID.text+" = "+$e.v);}
   ;
expr[boolean a]      // argument attribute named a in expr
   returns[int v]:   // result attribute named v in expr
   INT {
      if ($a)
         System.out.println("Wow! Used in an assignment!");
      $v = Integer.parseInt($INT.text);
   } ;
```

É clara a semelhança com a passagem de argumentos e resultados de métodos.

Diz que os atributos são **sintetizados** quando a informação provém de sub-regras, e **herdados** quando se envia informação para sub-regras.

2.2 Construção de gramáticas

A construção de gramáticas pode ser considerada uma forma de programação simbólica, em que existem símbolos que são equivalentes a sequências (que façam sentido) de outros símbolos (ou mesmo dos próprios).

Os símbolos utilizados dividem-se em **símbolos terminais e não terminais**.

Os símbolos terminais correspondem a caracteres na gramática lexical e tokens na sintática; e os símbolos não terminais são definidos por produções (regras).

No fim, todos os símbolos não terminais devem poder ser expressos em símbolos terminais.

Uma gramática é construída especificando as **regras** ou produções dos elementos gramaticais.

```
grammar SetLang;        // a grammar example
stat: set set;          // stat is a sequence of two set
set: '{' elem* '}';     // set is zero or more elem inside { }
elem: ID | NUM;         // elem is an ID or a NUM
ID: [a-z]+;             // ID is a non-empty sequence of letters
NUM: [0-9]+;            // NUM is a non-empty sequence of digits
```

Sendo a sua construção uma forma de programação, podemos beneficiar da identificação e reutilização de padrões comuns de resolução de problemas.

Surpreendentemente, o número de padrões base é relativamente baixo:

- **Sequência:** sequência de elementos;
- **Optativo:** aplicação optativa do elemento (zero ou uma ocorrência);
- **Repetitivo:** aplicação repetida do elemento (zero ou mais, uma ou mais);
- **Alternativa:** escolha entre diferentes alternativas (como por exemplo, diferentes tipos de instruções);
- **Recursão:** definição direta ou indiretamente recursiva de um elemento (por exemplo, instrução condicional é uma instrução que selecciona para execução outras instruções);

Nota: A recursão e a iteração são alternativas entre si. Admitindo a existência da sequência vazia, os padrões optativo e repetitivo são implementáveis com recursão.

No entanto, como em programação em geral, por vezes é mais adequado expressar recursão, e outras iteração.

Exemplo:

```
import static java.lang.System.*;
public class PrimeList {
   public static void main(String[] args) {
      if (args.length != 1) {
         out.println("Usage: PrimeList <n>");
         exit(1);
      }
      int n = 0;
      try {
         n = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
         out.println("ERROR: invalid argument '"+args[0]+"'");
         exit(1);
      }
      for(int i = 2;i <= n;i++)
         if(isPrime(i))
            out.println(i);
   }

   public static boolean isPrime(int n) {
      assert n > 1; // precondition

      boolean result = (n == 2 || n % 2 != 0);
      for(int i = 3;result && (i*i <= n);i+=2)
         result = (n % i != 0);
      return result;
   }
}
```

Mesmo sem uma gramática definida explicitamente, podemos neste programa inferir todos os padrões atrás referidos:

- **Sequência:** a instrução atribuição de valor é definida como sendo um identificador, seguido do carácter =, seguido de uma expressão.
- **Optativo:** a instrução condicional pode ter, ou não, a seleção de código para a condição falsa.
- **Repetitivo:** (1) uma classe é uma repetição de membros; (2) um algoritmo é uma repetição de comandos.
- **Alternativa:** diferentes instruções podem ser utilizadas onde uma instrução é esperada.
- **Recursão:** diferentes instruções podem ser utilizadas onde uma instrução é esperada.

2.2.1 Especificação de gramáticas

Uma linguagem para especificação de gramáticas precisa de suportar este conjunto de padrões.

Para especificar elementos léxicos (tokens) a notação utilizada assenta em expressões regulares.

A notação tradicionalmente utilizada para a análise sintáctica denomina-se por BNF (Backus-Naur Form):

$$< symbol > \mathbf{::=} < meaning >$$

Esta última notação teve origem na construção da linguagem Algol (1960).

O ANTLR4 utiliza uma variação alterada e aumentada (Extended BNF ou EBNF) desta notação onde se pode definir construções opcionais e repetitivas.

$$< symbol > \mathbf{:} < meaning >;$$

2.3 ANTLR4: Estrutura Léxica